КЛАССИКА COMPUTER SCIENCE

COMPUTER SCIENCE

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА JAVA, ООП, АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

РОБЕРТ СЕДЖВИК
КЕВИН УЭЙН

ПИТЕР

С Е Р И Я

КЛАССИКА COMPUTER SCIENCE

Computer Science

An Interdisciplinary Approach

Robert Sedgewick
Kevin Wayne

Princeton University

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

КЛАССИКА COMPUTER SCIENCE

РОБЕРТ СЕДЖВИК, КЕВИН УЭЙН

COMPUTER SCIENCE

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА JAVA, ООП, АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2018

ББК 32.973.2-018
УДК 004.42
С28

Седжвик Р., Уэйн К.

С28 Computer Science: основы программирования на Java, ООП, алгоритмы и структуры данных. — СПб.: Питер, 2018. — 1072 с.: ил. — (Серия «Классика computer science»).

ISBN 978-5-496-02700-7

Преподаватели Принстонского университета Роберт Седжвик и Кевин Уэйн создали универсальное введение в Computer Science на языке Java, которое идеально подходит как студентам, так и профессионалам. Вы начнете с основ, освоите современный курс объектно-ориентированного программирования и перейдете к концепциям более высокого уровня: алгоритмам и структурам данных, теории вычислений и архитектуре компьютеров.

И главное — вся теория рассматривается на практических и ярких примерах: прикладная математика, физика и биология, числовые методы, визуализация данных, синтез звука, обработка графики, финансовое моделирование и многое другое.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.42

Права на издание получены по соглашению с Addison-Wesley Longman. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0134076423 англ.
ISBN 978-5-496-02700-7

© 2017 Pearson Education, Inc.
© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Классика computer science», 2018

Краткое содержание

Предисловие . . . 16

Глава 1. Основы программирования . . . 25

Глава 2. Функции и модули . . . 199

Глава 3. Объектно-ориентированное программирование . . . 326

Глава 4. Алгоритмы и структуры данных . . . 479

Глава 5. Теория вычислений . . . 687

Глава 6. Вычислительная машина . . . 838

Глава 7. Построение вычислительного устройства . . . 948

Заключение . . . 1054

Глоссарий . . . 1057

API . . . 1066

Оглавление

Предисловие **16**

От научного редактора русского издания 23

От издательства 24

Глава 1. Основы программирования **25**

1.1. Ваша первая программа 25

Вопросы и ответы 32

Упражнения 34

1.2. Встроенные типы данных 35

Выводы 57

Вопросы и ответы (строки) 57

Вопросы и ответы (целые числа) 58

Вопросы и ответы (числа с плавающей точкой) 60

Вопросы и ответы (переменные и выражения) 61

Упражнения 63

1.3. Условные переходы и циклы 68

Выводы 95

Вопросы и ответы 96

Упражнения 99

1.4. Массивы 106

Выводы 131

Вопросы и ответы 132

Упражнения 133

Упражнения повышенной сложности 136

1.5. Ввод и вывод 140

Выводы 172

Вопросы и ответы 172

Упражнения 174

Упражнения повышенной сложности 178

1.6. Пример: случайный серфинг ... 180
Выводы ... 195
Упражнения ... 196
Упражнения повышенной сложности ... 198

Глава 2. Функции и модули ... 199
2.1. Определение функций ... 200
Вопросы и ответы ... 224
Упражнения ... 225
Упражнения повышенной сложности ... 228
2.2. Библиотеки и клиенты ... 231
Вопросы и ответы ... 260
Упражнения ... 260
Упражнения повышенной сложности ... 263
2.3. Рекурсия ... 265
Выводы ... 292
Вопросы и ответы ... 293
Упражнения ... 294
Упражнения повышенной сложности ... 296
2.4. Пример: задача о протекании ... 300
Выводы ... 318
Вопросы и ответы ... 321
Упражнения ... 322
Упражнения повышенной сложности ... 323

Глава 3. Объектно-ориентированное программирование ... 326
3.1. Использование типов данных ... 327
Вопросы и ответы ... 365
Упражнения ... 369
Упражнения повышенной сложности ... 372
3.2. Создание типов данных ... 375
Вопросы и ответы ... 409
Упражнения ... 411
Упражнения повышенной сложности ... 415

3.3. Проектирование типов данных 420
Вопросы и ответы 459
Упражнения 462
Упражнения по проектированию типов данных 463
Упражнения повышенной сложности 465
3.4. Пример: моделирование задачи n тел 467
Вопросы и ответы 477
Упражнения 477
Упражнения повышенной сложности 478
Глава 4. Алгоритмы и структуры данных 479
4.1. Быстродействие 480
Выводы 504
Вопросы и ответы 505
Упражнения 507
Упражнения повышенной сложности 512
4.2. Сортировка и поиск 515
Выводы 539
Вопросы и ответы 541
Упражнения 542
Упражнения повышенной сложности 544
4.3. Стеки и очереди 547
Вопросы и ответы 590
Упражнения 593
Упражнения для связных списков 596
Упражнения повышенной сложности 597
4.4. Таблицы символов 602
Вопросы и ответы 634
Упражнения 635
Упражнения для бинарных деревьев 639
Упражнения повышенной сложности 640
4.5. Пример: феномен «тесного мира» 646
Выводы 676
Вопросы и ответы 678

Упражнения ... 679
Упражнения повышенной сложности ... 683

Глава 5. Теория вычислений ... 687

5.1. Формальные языки ... 690
Выводы ... 729
Вопросы и ответы ... 730
Упражнения ... 731
Упражнения повышенной сложности ... 734
5.2. Машины Тьюринга ... 737
Вопросы и ответы ... 752
Упражнения ... 752
Упражнения повышенной сложности ... 755
5.3. Универсальность ... 755
Вопросы и ответы ... 767
Упражнения повышенной сложности ... 768
5.4. Вычислимость ... 774
Вопросы и ответы ... 787
Упражнения ... 788
Упражнения повышенной сложности ... 789
5.5. Вычислительная сложность ... 790
Главный вопрос ... 809
Вопросы и ответы ... 827
Упражнения ... 829
Упражнения повышенной сложности ... 835

Глава 6. Вычислительная машина ... 838

6.1. Представление информации ... 839
Выводы ... 866
Вопросы и ответы ... 867
Упражнения ... 870
Упражнения повышенной сложности ... 873
6.2. Машина TOY ... 874
Вопросы и ответы ... 896
Упражнения ... 897

6.3. Программирование на машинном языке . . . 900
Вопросы и ответы . . . 915
Упражнения . . . 916
Упражнения повышенной сложности . . . 921
6.4. Виртуальная машина TOY . . . 923
Вопросы и ответы . . . 943
Упражнения . . . 944
Упражнения повышенной сложности . . . 945

Глава 7. Построение вычислительного устройства . . . 948
7.1. Булева логика . . . 949
Упражнения . . . 960
Упражнения повышенной сложности . . . 961
7.2. Базовая модель электронной схемы . . . 963
Вопросы и ответы . . . 971
Упражнения . . . 972
7.3. Комбинационные схемы . . . 973
Вопросы и ответы . . . 1004
Упражнения . . . 1004
Упражнения повышенной сложности . . . 1007
7.4. Последовательностные схемы . . . 1010
Выводы . . . 1025
Вопросы и ответы . . . 1027
Упражнения . . . 1028
Упражнения повышенной сложности . . . 1030
7.5. Цифровые вычислительные устройства . . . 1031
Вопросы и ответы . . . 1052
Упражнения . . . 1053
Упражнения повышенной сложности . . . 1053

Заключение . . . 1054

Глоссарий . . . 1057

API . . . 1066

Список листингов

Глава 1. Основы программирования ... 25

1.1. Ваша первая программа ... 25

Листинг 1.1.1. Hello, World ... 28

Листинг 1.1.2. Использование аргумента командной строки ... 31

1.2. Встроенные типы данных ... 35

Листинг 1.2.1. Конкатенация строк ... 42

Листинг 1.2.2. Целочисленное умножение и деление ... 45

Листинг 1.2.3. Вычисление корней квадратного уравнения ... 47

Листинг 1.2.4. Високосный год ... 50

Листинг 1.2.5. Преобразование типа для получения случайного целого числа ... 55

1.3. Условные переходы и циклы ... 68

Листинг 1.3.1. Бросок монетки ... 71

Листинг 1.3.2. Первый цикл while ... 73

Листинг 1.3.3. Вычисление степеней 2 ... 76

Листинг 1.3.4. Первая программа с вложенным циклом ... 81

Листинг 1.3.5. Гармонические числа ... 84

Листинг 1.3.6. Метод Ньютона ... 85

Листинг 1.3.7. Преобразование в двоичную форму ... 87

Листинг 1.3.8. Моделирование задачи о разорении игрока ... 89

Листинг 1.3.9. Разложение целых чисел на простые множители ... 91

1.4. Массивы ... 106

Листинг 1.4.1. Случайная выборка без замены ... 115

Листинг 1.4.2. Моделирование задачи о собирателе купонов ... 119

Листинг 1.4.3. Решето Эратосфена ... 121

Листинг 1.4.4. Случайные блуждания без самопересечений ... 129

1.5. Ввод и вывод ... 140

Листинг 1.5.1. Генерирование случайной последовательности ... 142

Листинг 1.5.2. Интерактивный ввод ... 150

Листинг 1.5.3. Вычисление среднего арифметического для числового потока 151
Листинг 1.5.4. Простой фильтр 153
Листинг 1.5.5. Преобразование стандартного ввода в графику 160
Листинг 1.5.6. Столкновения шара 166
Листинг 1.5.7. Цифровая обработка сигналов 171
1.6. Пример: случайный серфинг 180
Листинг 1.6.1. Пример: случайный серфинг 184
Листинг 1.6.2. Моделирование случайного серфинга 186
Листинг 1.6.3. Аналитическая модель на основе марковской цепи 193

Глава 2. Функции и модули 199

2.1. Определение функций 200
Листинг 2.1.1. Гармонические числа (обновленная версия) 202
Листинг 2.1.2. Гауссовы функции 212
Листинг 2.1.3. Задача о собирателе купонов (новая версия) 215
Листинг 2.1.4. Программа PlayThatTune (обновленная версия) 221
2.2. Библиотеки и клиенты 231
Листинг 2.2.1. Библиотека для работы со случайными числами 239
Листинг 2.2.2. Библиотека ввода/вывода массивов 244
Листинг 2.2.3. Итерационные вычисления 247
Листинг 2.2.4. Библиотека анализа данных 251
Листинг 2.2.5. Графическое представление значений данных в массиве 252
Листинг 2.2.6. Испытания Бернулли 255
2.3. Рекурсия 265
Листинг 2.3.1. Алгоритм Евклида 271
Листинг 2.3.2. Ханойские башни 273
Листинг 2.3.3. Код Грея 279
Листинг 2.3.4. Рекурсивная графика 281
Листинг 2.3.5. Броуновский мост 283
Листинг 2.3.6. Наибольшая общая подстрока 291
2.4. Пример: задача о протекании 300
Листинг 2.4.1. Служебный код Percolation 304
Листинг 2.4.2. Обнаружение вертикального протекания 306
Листинг 2.4.3. Клиент для визуализации 309
Листинг 2.4.4. Оценка вероятности протекания 310
Листинг 2.4.5. Поиск протекания 313
Листинг 2.4.6. Клиент, строящий адаптивный график 316

Глава 3. Объектно-ориентированное программирование 326

3.1. Использование типов данных 327

Листинг 3.1.1. Проверка потенциального гена 334

Листинг 3.1.2. Квадраты Альберса 340

Листинг 3.1.3. Библиотека Luminance.............................342

Листинг 3.1.4. Преобразование цветного изображения в оттенки серого. .345

Листинг 3.1.5. Масштабирование изображений........................347

Листинг 3.1.6. Эффект растворения 349

Листинг 3.1.7. Конкатенация файлов 353

Листинг 3.1.8. Извлечение данных биржевых котировок 355

Листинг 3.1.9. Разбиение данных 357

3.2. Создание типов данных 375

Листинг 3.2.1. Класс «заряженная частица» 381

Листинг 3.2.2. Stopwatch.......................................385

Листинг 3.2.3. Класс Histogram 387

Листинг 3.2.4. Черепашья графика 390

Листинг 3.2.5. Логарифмическая спираль...........................393

Листинг 3.2.6. Комплексные числа 398

Листинг 3.2.7. Множество Мандельброта 404

Листинг 3.2.8. Портфель акций 407

3.3. Проектирование типов данных 420

Листинг 3.3.1. Комплексные числа (альтернативная реализация) 426

Листинг 3.3.2. Счетчик..429

Листинг 3.3.3. Пространственные векторы 436

Листинг 3.3.4. Конспект документа................................453

Листинг 3.3.5. Выявление сходства 455

3.4. Пример: моделирование задачи n тел 467

Листинг 3.4.1. Гравитационное тело...............................470

Листинг 3.4.2. Моделирование системы n тел 474

Глава 4. Алгоритмы и структуры данных 479

4.1. Быстродействие 480

Листинг 4.1.1. Подсчет троек.....................................482

Листинг 4.1.2. Проверка гипотезы удвоения..........................485

4.2. Сортировка и поиск 515

Листинг 4.2.1. Бинарный поиск (20 вопросов) 517

Листинг 4.2.2. Обращение функции методом половинного деления 521

Листинг 4.2.3. Бинарный поиск (отсортированный массив).............522

Листинг 4.2.4. Сортировка методом вставки .529
Листинг 4.2.5. Проверка гипотезы удвоения для сортировки методом вставки .531
Листинг 4.2.6. Сортировка слиянием .534
Листинг 4.2.7. Подсчет частот вхождения .540
4.3. Стеки и очереди 547
Листинг 4.3.1. Стек строк (реализация в виде массива) .551
Листинг 4.3.2. Стек строк (связный список) .556
Листинг 4.3.3. Стек строк (изменение размера массива) .561
Листинг 4.3.4. Обобщенный стек .566
Листинг 4.3.5. Вычисление значения выражения .569
Листинг 4.3.6. Обобщенная очередь FIFO (связный список) .576
Листинг 4.3.7. Моделирование очереди M/M/1 .581
Листинг 4.3.8. Моделирование распределения нагрузки .589
4.4. Таблицы символов 602
Листинг 4.4.1. Поиск по словарю .610
Листинг 4.4.2. Индексирование .612
Листинг 4.4.3. Таблица с хешированием .618
Листинг 4.4.4. Бинарное дерево поиска .626
Листинг 4.4.5. Фильтр Dedup .634
4.5. Пример: феномен «тесного мира» 646
Листинг 4.5.1. Тип данных Graph .654
Листинг 4.5.2. Использование графа для инвертирования индекса .658
Листинг 4.5.3. Клиент для поиска кратчайших путей .662
Листинг 4.5.4. Реализация кратчайших путей .667
Листинг 4.5.5. Проверка графа «тесного мира» .673
Листинг 4.5.6. Граф «актер — актер» .674

Глава 5. Теория вычислений 687

5.1. Формальные языки 690
Листинг 5.1.1. Проверка правильности (распознавание) .700
Листинг 5.1.2. Поиск по шаблону с использованием обобщенных регулярных выражений .707
Листинг 5.1.3. Универсальный виртуальный ДКА .715
5.2. Машины Тьюринга 737
Листинг 5.2.1. Класс ленты для виртуальной машины Тьюринга .749
Листинг 5.2.2. Универсальная виртуальная машина Тьюринга .751
5.5. Вычислительная сложность 790
Листинг 5.5.1. Система решения задачи выполнимости .825

Глава 6. Вычислительная машина 838

6.1. Представление информации 839

Листинг 6.1.1. Преобразование натурального числа в другую систему счисления 847

Листинг 6.1.2. Извлечение компонентов числа в формате с плавающей точкой 863

6.2. Машина TOY 874

Листинг 6.2.1. Первая программа для машины TOY 884

Листинг 6.2.2. Условные переходы и циклы: алгоритм Евклида 891

Листинг 6.2.3. Самомодифицирующийся код: вычисление суммы 893

6.3. Программирование на машинном языке 900

Листинг 6.3.1. Вызов функции: проверка числа на простоту 903

Листинг 6.3.2. Стандартный вывод: числа Фибоначчи 905

Листинг 6.3.3. Стандартный ввод: вычисление суммы 907

Листинг 6.3.4. Обработка массива: чтение 909

Листинг 6.3.5. Связные структуры: поиск/вставка в бинарном дереве поиска 913

6.4. Виртуальная машина TOY 923

Листинг 6.4.1. Виртуальная машина TOY (без стандартного ввода и вывода) 933

Предисловие

Основой образования в прошлом тысячелетии были «чтение, письмо и арифметика», теперь это «чтение, письмо и компьютер». Изучение программирования стало неотъемлемой частью образования любого студента в области науки и техники. Кроме непосредственного практического применения, оно является первым шагом в понимании бесспорного влияния компьютерных технологий на современный мир. Эта книга была написана для того, чтобы научить программированию тех, кто хочет изучить его, в научном контексте.

Наша главная цель — вооружить читателя необходимыми инструментами и опытом для эффективного использования компьютера. Мы постараемся убедить читателя в том, что написание программ — естественное, увлекательное и творческое занятие. Мы постепенно введем основные концепции, опишем классические задачи из прикладной математики и различных дисциплин для демонстрации концепций и предоставим читателям возможность самостоятельно писать программы для решения интересных задач. Также мы постараемся снять покров тайны с компьютерных технологий и дать представление о важнейших интеллектуальных основах в области информатики.

Для написания всех программ в книге будет использоваться язык Java. В первой части книги излагаются основы решения вычислительных задач, применимые во многих современных вычислительных средах; по сути, это самостоятельный учебный курс для людей, не имеющих опыта программирования. Эта часть книги посвящена *фундаментальным концепциям программирования*, а не языку Java как таковому. Вторая часть книги показывает, что информатика вовсе не ограничивается одним только программированием, но для демонстрации основных идей в ней тоже часто используются программы на Java.

В книге применяется междисциплинарный подход к традиционному учебному курсу информатики: мы подчеркиваем роль вычислительных технологий в других дисциплинах, от материаловедения и геномики до астрофизики и сетевых систем. Этот подход подтверждает главную мысль о том, что математика, наука, техника и вычислительные технологии тесно переплетены в современном мире. И хотя перед вами учебник, ориентированный на студента-первокурсника, книга может использоваться также и для самообразования.

Краткий обзор содержания книги

Первая часть книги охватывает следующие основные этапы изучения программирования: основы программирования, функции, объектно-ориентированное про-

граммирование. Прежде чем переходить на следующий уровень, мы дадим читателю все необходимое для уверенного написания программ на текущем уровне. Одна из характерных особенностей нашего подхода — примеры программ для решения интересных задач, подкрепляемые упражнениями, — от обычного самообразования до нетривиальных задач, требующих творческих решений.

Среди *основных элементов программирования* рассматриваются переменные, операции присваивания, встроенные типы данных, передача управления в программе, массивы и средства ввода/вывода, включая графику и звук.

В разделе, посвященном *функциям и модулям*, читатели впервые встретятся с модульным программированием. Читатели уже знакомы с математическими функциями; мы воспользуемся этим обстоятельством для представления функций Java, а затем рассмотрим эффект от использования функций в программах, включая библиотеки функций и рекурсию. Особое внимание уделяется фундаментальной идее разбиения программы на компоненты с возможностью независимой отладки, сопровождения и повторного использования.

С *объектно-ориентированного программирования* начинается наше знакомство с абстракциями данных. На переднем плане находится концепция типа данных и его реализации с использованием механизма классов Java. Читатель научится использовать, создавать и проектировать типы данных. На этой стадии основными являются такие концепции, как модульность, инкапсуляция и другие парадигмы современного программирования.

Во второй части книги представлены концепции более высокого уровня: алгоритмы и структуры данных, теория вычислений, архитектура компьютеров.

Алгоритмы и структуры данных объединяют перечисленные выше парадигмы современного программирования с классическими методами структурирования и обработки данных, которые остаются эффективными и в новых приложениях. Мы познакомим читателя с классическими алгоритмами сортировки и поиска, а также расскажем о фундаментальных структурах данных и их практическом применении, уделяя особое внимание научному подходу для анализа быстродействия различных реализаций.

Теория вычислений позволяет получить ответы на основные вопросы о вычислениях с использованием простых абстрактных моделей вычислительных устройств. Дело не только в том, что на основании этого анализа делается ряд ценнейших теоретических выводов — многие из этих идей также напрямую применимы в практических вычислительных приложениях.

Архитектура компьютеров помогает понять, как происходят вычисления в реальном мире, — она связывает абстрактные машины из теории вычислений с реальными компьютерами, которыми мы пользуемся. Кроме того, изучение архитектуры раскрывает историческую перспективу, потому что микропроцессоры современных компьютеров и мобильных устройств не так уж сильно отличаются от первых вычислительных машин, разработанных в середине XX века.

Ключевая особенность этой книги — *практические применения теории в научной и технической области.* Мы объясняем каждую представленную концепцию, рассматривая примеры ее применения в конкретных областях. В книге встречаются примеры из прикладной математики, физики и биологии, а также из самой информатики; в их число входят модели физических систем, числовые методы, визуализация данных, синтез звука, обработка графики, финансовое моделирование и информационные технологии. Например, в первой главе встречаются примеры использования цепей Маркова для ранжирования веб-страниц и примеры, моделирующие задачу протекания, систему n тел и феномен «тесного мира». Эти примеры являются неотъемлемой частью текста. Они мотивируют студента к изучению материала, демонстрируют важность концепций программирования, а также убедительно доказывают ключевую роль вычислений в современной науке и технике.

В последних главах на первое место выходит *исторический контекст*. Здесь приводится увлекательная история разработки и практического применения фундаментальных идей информатики и вычислительной техники, авторами которых были Алан Тьюринг, Джон фон Нейман и многие другие ученые.

Наша главная цель – дать вам конкретные знания и навыки, необходимые для разработки эффективных решений любой задачи по программированию. Мы работаем с полноценными программами Java и рекомендуем читателям пользоваться ими. При этом мы сосредоточимся на индивидуальном программировании, а не на коллективной разработке программ.

Место в учебной программе

Эта книга предназначена для начинающих программистов. Приведенный в ней материал может быть использован в учебном курсе информатики начального уровня в контексте научного применения. Когда такой курс преподается по книге, студенты учатся программированию в знакомом контексте. Любой интересующийся, кто пройдет курс, основанный на этой книге, будет готов применить свои навыки при последующем обучении и будет понимать, в каком случае ему пригодится дальнейшее образование в области информатики.

Изучение программирования в контексте научных практических применений принесет пользу в том числе и студентам, для которых профильной дисциплиной станет информатика. Специалист по информатике должен обладать такой же базовой подготовкой в области научной методологии и так же хорошо понимать роль вычислений в науке, как и биолог, инженер или физик.

Более того, наш междисциплинарный подход позволяет колледжам и университетам проводить общий учебный курс для студентов с разными профильными дисциплинами. В книге излагается материал курса информатики первого года обучения, но внимание, уделяемое практическим примерам, оживляет теоретические концепции и повышает интерес студентов к их изучению. Благодаря междисципли-

нарному подходу студенты будут знакомиться с задачами из многих прикладных областей, и это поможет им более осознанно выбрать свою специализацию.

Впрочем, независимо от конкретного способа книгу лучше использовать в первых семестрах учебы. Во-первых, это позволит задействовать недавно изученный материал из школьного курса математики. Во-вторых, студенты, которые изучают программирование в начале своего учебного плана, смогут более эффективно применять компьютеры после перехода к профильным дисциплинам. Программирование, как и умение читать и писать, безусловно является важнейшим навыком для любого ученого или инженера. Студенты, которые усвоят концепции из этой книги, будут постоянно развивать этот навык на протяжении своей карьеры. Это поможет им пользоваться преимуществами вычислительных технологий для решения или лучшего понимания задач и проектов, встречающихся в выбранной ими области.

Что нужно для работы с книгой

Книга рассчитана на уровень начинающего программиста. Другими словами, мы не ожидаем от читателя никаких специальных знаний, выходящих за рамки того, что необходимо для других учебных курсов по научным дисциплинам и математике начального уровня.

Математическая подготовка важна. Хотя в книге мы не задерживаемся надолго на математической стороне происходящего, в ней нередко встречаются отсылки к материалу из курса математики старших классов, включая алгебру, геометрию и тригонометрию. Большинство студентов из нашей целевой аудитории соответствуют этому критерию изначально. И разумеется, мы опираемся на их знакомство с математическими основами при рассмотрении базовых концепций программирования.

Другая необходимая составляющая — *научная любознательность*. Студенты, изучающие науку и технику, с энтузиазмом относятся к возможности применения научного подхода для объяснения того, что происходит в природе. Мы пользуемся этой склонностью в примерах простых программ, которые рассказывают об окружающем мире красноречивее любых слов. При этом от читателя не ожидается никаких специальных знаний, кроме тех, которые можно получить в типичном школьном курсе математики, физики, биологии и химии.

Опыт программирования необязателен, но и вреда от него не будет. Одна из наших основных целей — обучение программированию, поэтому мы не предполагаем наличия у читателя готового опыта. Тем не менее составление программы для решения новой задачи – достаточно сложное интеллектуальное занятие, поэтому даже студенты, уже занимавшиеся программированием в школе, извлекут пользу из вводного курса, основанного на этой книге. Книга поможет в обучении студентов с разной подготовкой, поскольку рассмотренные примеры будут интересны как новичку, так и опытному программисту.

Опыт работы с компьютером необязателен, но студенты обычно регулярно пользуются компьютерами — например, для общения с друзьями или родственниками, прослушивания музыки, обработки фотографий и т. д. Понимание того, что силы вашего компьютера можно приложить к решению более важных и интересных задач — полезный, надолго запоминающийся урок.

Короче говоря, практически любой студент готов к прохождению учебного курса по материалам этой книги в своем первом семестре.

Цели

Чего может ожидать преподаватель от студента, прошедшего учебный курс по материалам этой книги?

Каждый, кто вел курсы программирования начального уровня, знает, что ожидания преподавателей курсов более высокого уровня обычно высоки: преподаватель считает, что все студенты знакомы с вычислительной средой и методологией, которые он собирается использовать. Лектор по физике полагает, что студенты могут написать за выходные программу физического моделирования; лектор по математике ожидает, что они умеют использовать конкретный математический пакет для числового решения дифференциальных уравнений; лектор по информатике ожидает досконального знания конкретной среды программирования. Может ли один курс вводного уровня удовлетворить столь разнородные ожидания? Нужно ли создавать разные учебные курсы для всех групп студентов?

Образовательные учреждения мучаются над этими вопросами с момента широкого распространения компьютеров во второй половине XX века. Наш ответ на них заключается в общем вводном учебном курсе информатики и программирования, аналогичном общепринятым вводным курсам математики, физики, биологии и химии. Книга стремится предоставить базовую информацию, необходимую всем студентам, изучающим научные и технические дисциплины, но при этом донести до читателя мысль о том, что информатика отнюдь не ограничивается одним лишь программированием. Преподаватели могут рассчитывать, что студенты, обучавшиеся по этой книге, обладают необходимыми знаниями и опытом, чтобы приспособиться к новой вычислительной среде и эффективно пользоваться компьютером в различных практических областях.

Чего может ожидать *студент*, прошедший курс обучения по материалам книги, при последующем обучении?

Мы хотим показать, что научиться программировать несложно, а усилия по овладению всей мощью компьютера будут вознаграждены. Читатели, освоившие материал книги, будут готовы к решению вычислительных задач в любой ситуации в своей последующей карьере. Они узнают, что современные среды программирования, такие как среда Java, открывают путь к решению любых задач, с которыми они могут столкнуться. Также читатели обретут уверенность, необходимую для изучения, оценки и использования других вычислительных инструментов. Студенты,

интересующиеся информатикой, будут хорошо подготовлены к дальнейшему развитию своих интересов; студенты, специализирующиеся на научных и инженерных дисциплинах, будут готовы к тому, чтобы использовать вычислительные технологии в своей учебе.

Онлайн-лекции

Полный видеокурс лекций (на английском языке), который может быть использован вместе с текстом книги, доступен по адресу:

http://www.informit.com/title/9780134493831

Как и традиционные лекции, этот видеокурс призван информировать и вдохновлять, побуждать к учебе и учить, отталкиваясь от содержимого книги. Наш опыт показывает, что эффективность усвоения студентами такого материала выше, чем традиционных лекций, благодаря возможности обращаться к нему в удобное время, в удобном темпе и с необходимым числом повторов.

Сайт книги

Большой объем информации, дополняющей текст книги, доступен в Интернете по адресу:

http://introcs.cs.princeton.edu/java[1]

Для краткости этот ресурс в дальнейшем будет называться «сайтом книги». Он содержит материалы для преподавателей, студентов и сторонних читателей. Весь материал находится в открытом доступе (с немногочисленными исключениями для поддержки тестирования).

Сайт книги дает преподавателям и студентам возможность пользоваться их собственными компьютерами для преподавания и изучения материала. Любой пользователь с компьютером и браузером может взяться за изучение программирования, следуя простым инструкциям на сайте книги. Этот процесс не намного сложнее загрузки программы-проигрывателя или песни. Как и любой веб-сайт, сайт книги непрерывно совершенствуется. Это исключительно важный ресурс для каждого обладателя книги. В частности, размещенные на нем материалы чрезвычайно важны для нашей цели — сделать компьютер неотъемлемым компонентом образования любого ученого или инженера.

Информация для *преподавателей*, размещенная на сайте, в основном опирается на методику преподавания, разработанную нами за последнее десятилетие: мы проводим две лекции в неделю для большой аудитории, дополняя их двумя еженедельными семинарами для небольших групп с преподавателями или ассистентами. На сайте книги размещены слайды для презентаций, которые задают тон лекций.

[1] Доступ к лекциям платный.

Для *ассистентов* на сайте книги доступны подборки задач и программные проекты, основанные на упражнениях из книги, но гораздо более подробные. Каждый учебный проект по программированию представляет важную концепцию в контексте интересной практической задачи. Последовательность назначения заданий отражает наш подход к преподаванию программирования. На сайте приведены полные описания всех заданий с подробной структурированной информацией, которая поможет студентам справиться с ними за отведенное время, с описаниями рекомендуемых подходов и планами проведения семинаров.

Студенты на сайте книги смогут получить доступ к большей части материала книги, включая исходный код и дополнительные материалы для самообразования. Для многих упражнений книги на сайте приводятся решения с полными исходными текстами программ и тестовыми данными. К заданиям по программированию прилагается разнообразная информация, включающая предлагаемые пути решения, контрольные вопросы, ответы на типичные вопросы и наборы тестовых данных.

Сторонний читатель найдет на сайте книги всевозможную дополнительную информацию, относящуюся к материалу. По всему сайту присутствуют ссылки и другие пути для получения дополнительной информации по рассматриваемым темам. Объем доступного материала намного превышает возможности любого отдельного читателя, но мы лишь стараемся вызвать у читателя интерес к поиску дополнительной информации по тематике книги.

Благодарности

Этот проект находился в разработке с 1992 года, поэтому количество людей, внесших свой вклад в работу, слишком велико, чтобы мы могли их всех здесь упомянуть. Мы хотим особо поблагодарить Энн Роджерс (Anne Rogers) за помощь в запуске проекта, Дэйва Хэнсона (Dave Hanson), Эндрю Эппла (Andrew Appel) и Криса ван Вика (Chris van Wyk) за их терпение при объяснении абстракции данных, Лизу Уортингтон (Lisa Worthington) и Донну Габай (Donna Gabai), которые первыми в полной мере столкнулись с задачей преподавания этого материала студентам-первокурсникам, и Дуга Кларка (Doug Clark) за терпение в изложении материала по машинам Тьюринга и схемотехнике. Также мы хотим поблагодарить факультет, аспирантов и преподавателей, занимавшихся преподаванием этого материала последние 25 лет в Принстонском университете, и тысячи студентов, которые посвятили себя его изучению.

Роберт Седжвик

Кевин Уэйн

май 2016 г.

От научного редактора русского издания

Ключевая характеристика этой книги, отраженная и в ее названии, — междисциплинарность. Во-первых, ее содержание охватывает очень широкий спектр вопросов — от основ языка Java до сложных структур данных, от теории алгоритмов до аппаратных основ вычислительной техники. Во-вторых, постоянно привлекаются примеры реальных задач (конечно, упрощенных для учебных целей), относящихся к излагаемым темам. В результате читатель получает, как и обещают авторы, разностороннюю «стартовую» подготовку в различных областях, общим для которых является активное использование вычислительной техники и программирования (если понимать программирование в достаточно широком смысле).

Материал книги структурирован и выстроен наиболее удобным для учебного курса образом — в таком виде, как его представляют себе авторы. Теоретическая часть дополняется большим количеством практических примеров и самостоятельных заданий, и многие из них неоднократно возвращают читателя к одним и тем же задачам, но каждый раз уже на более высоком уровне. Порядок изложения может показаться непривычным по сравнению с нашими типичными учебными программами: от основ конкретного языка программирования к теории вычислений и далее к архитектурным и аппаратным основам. Возможно, это отступает от «академической» последовательности рассмотрения, но зато лучше отвечает ожиданиям учащегося: быстро освоить наиболее востребованную «базу», в первую очередь практическую, и затем расширять и углублять свои знания. Книга (действительно Computer Science!) может служить обширным *универсальным* фундаментом для будущих специалистов различных профессий.

Еще одна небольшая, но интересная особенность — использование авторами собственной «учебно-демонстрационной» вычислительной машины TOY, сначала в виде лишь архитектуры и программной модели, но затем и в виде функциональной схемы (физическая реализация которой к этому моменту уже рассмотрена!). Вероятно, это не стоило бы отдельного упоминания, но в качестве косвенной и совершенно субъективной характеристики: лично у меня если первые главы и не вызвали особенно сильного отклика (все-таки это просто основы программирования), то к концу появился и оформился интерес воспроизвести архитектуру TOY самостоятельно. Причем «в железе» и с доработанным набором инструкций!

Сиротко С. И.,
доцент Белорусского государственного университета информатики и радиоэлектроники, кандидат физико-математических наук

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1
Основы программирования

В этой главе мы постараемся убедить вас, что написать программу проще, чем написать какой-нибудь текст — скажем, курсовую работу или сочинение. Писать сочинения сложно: в школе нас этому учат годами. Напротив, зная всего несколько основных конструкций, вы сможете писать программы, которые позволяют вам решать множество интересных, но не решаемых традиционными средствами задач. В этой главе вы освоите эти конструкции, сделаете свои первые шаги в области программирования на Java и познакомитесь с некоторыми интересными программами. Всего за несколько недель вы сможете выражать свои мысли и намерения в программах. Умение программировать, как и умение писать сочинения, — долгосрочный навык, который вы будете постоянно совершенствовать в будущем.

В этой книге вы будете изучать *язык программирования Java*. Эта задача будет намного проще, чем, допустим, задача изучения иностранного языка. На самом деле язык программирования характеризуется словарем из нескольких десятков слов и правил грамматики. Большая часть материала, представленного в книге, с таким же успехом может быть выражена на Python, C++ или любом другом современном языке программирования. Во всех описаниях язык Java используется именно для того, чтобы вы могли с ходу взяться за создание и запуск программ. С другой стороны, в ходе изучения программирования бывает трудно определить, какие подробности актуальны в каждой конкретной ситуации. Язык Java получил широкое распространение, поэтому умение работать на этом языке поможет вам писать программы на многих компьютерах (прежде всего — на вашем собственном). Кроме того, умение программировать на Java упрощает изучение других языков, включая низкоуровневые языки вроде C и специализированные языки вроде Matlab.

1.1. Ваша первая программа

Наша цель в этом разделе — познакомить читателя с миром программирования на языке Java. Для этого будет рассмотрена последовательность основных действий, необходимых для запуска простой программы. *Платформа Java* (в даль-

нейшем просто *Java*) представляет собой совокупность приложений, не так уж сильно отличающихся от других привычных вам приложений (редакторы документов, почтовые клиенты или браузеры). Как и с любым приложением, вы должны убедиться в том, что платформа Java правильно установлена на вашем компьютере. На многих компьютерах она установлена изначально, но при необходимости вы сможете легко загрузить ее. Также вам понадобится текстовый редактор и терминальное приложение, поддерживающее командную строку («консоль»). Ваша первая задача — найти инструкции для установки и настройки такой среды программирования на вашем компьютере на странице *http://introcs.cs.princeton.edu/java*.

В дальнейшем мы будем называть этот сайт *«сайтом книги»*. На нем вы найдете большое количество вспомогательной информации по материалу книги, которой вы сможете пользоваться в процессе программирования.

Программирование на Java

Чтобы вам было проще взяться за разработку программ на языке Java, мы разобьем процесс программирования на три шага. Вы должны:

- создать программу — ввести ее исходный код в файл (допустим, с именем MyProgram.java);
- откомпилировать ее командой `javac MyProgram.java` в терминальном окне;
- выполнить (запустить) программу командой `java MyProgram` в терминальном окне.

Первый шаг начинается с пустого экрана, а заканчивается последовательностью введенных символов — по аналогии с тем, как вы пишете сообщение электронной почты или реферат. Программисты называют текст программы *«исходным кодом»*, а процесс его написания — «кодированием». На втором шаге запускается системное приложение, которое компилирует программу (преобразует ее в форму, более подходящую для компьютера) и помещает результат в файл с именем MyProgram.class. На третьем шаге управление компьютером передается от системы вашей программе (которая снова возвращает управление системе после своего завершения). В разных системах применяются разные способы создания, компиляции и выполнения программ. Мы выбрали последовательность, приведенную в тексте, как самую простую для описания и использования для небольших программ.

Создание программы

Программа на Java представляет собой не что иное, как последовательность символов (такую же, как в абзаце текста или стихотворении), хранящуюся в файле с расширением .java. Следовательно, вы должны просто ввести последовательность символов точно так же, как вы делаете это в почтовом клиенте или любом другом компьютерном приложении. Для этого можно воспользоваться любым

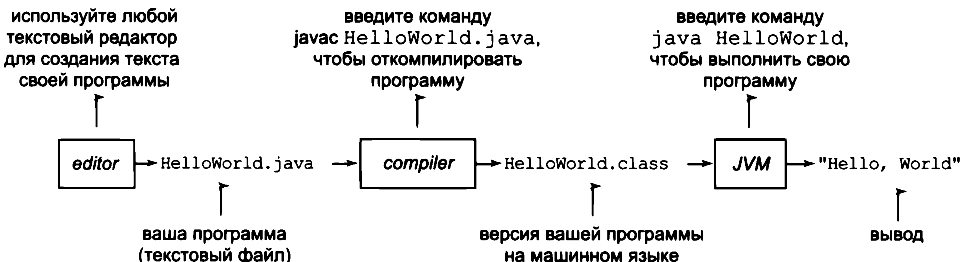
текстовым редактором или же одной из более совершенных *интегрированных сред разработки*, описанных на сайте книги. Пожалуй, для программ, представленных в книге, применение таких сред — это перебор, но они несложны в использовании, обладают множеством полезных функций и широко используются профессионалами.

Компиляция программы

На первый взгляд может показаться, что создатели языка Java старались сделать его более понятным для компьютера. В действительности же язык Java создавался как более понятный для программиста, то есть для вас. Язык компьютера намного примитивнее языка Java. *Компилятор* — приложение, которое переводит программу Java на язык, более подходящий для выполнения на компьютере. Компилятор получает на входе файл с расширением .java (вашу программу) и создает файл с тем же именем, но расширением .class (версия на машинном языке[1].) Чтобы воспользоваться компилятором Java, введите в терминальном окне команду `javac` с именем файла компилируемой программы.

Выполнение (запуск) программы

После того как программа будет откомпилирована, ее можно запустить. Это самая интересная часть, в которой ваша программа получает управление над компьютером (в границах того, что разрешено языку Java). Вероятно, правильнее будет сказать, что компьютер выполняет ваши инструкции. А еще правильнее — что часть Java, называемая *виртуальной машиной Java* (Java Virtual Machine, сокращенно JVM), приказывает вашему компьютеру выполнять ваши инструкции. Чтобы воспользоваться JVM для выполнения вашей программы, введите в терминальном окне команду `java` с именем вашей программы.



Разработка программы на языке Java

[1] При использовании Java это не собственно *машинный* язык, пригодный для выполнения непосредственно процессором, а специальный «промежуточный» язык, так называемый *байт-код*, интерпретируемый *виртуальной машиной Java*. Но пока это различие можно считать несущественным. — *Примеч. науч. ред.*

Листинг 1.1.1. Hello, World

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      // Выводит сообщение "Hello, World" в терминальном окне.
      System.out.println("Hello, World");
   }
}
```

Перед вами программа на языке Java, предназначенная для решения одной простой задачи. Традиционно это первая программа, которую пишут начинающие программисты. Ниже показано, что происходит при компиляции и выполнении программы. Терминальное приложение выдает приглашение командной строки (% в этой книге) и выполняет введенные команды (`javac` и `java` в приведенном ниже примере). В этой книге мы будем выделять жирным шрифтом текст, вводимый пользователем, а выводимые программой результаты будут записаны обычным шрифтом. В данном примере программа выводит в терминальном окне сообщение «Hello, World».

```
% javac HelloWorld.java
% java HelloWorld
Hello, World
```

В листинге 1.1.1 приводится пример полноценной программы на Java. Программе присвоено имя `HelloWorld`; это означает, что ее код хранится в файле с именем HelloWorld.java (по соглашениям Java). Программа выполняет всего одно действие: она выводит сообщение в терминальном окне. Ради целостности изложения для описания программы будут использоваться стандартные термины Java, но их определения в книге будут приведены намного позднее: листинг 1.1.1 состоит из единственного класса с именем `HelloWorld`, который содержит метод с именем `main()`. (При упоминании метода в тексте книги после имени будут ставиться круглые скобки `()`, чтобы имя метода можно было отличить от других имен.) До раздела 2.1 все наши классы будут иметь одинаковую структуру. А пока считайте, что «класс» — то же самое, что «программа».

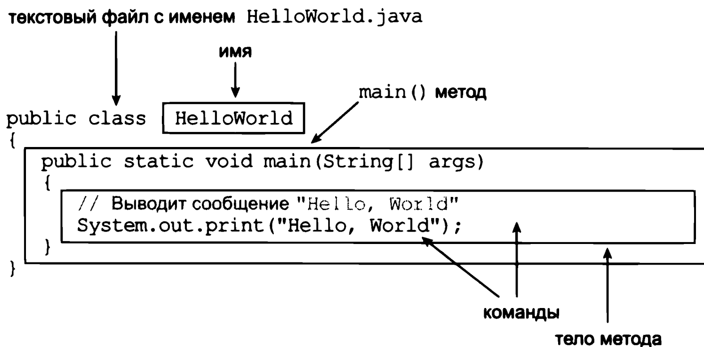
В первой строке метода указывается его имя и другая информация; далее идет последовательность строк кода, заключенная в фигурные скобки, при этом каждая строка обычно завершается точкой с запятой (`;`). Пока что можно считать, что «программировать» — значит задавать имя класса и последовательность строк кода, составляющих его метод `main()` (*тело* метода). В листинге 1.1.1 таких строк две.

- Первая строка содержит *комментарий*, предназначенный для документирования программы. В языке Java однострочный комментарий начинается с двух символов `'/'` и продолжается до конца строки. В этой книге комментарии будут выделяться серым шрифтом. Язык Java игнорирует комментарии — они предназначены только для людей, которые будут читать программу.

- Вторая строка — *операция вывода* — вызывает метод с именем `System.out.println()` для вывода в терминальном окне текстового сообщения — того, что заключено в пару двойных кавычек.

В следующих двух разделах вы узнаете о разных конструкциях и выражениях, используемых в программах. Пока в наших программах будут использоваться только комментарии и операции вывода, как в `HelloWorld`.

Когда вы вводите в терминальном окне команду `java` с именем класса, система вызывает метод `main()`, определенный в этом классе, и последовательно выполняет его код, строка за строкой. Таким образом, команда `java HelloWorld` заставляет систему вызвать метод `main()` из листинга 1.1.1 и выполнить две строки этого метода. Первая строка содержит комментарий, который игнорируется языком Java. Вторая выводит заданное сообщение в терминальном окне.



Строение программы

С 1970-х годов существует традиция: первая программа начинающего программиста выводит сообщение «Hello, World». Введите код из листинга 1.1.1 в файл, откомпилируйте и выполните его. При этом вы повторите опыт множества других разработчиков, когда те учились программировать. Не забудьте, что вам понадобится нормальный редактор и терминальное приложение. На первый взгляд вывод сообщения в терминальном окне не кажется таким уж серьезным достижением; но если задуматься, становится ясно, что речь идет о важнейшей функции, которая должна поддерживаться любой программой, — получении информации о том, что сейчас в ней происходит.

Некоторое время код всех наших программ будет напоминать листинг 1.1.1, не считая иного набора строк в `main()`. А это означает, что вам не обязательно начинать написание программы «с нуля». Вместо этого можно:

- скопировать содержимое HelloWorld.java в новый файл с именем, выбранным вами для программы (за именем следует расширение .java);
- заменить `HelloWorld` в первой строке новым именем программы;

- заменить комментарий и строку с операцией вывода в теле `main()` другими операциями.

Ваша программа характеризуется своей последовательностью операций и именем. Каждая программа на языке Java должна храниться в файле с именем, совпадающим с именем класса в первой строке[1], и расширением .java.

Ошибки

Начинающий программист не всегда четко понимает суть различий между редактированием, компиляцией и исполнением программ. Постарайтесь четко разделить эти процессы, когда вы делаете свои первые шаги в программировании; это поможет вам лучше понять, к каким последствиям приведут неизбежно возникающие ошибки.

Большинство ошибок можно исправить или предотвратить, тщательно проверяя исходный код программы во время работы над ним, — по аналогии с тем, как вы проверяете правописание и грамматику во время составления сообщения электронной почты. Некоторые ошибки, называемые *ошибками компиляции*, распознаются во время компиляции, потому что они не позволяют компилятору выполнить преобразование. Еще одна категория ошибок — *ошибки времени выполнения* — обнаруживаются только во время выполнения программы.

Ошибки в программах (или «*баги*») сильно отравляют жизнь программиста: сообщения об ошибках бывают запутанными или невразумительными, а найти источник ошибки бывает очень трудно. Выявление ошибок станет одним из первых навыков, которые вам предстоит освоить; вы также научитесь проявлять осторожность во время написания кода, чтобы не вносить их в свои программы. Некоторые примеры таких ошибок приведены в разделе «Вопросы и ответы» в конце раздела.

Ввод и вывод

Обычно программам должны передаваться *входные данные* — то есть данные, которые обрабатываются программой для получения результата. Самый простой способ передачи входных данных представлен в программе `UseArgument` (листинг 1.1.2). При запуске программы `UseArgument` передается *аргумент командной строки*, который указывается после имени программы. Переданный аргумент выводится в терминальном окне в составе сообщения. Результат выполнения программы зависит от того, что было введено после имени программы. Запуская программу с разными аргументами командной строки, вы будете получать разные результаты. Механизм передачи аргументов командной строки будет подробно рассмотрен позднее, в разделе 2.1. А пока достаточно понять, что `args[0]` — обозначение первого аргумента командной строки, введенного после имени программы, `args[1]` — обозначение второго аргумента и т. д. Таким образом, обозначение `args[0]` может

---

[1] Строго говоря, соответствие не обязательно, но желательно его соблюдать. — *Примеч. науч. ред.*

Листинг 1.1.2. Использование аргумента командной строки

```
public class UseArgument
{
   public static void main(String[] args)
   {
      System.out.print("Hi, ");
      System.out.print(args[0]);
      System.out.println(". How are you?");
   }
}
```

В этой программе представлен один из способов управления действиями ваших программ: передача аргумента в командной строке. Передавая аргументы, мы можем влиять на поведение наших программ.

```
% javac UseArgument.java
% java UseArgument Alice
Hi, Alice. How are you?
% java UseArgument Bob
Hi, Bob. How are you?
```

использоваться внутри программы для представления первого слова, введенного в командной строке[1] при запуске программы на выполнение (как в примере с `UseArgument`).

Кроме метода `System.out.println()`, `UseArgument` вызывает метод `System.out.print()`. Этот метод очень похож на `System.out.println()`, но он выводит только заданную строку (без завершающего перевода строки.)

И снова программа, которая просто выводит полученное значение, на первый взгляд не кажется особо интересной, но если задуматься, вы поймете, что другой базовой функцией любой программы должна быть возможность реагирования на информацию, вводимую пользователем для управления работой программы. Простой модели, представленной программой `UseArgument`, будет достаточно для описания основного механизма программирования Java и решения многих интересных вычислительных задач.

Если взглянуть на происходящее в перспективе, мы видим, что `UseArgument` попросту реализует функцию, отображающую строку (аргумент командной строки) на другую строку (сообщение, выводимое в терминальном окне.) Java-программу можно представить в виде «черного ящика», который превращает входную строку в выходную строку.

Эта модель привлекательна не только своей простотой, но и тем, что она достаточно универсальна для выполнения, в принципе, любой вычислительной задачи. Напри-

[1] Здесь имеется в виду часть строки, отделенная от других ее частей пробелами. — *Примеч. науч. ред.*

Общее представление программы Java

мер, сам компилятор Java представляет собой не что иное, как программу, которая получает строку символов на входе (файл .java) и создает другую строку символов на выходе (соответствующий файл .class). Позднее вы научитесь писать программы для решения многих интересных задач (хотя эти программы будут гораздо проще компилятора). А пока придется смириться с ограничениями на размер и тип входных и выходных данных ваших программ; в разделе 1.5 будет показано, как используются более сложные механизмы передачи входных и выходных данных. В частности, вы научитесь работать с входными и выходными строками произвольной длины и другими типами данных (например, звуком и графикой).

Вопросы и ответы

В. Почему именно Java?

О. Программы, которые мы пишем, будут очень похожи на свои аналоги на многих других языках, и выбор языка не столь важен. Мы используем Java, потому что этот язык популярен, он включает полный набор современных абстракций, а также разнообразные автоматические проверки ошибок в программах; все это делает его хорошо подходящим для изучения программирования. Впрочем, идеальных языков вообще не бывает, и в будущем вы наверняка будете программировать также и на других языках.

В. Мне действительно нужно набирать все программы из книги, чтобы опробовать их? Я верю, что они запускаются и выводят указанный результат.

О. Каждый читатель должен набрать и запустить программу `HelloWorld`. Вы намного лучше поймете материал, если вы также запустите программу `UseArgument`, опробуете ее на разных вариантах ввода и внесете изменения для проверки собственных идей. Чтобы сэкономить время, вы найдете весь код из этой книги (и много другого кода) на сайте книги. На сайте также имеется информация об установке и запуске Java на вашем компьютере, ответы на некоторые упражнения, веб-ссылки и другая дополнительная информация, которая может пригодиться вам в процессе программирования.

В. Что означают слова `public`, `static` и `void`?

О. Эти ключевые слова описывают некоторые свойства `main()`, которые будут описаны позднее в книге. А пока мы просто включаем эти ключевые слова в код (потому что они необходимы), но не упоминаем их в тексте.

В. Что означают комбинации символов `//`, `/*` и `*/` в коде?

О. Это *комментарии*, и они игнорируются компилятором. Комментарием является либо текст между `/*` и `*/`, либо текст от `//` до конца строки. Комментарии чрезвычайно полезны, потому что они помогают другим программистам понять ваш код, — более того, они даже помогут вам разобраться в вашем собственном коде, написанном некоторое время назад. Из-за нехватки места в книге мы не сможем снабжать программы обильными комментариями; вместо этого каждая программа будет подробно описана в сопроводительном тексте и иллюстрациях. Программы, размещенные на сайте, прокомментированы более подробно.

В. Какие правила действуют в Java относительно табуляций, пробелов и символов новой строки?

О. Такие символы относятся к категории *пробельных*, или *разделителей* (whitespace). Компиляторы Java рассматривают все разделители в тексте программы как эквивалентные. Например, программу `HelloWorld` можно переписать в следующем виде:

```
public class HelloWorld { public static void main ( String
[] args) { System.out.println("Hello, World")        ; } }
```

Но обычно в книге при написании программ на Java будут соблюдаться соглашения о разделителях и отступах, аналогичные разбиению на абзацы и строки при написании прозы или стихов.

В. Каковы правила применения кавычек?

О. Символы, заключенные в двойные кавычки, являются исключением для правила из предыдущего вопроса: обычно символы в кавычках интерпретируются буквально, чтобы вы получили в точности то, что выводилось на печать. Если заключить в кавычки любое количество смежных пробелов, вы получите в выводе точно такое же количество пробелов. Если случайно пропустить кавычку, компилятор может прийти в полное замешательство, потому что по кавычкам он отличает символы строки от других частей программы.

В. Что произойдет, если пропустить фигурную скобку или неправильно ввести одно из ключевых слов, например `public`, `static`, `void` или `main`?

О. Все зависит от того, что именно вы сделаете. Такие ошибки называются синтаксическими ошибками, и обычно они обнаруживаются компилятором. Например, если написать программу `Bad`, которая полностью повторяет программу `HelloWorld`, за исключением строки с левой фигурной скобкой (и сменить имя программы с `HelloWorld` на `Bad`), вы получите следующее полезное сообщение:

```
% javac Bad.java
Bad.java:1: error: '{' expected
public class Bad
                ^
1 error
```

Из сообщения можно сделать правильный вывод, что в строку нужно вставить левую фигурную скобку. Но компилятор не всегда может точно сообщить, какая

ошибка была допущена, и сообщения об ошибках порой трудно понять. Например, если пропустить вторую левую фигурную скобку вместо первой, сообщение будет выглядеть так:

```
% javac Bad.java
Bad.java:3: error: ';' expected
   public static void main(String[] args)
                                         ^
Bad.java:7: error: class, interface, or enum expected
}
^
2 errors
```

Чтобы привыкнуть к таким сообщениям, попробуйте намеренно вставить ошибку в простую программу и посмотрите, что получится. Что бы ни говорилось в сообщении об ошибке, помните: компилятор — ваш друг, и он просто пытается сообщить вам, что с программой что-то не так.

В. Какие методы Java можно использовать в программах?

О. Существуют тысячи таких методов. Мы будем представлять их вам постепенно (начиная со следующего раздела), чтобы вы не растерялись от слишком богатого выбора.

В. При запуске программы `UseArgument` я получаю странное сообщение об ошибке. В чем дело?

О. Скорее всего, вы забыли включить аргумент командной строки:

```
% java UseArgument
Hi, Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at UseArgument.main(UseArgument.java:6)
```

Java (точнее, виртуальная машина Java, JVM, см. выше) жалуется на то, что вы запустили программу, но не передали обещанный аргумент командной строки. Массивы и индексы более подробно рассматриваются в разделе 1.4. Запомните это сообщение об ошибке — скорее всего, вы еще столкнетесь с ним. Даже опытные программисты время от времени забывают указывать аргументы командной строки.

Упражнения

1.1.1. Напишите программу, которая выводит сообщение «Hello, World» 10 раз.

1.1.2. Опишите, что произойдет, если пропустить в программе HelloWorld.java:

a. `public`

b. `static`

c. `void`

d. `args`

1.1.3. Опишите, что произойдет, если в программе HelloWorld.java совершить опечатку в следующем слове (скажем, пропустить вторую букву):

a. `public`

b. `static`

c. `void`

d. `args`

1.1.4. Опишите, что произойдет, если в программе HelloWorld.java разместить двойные кавычки в команде вывода в разных строках, как в следующем фрагменте:

```
System.out.println("Hello,
                    World");
```

1.1.5. Опишите, что произойдет при попытке выполнить программу `UseArgument` каждой из следующих командных строк:

a. `java UseArgument java`

b. `java UseArgument @!&^%`

c. `java UseArgument 1234`

d. `java UseArgument.java Bob`

e. `java UseArgument Alice Bob`

1.1.6. Модифицируйте программу UseArgument.java — создайте на ее основе программу UseThree.java. Новая программа должна получать три имени в аргументах командной строки и выводить предложение, в котором эти имена перечисляются в обратном порядке, — например, для команды `java UseThree Alice Bob Carol` должно выводиться сообщение `Hi Carol, Bob, and Alice`.

1.2. Встроенные типы данных

Программируя на Java, необходимо всегда учитывать тип данных, которые обрабатывает ваша программа. Программы из раздела 1.1 работали со строками символов, многие программы этого раздела работают с числами, а позднее в книге будет представлено много других типов. Понимать различия между ними настолько важно, что мы формально определим концепцию: *тип данных* представляет собой *множество значений* и *множество операций*, определенных для этих значений. Вам уже знакомы различные типы чисел (например, целые и вещественные числа) и определенные для них операции (например, сложение и умножение.) В математике мы привыкли рассматривать множества чисел как бесконечные; в компьютерных программах приходится работать с конечными множествами. Каждая выполняемая операция определяется *только* для конечного множества значений из связанного с ней типа данных.

В Java существуют восемь *примитивных* типов данных; большинство из них предназначено для числовых данных разных видов. Из восьми примитивных типов чаще всего используются следующие: `int` для целых чисел; `double` для вещественных чисел; `boolean` для логических значений «истина-ложь». Другие типы данных предоставляются библиотеками Java: например, в программах из раздела 1.1 тип `String` используется для работы со строками символов. В языке Java тип `String` занимает особое место, потому что он играет очень важную роль во вводе и выводе. Соответственно он обладает некоторыми характеристиками примитивных типов — например, некоторые из его операций встроены в язык Java. Для удобства мы будем обозначать примитивные типы и `String` объединяющим термином «*встроенные типы*». А пока основное внимание будет уделяться программам, основанным на вычислениях со встроенными типами. Позднее вы узнаете о типах данных из библиотеки Java и научитесь создавать собственные типы. Собственно, проектирование типов данных часто занимает центральное место в программировании на Java; об этом будет рассказано в главе 3.

После определения основных терминов будут представлены примеры программ и фрагменты кода, демонстрирующие использование разных типов данных. Никаких серьезных вычислений в этих фрагментах не будет, но похожий код вскоре встретится вам в более длинных программах. Понимание типов данных (значений и определенных для них операций) абсолютно необходимо для тех, кто делает свои первые шаги в карьере программиста. Оно создает условия для работы с более сложными программами, представленными в следующем разделе. Во всех написанных вами программах будут встречаться конструкции, напоминающие крошечные фрагменты из этого раздела.

тип	множество значений	типичные операторы	примеры литералов (значений)
`int`	целые числа	`+ - * / %`	`99 12 2147483647`
`double`	вещественные числа	`+ - * /`	`3.14 2.5 6.022e23`
`boolean`	логические значения	`&& || !`	`true false`
`char`	символы		`'A' '1' '%' '\n'`
`String`	последовательности символов	`+`	`"AB" "Hello" "2.5"`

Основные типы данных

Терминология

Для рассмотрения типов данных необходимо ввести ряд терминов. Начнем со следующего фрагмента кода:

```
int a, b, c;
a = 1234;
b = 99;
c = a + b;
```

Первая строка содержит *объявление* трех переменных с *идентификаторами* a, b и c и типом данных int. В следующих трех строках расположены *операции присваивания*, изменяющие значения этих переменных; в них используются *литералы* 1234 и 99, а также *выражение* a + b, в результате вычисления которого переменной c присваивается значение 1333.

Литералы

Литерал — представление значения определенного типа в коде Java. Для представления целочисленных значений типа int используются последовательности цифр, такие как 1234 или 99. Последовательности цифр с добавлением точки (разделителя дробной части) представляют значения типа double — например, 3.14159 или 2.71828. Два возможных значения типа boolean обозначаются ключевыми словами true и false, а последовательности символов в кавычках (например, "Hello, World") используются для представления значений типа String.

Операторы

Оператор представляет действие, выполняемое над данными определенного типа. Операторы + и * используются в Java для представления сложения и умножения целых и вещественных чисел; операторы &&, || и ! — для представления логических операций и т. д. Наиболее часто используемые операторы для встроенных типов данных будут описаны позднее в этом разделе.

Идентификаторы

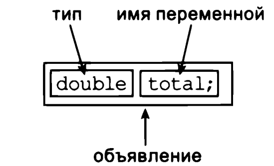
Идентификатор — это некоторое имя (например, имя переменной) в коде Java. Любой идентификатор состоит из букв, цифр, символов подчеркивания (_) и символов денежной единицы ($), при этом первый символ не является цифрой. Например, последовательности символов abc, Ab$, abc123 и a_b являются допустимыми идентификаторами Java, а Ab*, 1abc и a+b — нет. В идентификаторах учитывается регистр символов, так что Ab, ab и AB считаются разными именами. Некоторые зарезервированные слова, такие как public, static, int, double, String, true, false и null, имеют особый смысл в языке Java и не могут использоваться в качестве идентификаторов.

Переменные

Переменная используется для хранения значения некоторого типа и для обращения к нему по имени. В Java каждая переменная относится к определенному типу и хранит одно из возможных значений этого типа. Например, в переменной типа int могут храниться значения 99 или 1234, но не значения 3.14159 или "Hello, World". В разных переменных одного типа могут одновременно храниться одинаковые значения. Кроме того, как подсказывает сам термин, значение переменной может изменяться в ходе вычислений. Например, в некоторых примерах этой книги переменная с именем sum используется для хранения накапливаемой суммы числового ряда. Переменные создаются их объявлениями, а для вычисления их значения используются выражения.

Объявления

Для создания переменных в языке Java используются *декларации* (или *объявления*). Объявление состоит из типа и имени переменной. Компилятор Java выделяет блок памяти, достаточный для хранения значения заданного типа, и связывает имя переменной (идентификатор) с этим блоком памяти, чтобы обращаться к значению при последующих обращениях к переменной в коде. Для экономии места можно в одной декларации объявить несколько однотипных переменных.



Строение объявления

Соглашения об именах переменных

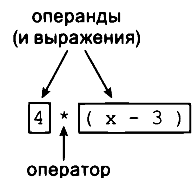
При выборе имен программисты обычно соблюдают соглашения о стиле программирования. В этой книге мы будем присваивать каждой переменной осмысленное имя, которое состоит из буквы нижнего регистра, за которой следуют буквы нижнего регистра, буквы верхнего регистра и цифры. Буквы верхнего регистра будут использоваться для выделения первых букв слов в составных именах переменных. Например, в книге будут использоваться имена переменных `i`, `x`, `y`, `sum`, `isLeapYear` и `outDegrees` (среди многих других.) У программистов такая схема записи имен называется «верблюжьей» («camelCase») — буквы верхнего регистра выделяются, словно горбы верблюда.

Константы

Константами будут называться переменные[1], значение которых не изменяется в ходе выполнения программы (или между двумя запусками программы.) В этой книге константам будут присваиваться имена, состоящие из буквы верхнего регистра, за которой следуют буквы верхнего регистра, цифры и символы подчеркивания, например `SPEED_OF_LIGHT` или `DARK_RED`.

Выражения

Выражение представляет собой комбинацию литералов, переменных и операторов, обрабатываемых Java для получения значения результата. Выражения с примитивными типами часто напоминают математические формулы: операции задают операции типа данных, выполняемые с одним или несколькими операндами. Чаще всего используются *двухместные* операторы, получающие ровно два операнда, например `x - 3` или `5 * x`.



Строение выражения

---

[1] Здесь затронуто глубокое различие между литералами и константами: литерал — непосредственно заданное значение, а константа — имеющий такое значение элемент данных, подобный переменной, но не изменяющий свое содержимое. — *Примеч. науч. ред.*

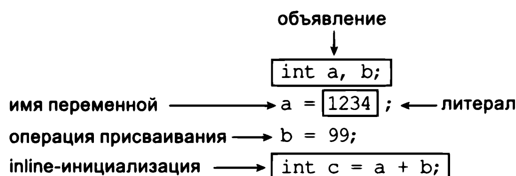
Операнды могут быть произвольными выражениями, их можно заключать в круглые скобки. Например, если компилятор Java встретит в программе выражение `4 * (x - 3)` или `5 * x - 6`, он поймет, что вы имеете в виду. В сущности, выражение — это директива для выполнения последовательности операций; в коде программы выражение представляет значение, полученное в результате вычислений.

Приоритет операторов

Выражение является формализованной записью для последовательности операций; но в каком порядке должны применяться операторы? В Java этот порядок полностью определяется набором естественных, четко определенных правил. В арифметических операциях умножение и деление выполняются до сложения и вычитания, так что выражения `a - b * c` и `a - (b * c)` представляют одну и ту же последовательность действий. Если арифметические операторы имеют одинаковый приоритет, то порядок определяется принципом *левосторонней ассоциативности*; следовательно, выражения `a - b - c` и `(a - b) - c` представляют одну последовательность действий. Круглые скобки могут использоваться для изменения приоритета — используйте запись `a - (b - c)`, если вам нужен именно такой порядок. Возможно, когда-нибудь вам встретится код Java с неочевидной последовательностью выполнения, но мы будем использовать круглые скобки, чтобы в книге такого кода не было. Если вас заинтересует эта тема, полную информацию о правилах приоритета можно найти на сайте книги.

Операции присваивания

Присваивание связывает значение типа данных с переменной. Когда мы используем запись `c = a + b` в языке Java, это не математическое равенство, а *действие*: значением переменной `c` становится сумма значений `a` и `b`. Правда, значение `c` математически равно значению `a + b` сразу же после выполнения команды присваивания, но настоящей целью является изменение (или инициализация) значения `c`. В левой части присваивания должна стоять одна переменная; в правой части размещается любое выражение, результатом которого является значение совместимого типа. Это означает, например, что команды `1234 = a;` или `a + b = b + a;` в языке Java недопустимы. Короче говоря, знак `=` *вовсе не означает математическое равенство.*



Использование примитивного типа данных

Inline-инициализация

Прежде чем использовать переменную в выражении, сначала вы должны объявить ее и присвоить начальное значение. Если хотя бы одно из этих условий не будет выполнено, произойдет ошибка компиляции. Для экономии места объявление можно объединить с присваиванием в конструкции, называемой *inline-инициализацией*. Например, в следующем коде объявляются две переменные a и b, которые инициализируются значениями 1234 и 99 соответственно:

```
int a = 1234;
int b = 99;
```

Как правило, в книге используется этот способ объявления и инициализации переменной в точке первого использования ее в программе.

Трассировка изменений в значениях переменной

Давайте проверим, насколько хорошо вы поняли смысл команд присваивания. Убедитесь в том, что следующий фрагмент меняет местами значения a и b (предполагается, что a и b — переменные типа int):

```
int t = a;
a = b;
b = t;
```

	a	b	t
`int a, b;`	*undefined*	*undefined*	
`a = 1234;`	**1234**	*undefined*	
`b = 99;`	1234	**99**	
`int t = a;`	1234	99	**1234**
`a = b;`	**99**	99	1234
`b = t;`	99	**1234**	1234

Ваша первая трассировка

Воспользуйтесь проверенным методом анализа поведения программы: изучите таблицу значений переменных после каждой команды (такая таблица называется *трассировочной*).

Безопасность типов

Язык Java требует, чтобы программист объявлял в программе тип каждой переменной. Это позволяет Java выявлять ошибки несоответствия типов во время компиляции и сообщать о потенциальных ошибках в программе. Например, переменной типа int нельзя присвоить значение double, умножить String на boolean или использовать в выражении неинициализированную переменную. Такая ситуация аналогична проверке единиц измерения в научных расчетах (например, бессмысленно складывать величину в сантиметрах с величиной, измеряемой в килограммах).

значения	символы
типичные значения	'a' '\n'

Встроенный тип данных Java char

А теперь мы более подробно рассмотрим базовые встроенные типы, которые вы будете использовать чаще всего (строки, целые числа, вещественные числа в формате с плавающей точкой, значения true-false), вместе с примерами кода, демонстрирующими их использование. Чтобы понять, как использовать тип данных,

необходимо знать не только определенное для него множество значений, но и то, какие действия можно выполнять с этими значениями, языковые механизмы выполнения операций и правила определения литералов в коде.

Символы и строки

Тип `char` представляет отдельные алфавитно-цифровые символы или знаки. Всего существует 216 разных значений `char`, но обычно мы ограничиваемся теми, которые используются для представления букв, числовых знаков и пропусков (табуляции, символы новой строки и т. д.). Литерал типа `char` записывается в виде символа, заключенного в одинарные кавычки: например, литерал `'a'` представляет букву `a`. Для символов табуляции, новой строки, обратного слеша, одинарной кавычки и двойной кавычки используются специальные *служебные последовательности* (или *escape-последовательности*)[1] `\t`, `\n`, `\\`, `\'` и `\"` соответственно. Символы кодируются 16-разрядными целыми числами по схеме кодировки, называемой *Юникодом* (Unicode), но также существуют служебные последовательности для определения специальных символов, отсутствующих на клавиатуре (см. сайт книги). Обычно с символами непосредственно не выполняются никакие действия, кроме присваивания значений переменным.

Тип `String` представляет последовательности символов. Литерал типа `String` представляет собой последовательность символов, заключенную в двойные кавычки, например `"Hello, World"`. Тип данных `String` не относится к примитивным типам, хотя Java иногда обходится с ним как с примитивным типом. Например, оператор *конкатенации* (*сцепления*) (+) получает два операнда типа `String` и создает третье значение типа `String`, полученное присоединением символов второго операнда к символам первого операнда.

значения	последовательности символов
типичные литералы	`"Hello, World"` `" * "`
операции	конкатенация
оператор	+

Встроенный тип данных Java String

Операция конкатенации (в сочетании с возможностью объявления переменных `String` и их использования в выражениях и командах присваивания) обладает достаточно мощными возможностями для решения нетривиальных вычислительных задач. Например, программа `Ruler` (листинг 1.2.1) строит таблицу значений функции, описывающей относительные длины делений на линейке. Обратите внимание, как просто строится короткая программа, выдающая большой объем выходных данных. Выполняя очевидное масштабирование этой программы для вывода пяти строк, шести строк, семи строк и т. д., вы увидите, что добавление всего двух строк кода в программу приводит к удвоению размера выходных данных. Говоря конкретнее, если программа выводит `n` строк данных, `n`-я строка содержит $2^n - 1$ чисел.

[1] Исторически сложившееся название. Словом escape было принято обозначать символ, модифицирующий смысл следующих за ним символов: некоторые специальные значения, комбинации, команды и т. п. — *Примеч. науч. ред.*

Листинг 1.2.1. Конкатенация строк

```
public class Ruler
{
   public static void main(String[] args)
   {
      String ruler1 = "1";
      String ruler2 = ruler1 + " 2 " + ruler1;
      String ruler3 = ruler2 + " 3 " + ruler2;
      String ruler4 = ruler3 + " 4 " + ruler3;
      System.out.println(ruler1);
      System.out.println(ruler2);
      System.out.println(ruler3);
      System.out.println(ruler4);
   }
}
```

Программа выводит относительные длины делений на линейке. n-я строка вывода содержит относительные длины делений на линейке, разбитой на интервалы в $1/2^n$ дюйма. Например, четвертая строка вывода содержит относительные длины делений, обозначающих интервалы в 1/16 дюйма на линейке.

```
% javac Ruler.java
% java Ruler
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

Размеры делений для $n = 4$

Например, если вы сделаете программу для вывода 30 строк, количество выводимых чисел превысит 1 *миллиард*.

выражение	значение
`"Hi, " + "Bob"`	`"Hi, Bob"`
`"1" + " 2 " + "1"`	`"1 2 1"`
`"1234" + " + " + "99"`	`"1234 + 99"`
`"1234" + "99"`	`"123499"`

Типичные выражения со значениями типа `String`

Операция конкатенации часто (с большим отрывом от других случаев) применяется для вывода результатов вычислений с помощью метода `System.out.println()`. Например, мы можем упростить программу `UseArgument` (листинг 1.1.2), заменив три строки `main()` одной:

```
System.out.println("Hi, " + args[0] + ". How are you?");
```

Мы начали с типа `String` именно потому, что он понадобится нам для вывода (и для работы с аргументами командной строки) в программах, работающих не только со

строками, но и с другими типами данных. А сейчас рассмотрим два удобных механизма Java для преобразования чисел в строки и строк в числа.

Преобразование чисел в строки для вывода

Как упоминалось в начале этого раздела, встроенный тип `String` в Java подчиняется особым правилам. Одно из этих специальных правил заключается в том, что значение любого типа может быть легко преобразовано в значение `String`: каждый раз, когда при использовании оператора `+` один из операндов относится к типу `String`, Java автоматически преобразует другой операнд к типу `String`. Результатом такой операции становится значение `String`, состоящее из символов первого операнда, за которыми следуют символы второго операнда. Например, результаты выполнения следующих двух фрагментов кода:

```
String a = "1234";          String a = "1234";
String b = "99";            int b = 99;
String c = a + b;           String c = a + b;
```

полностью эквивалентны: переменной `c` присваивается значение `"123499"`. Этот механизм автоматического преобразования широко применяется для формирования значений типа `String`, используемых с вызовами `System.out.print()` и `System.out.println()`. Например:

```
System.out.println(a + " + " + b + " = " + c);
```

Если `a`, `b` и `c` — переменные типа `int` со значениями `1234`, `99` и `1333` соответственно, будет выведена строка `1234 + 99 = 1333`.

Преобразование строк в числа (входные значения)

Java также предоставляет библиотечные методы для преобразования строковых значений, вводимых в аргументах командной строки, в числовые значения для примитивных типов. Для этой цели используются библиотечные методы Java `Integer.parseInt()` и `Double.parseDouble()`. Например, вызов `Integer.parseInt("123")` в тексте программы эквивалентен вводу целочисленного литерала `123`. Если пользователь вводит `123` в первом аргументе командной строки, код `Integer.parseInt(args[0])` преобразует значение `"123"` типа `String` в значение `123` типа `int`. Несколько примеров такого рода встречаются в программах этого раздела.

Наше представление о программах Java как о «черных ящиках», получающих строковые аргументы и производящих строковые результаты, остается действительным, но теперь мы можем интерпретировать эти строки как числа и взять их за основу для содержательных вычислений.

Целые числа

Тип `int` представляет целые (натуральные) числа в диапазоне от `-2147483648` (-2^{31}) до `2147483647` ($2^{31} - 1$). Эти границы обусловлены тем фактом, что целые числа

в двоичном формате представляются 32 двоичными цифрами, или *разрядами*; всего существуют 2^{32} возможных значения. (Термин «*двоичная цифра*» — 0 или 1 — встречается в информатике настолько часто, что для него почти всегда используется сокращение «*бит*».) Диапазон возможных значений `int` асимметричен, потому что нуль относится к положительным значениям. В разделе «Вопросы и ответы» в конце этого раздела представление чисел рассматривается более подробно, а в текущем контексте достаточно знать, что `int` — всего лишь конечное множество значений из заданного диапазона. Литералы типа `int` задаются последовательностью десятичных цифр от 0 до 9 (которая в десятичной интерпретации относится к определенному диапазону). Тип `int` часто используется в программах, потому что он естествен для многих алгоритмов.

выражение	значение	комментарий
99	99	целочисленный литерал
+99	99	знак положительного числа
-99	-99	знак отрицательного числа
5 + 3	8	сложение
5 - 3	2	вычитание
5 * 3	15	умножение
5 / 3	1	целая часть деления
5 % 3	2	остаток от деления
1 / 0		ошибка времени выполнения
3 * 5 - 2	13	* имеет более высокий приоритет
3 + 5 / 2	5	/ имеет более высокий приоритет
3 - 5 - 2	-4	левосторонняя ассоциативность
(3 - 5) - 2	-4	более правильный стиль
3 - (5 - 2)	0	однозначная интерпретация

Типичные выражения int

Стандартные арифметические операторы сложения/вычитания (`+` и `-`), умножения (`*`), деления (`/`) и вычисления остатка от деления (`%`) для типа данных `int` являются для Java встроенными. Эти операторы получают два операнда типа `int` и выдают результат типа `int` с одним важным исключением — деление на нуль или вычисление остатка от деления на нуль невозможно. Эти операции знакомы нам по средней школе (помните, что все результаты должны быть целыми числами): для двух целочисленных значений `a` и `b` значение `a / b` указывает, сколько раз `b` укладывается в `a` (*без учета дробной части*), а значение `a % b` равно остатку от деления `a` на `b`. Например, значение `17 / 3` равно 5, а значение `17 % 3` равно `2`. Результаты типа `int`, возвращаемые арифметическими операциями, в общем соответствуют ожиданиям, кроме одного: если результат слишком велик для размещения в 32-разрядном

Листинг 1.2.2. Целочисленное умножение и деление

```
public class IntOps
{
   public static void main(String[] args)
   {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      int p = a * b;
      int q = a / b;
      int r = a % b;
      System.out.println(a + " * " + b + " = " + p);
      System.out.println(a + " / " + b + " = " + q);
      System.out.println(a + " % " + b + " = " + r);
      System.out.println(a + " = " + q + " * " + b + " + " + r);
   }
}
```

Арифметические операции с целыми числами встроены в Java. Основная часть кода обеспечивает ввод и вывод данных; собственно вычисления выполняются в середине программы при inline-инициализации переменных p, q и r.

```
% javac IntOps.java
% java IntOps 1234 99
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
1234 = 12 * 99 + 46
```

представлении, он усекается строго определенным способом[1]. Это явление называют *переполнением*. В общем случае необходимо следить за тем, чтобы такой результат не был ошибочно интерпретирован в коде. Пока мы будем выполнять вычисления с небольшими числами, поэтому вам не придется беспокоиться об этих граничных условиях.

значения	целые числа в диапазоне от -2^{31} до $+2^{31}-1$					
типичные значения	1234 99 0 1000000					
операции	знак	сложение	вычитание	умножение	деление	остаток
операторы	+ -	+	-	*	/	%

Встроенный тип данных Java `int`

Программа в листинге 1.2.2 демонстрирует три основные операции с целыми числами (умножение, деление, вычисление остатка). Также она показывает, как использовать `Integer.parseInt()` для преобразования значений типа `String` из

[1] В зависимости от дополнительных факторов может происходить «заворачивание» значения к 0 из положительной области в отрицательную (или наоборот). — *Примеч. науч. ред.*

командной строки в значения `int` и как автоматическое преобразование типа превращает значения `int` в `String` при выводе.

В языке Java есть еще три встроенных типа для работы с целыми числами. Типы `long`, `short` и `byte` похожи на `int`, но для представления числа они используют 64, 16 и 8 бит соответственно, поэтому и диапазоны допустимых значений у них тоже другие. Программисты используют тип `long` для работы с большими целыми числами, а два других — для экономии места. На сайте книги имеется таблица с максимальными и минимальными значениями для всех типов, хотя вы можете рассчитать их самостоятельно по их разрядности.

Числа с плавающей точкой

Тип `double` предназначен для чисел *с плавающей точкой*, применяемых в научных и коммерческих приложениях. Внутреннее представление чисел напоминает экспоненциальную («научную») запись, что позволяет хранить числа в огромном диапазоне. Числа с плавающей точкой используются для представления вещественных чисел, но определенно не эквивалентны им! Количество вещественных чисел бесконечно, тогда как любое дискретное представление, используемое в цифровых компьютерах, способно представить лишь конечное множество чисел с плавающей точкой. Впрочем, числа с плавающей точкой обеспечивают достаточно хорошее приближенное представление вещественных чисел, чтобы их можно было использовать в приложениях. Но все же программисту часто приходится учитывать тот факт, что точное представление числа возможно не всегда.

выражение	*значение*
`3.141 + 2.0`	`5.141`
`3.141 - 2.0`	`1.111`
`3.141 / 2.0`	`1.5705`
`5.0 / 3.0`	`1.6666666666666667`
`10.0 % 3.141`	`0.577`
`1.0 / 0.0`	`Infinity`
`Math.sqrt(2.0)`	`1.4142135623730951`
`Math.sqrt(-1.0)`	`NaN`

Типичные выражения со значениями типа `double`

Литералы типа `double` записываются в виде последовательности цифр с точкой. Например, литерал `3.14159` представляет приближенное значение числа π из шести цифр. Также возможно определить литерал `double` в записи, напоминающей экспоненциальную: литерал `6.022e23` представляет число $6{,}022 \times 10^{23}$. Как и для целых чисел, эти соглашения могут использоваться как для ввода литералов с плавающей точкой в программах, так и для передачи чисел с плавающей точкой в строковых аргументах командной строки.

Листинг 1.2.3. Вычисление корней квадратного уравнения

```
public class Quadratic
{
   public static void main(String[] args)
   {
      double b = Double.parseDouble(args[0]);
      double c = Double.parseDouble(args[1]);
      double discriminant = b*b - 4.0*c;
      double d = Math.sqrt(discriminant);
      System.out.println((-b + d) / 2.0);
      System.out.println((-b - d) / 2.0);
   }
}
```

Программа вычисляет корни квадратного уравнения $x^2 + bx + c$. Например, корнями уравнения $x^2 - 3x + 2$ являются значения 1 и 2, так как полином можно записать в виде $(x - 1)(x - 2)$; корнями уравнения $x^2 - x - 1$ являются Φ и $1 - \Phi$, где Φ — пропорция золотого сечения; а корни $x^2 + x + 1$ не являются вещественными числами.

```
% javac Quadratic.java
% java Quadratic -3.0 2.0
2.0
1.0
```

```
% java Quadratic -1.0 -1.0
1.618033988749895
-0.6180339887498949
% java Quadratic 1.0 1.0
NaN
NaN
```

Для типа `double` определены арифметические операторы +, -, * и /. Кроме этих встроенных операторов, в библиотеке `Math` языка Java определены функции вычисления квадратного корня, тригонометрические, логарифмические/экспоненциальные и другие, часто используемые при работе с числами в формате с плавающей точкой. Чтобы использовать одну из таких функций в выражении, укажите ее имя со списком аргументов, заключенных в круглые скобки. Например, результатом вычисления `Math.sqrt(2.0)` является значение типа `double`, которое приблизительно равно квадратному корню из 2. Этот механизм рассматривается в разделе 2.1, а более подробная информация о библиотеке `Math` приведена в конце этого раздела.

значения	вещественные числа (в соответствии со стандартом IEEE 754)			
типичные литералы	3.14159	6.022e23	2.0	1.4442135623730951
операции	сложение	вычитание	умножение	деление
операторы	+	-	*	/

Встроенный тип данных Java double

Одна из первых проблем, с которыми сталкивается программист при работе с числами с плавающей точкой, — проблема *точности*. Например, при выводе результата `5.0/2.0`, как и следовало ожидать, выводится `2.5`, но при выводе результата `5.0/3.0` выводится `1.6666666666666667`. В разделе 1.5 описан механизм управления

количеством значащих цифр при выводе, используемый в Java. А до того мы будем использовать формат вывода Java по умолчанию.

Результатом вычислений может быть одно из специальных значений, называемых `Infinity` («бесконечность», используется, если число слишком велико для представления) и `NaN` («Not a number» — «нечисло», используется, если результат вычисления не определен или невозможен). В вычислениях, в которых задействованы эти значения, приходится учитывать бесчисленные нюансы, но вы уже сейчас можете использовать `double` естественным образом и писать программы Java для любых вычислений. Например, программа в листинге 1.2.3 демонстрирует применение значений `double` для вычисления корней квадратного уравнения. Примеры в конце этого раздела дополнительно иллюстрируют сказанное.

Как и в случае с типами `long`, `short` и `byte` для целых чисел, у вещественных чисел также существует другое представление — тип `float`. Программисты иногда используют `float` для экономии памяти, если точность не столь важна. Тип `double` позволяет хранить около 15 значащих цифр; возможности типа `float` ограничиваются примерно 7 цифрами. В этой книге `float` не используется.

Boolean

значения	`true` или `false`		
литералы	`true` `false`		
операции	`and`	`or`	`not`
операторы	`&&`	`||`	`!`

Встроенный тип данных Java boolean

Тип `boolean` представляет логические значения. Их всего два: «истина» (`true`) и «ложь» (`false`). Любая переменная типа `boolean` содержит одно из этих двух значений, операнды и результат любой логической («булевой») операции тоже принимают одно из этих двух значений. Их внешняя простота обманчива — значения типа `boolean` лежат в основе всей информатики.

Самые важные операции, определенные для `boolean`, — логическое AND (И, конъюнкция) `&&`, логическое OR (ИЛИ, дизъюнкция) `||` и логическое NOT (НЕ, отрицание) `!` — имеют знакомые определения.

- Выражение `a && b` истинно (`true`) только в том случае, если истинны оба операнда, и ложно (`false`), если ложен хотя бы один из операндов.
- Выражение `a || b` ложно, если ложны оба операнда, и истинно, если истинен хотя бы один из операндов.
- Выражение `!a` истинно, если значение `a` ложно, и ложно при истинном `a`.

Хотя эти определения интуитивно понятны, полезно перечислить все комбинации операндов для каждой операции в таблицах, называемых *таблицами истинности*. Операция отрицания имеет только один операнд: ее значение для каждого из двух возможных значений операнда указано во втором столбце. Операции логических AND и OR имеют по два операнда; возможны четыре разные комбинации; значения операции для всех комбинаций перечислены в двух правых столбцах.

a	!a
true	false
false	true

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Таблицы истинности операций типа `boolean`

Объединяя эти операторы с круглыми скобками, можно создавать выражения произвольной сложности, каждое из которых задает определенную логическую операцию. Часто разные выражения оказываются эквивалентными. Так, выражения `(a && b)` и `!(!a || !b)` эквивалентны.

a	b	a && b	!a	!b	!a || !b	!(!a || !b)
false	false	false	true	true	true	false
false	true	false	true	false	true	false
true	false	false	false	true	true	false
true	true	true	false	false	false	true

Таблицы истинности доказывают, что выражения a && b и !(!a || !b) эквивалентны

Дисциплина, изучающая подобные манипуляции с выражениями, называется *булевой алгеброй*. Эта область математики чрезвычайно важна для компьютерных технологий; она играет исключительную роль в проектировании и работе компьютерного оборудования, а также становится отправной точкой для теории организации вычислений. В текущем контексте нас интересуют логические выражения, потому что мы можем использовать их для управления поведением программ. Условие, от которого зависит выполнение программы, определяется в виде логического выражения; программист пишет код, который выполняет один набор команд, если выражение истинно, и другой набор команд — если условие ложно. Механике такой проверки посвящен раздел 1.3.

Сравнения

Некоторые операторы получают операнды одного типа и выдают результат другого типа. Самые важные операторы такого рода — операторы сравнения `==`, `!=`, `<`, `<=`, `>` и `>=`, которые определяются для каждого примитивного числового типа и выдают результат `boolean`. Так как любые операции определяются только в отношении некоторого типа данных, каждое из этих обозначений подразумевает много операций, по одной для каждого типа данных (оба операнда должны относиться к одному типу).

дискриминант не отрицателен?	`(b*b - 4.0*a*c) >= 0.0`
начало века?	`(year % 100) == 0`
действительный номер месяца	`(month >= 1) && (month <= 12)`

Типичные выражения сравнения

Листинг 1.2.4. Високосный год

```
public class LeapYear
{
   public static void main(String[] args)
   {
      int year = Integer.parseInt(args[0]);
      boolean isLeapYear;
      isLeapYear = (year % 4 == 0);
      isLeapYear = isLeapYear && (year % 100 != 0);
      isLeapYear = isLeapYear || (year % 400 == 0);
      System.out.println(isLeapYear);
   }
}
```

Программа проверяет, что целое число представляет високосный год по григорианскому календарю. Год является високосным, если он нацело делится на 4 (как, например, 2004), но при этом не делится нацело на 100 (как, например, 1900); впрочем, если при этом год делится нацело на 400, он все же является високосным (например, 2000).

```
% javac LeapYear.java
% java LeapYear 2004
true
% java LeapYear 1900
false
% java LeapYear 2000
true
```

Даже если не углубляться в подробности представления чисел, становится ясно, что операции для разных типов достаточно сильно различаются. Например, одно дело — сравнить два числа `int` и проверить истинность выражения `(2 <= 2)`, и совсем другое — сравнить два значения `double` и проверить истинность выражения `(2.0 <= 0.002e3)`. Тем не менее такие операции достаточно четко определены и полезны, чтобы вы могли писать в программах проверки условий вида `(b*b - 4.0*a*c) >= 0.0`, часто встречающиеся в программах.

Приоритет операторов сравнения ниже, чем у арифметических операторов, и выше, чем у операторов типа `boolean`, поэтому в выражении вида `(b*b - 4.0*a*c) >= 0.0` круглые скобки не обязательны. Выражение `month >= 1 && month <= 12`, которое проверяет, что значение переменной `month` типа `int` лежит в диапазоне от 1 до 12, тоже может быть записано без круглых скобок. (Впрочем, круглые скобки лучше включить — это считается хорошим стилем программирования.)

Операции сравнения в сочетании с логическими операциями закладывают основу для принятия решений в программах Java. Пример их использования приведен в листинге 1.2.4; другие примеры встречаются в упражнениях в конце этого раздела. Что еще важнее, в разделе 1.3 будет продемонстрирована роль логических выражений в более сложных программах.

оператор	смысл	true	false
`==`	равно	`2 == 2`	`2 == 3`
`!=`	не равно	`3 != 2`	`2 != 2`
`<`	меньше	`2 < 13`	`2 < 2`
`<=`	меньше или равно	`2 <= 2`	`3 <= 2`
`>`	больше	`13 > 2`	`2 > 13`
`>=`	больше или равно	`3 >= 2`	`2 >= 3`

***Сравнения с операндами типа* `int` *и результатом типа* `boolean`**

Библиотечные методы и API

Как было показано выше, при решении многих задач программирования, помимо встроенных операторов, используются библиотечные методы Java. Количество библиотечных методов огромно. Во время изучения программирования вы начинаете пользоваться все бóльшим количеством библиотечных методов, и все же на первых порах лучше ограничиться относительно небольшим подмножеством методов. В этой главе мы уже использовали некоторые методы Java для вывода, для преобразования данных из одного типа в другой и для вычисления математических функций (библиотека Java `Math`). В последующих главах вы научитесь не только пользоваться другими методами, но и создавать и использовать собственные методы.

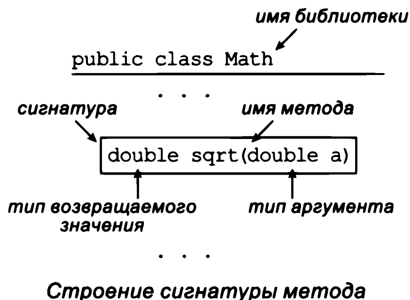
Для удобства мы будем приводить сводку библиотечных методов, которые вам необходимо знать, в таблицах следующего вида.

`void System.out.print(String s)`	*выводит значение s*
`void System.out.println(String s)`	*выводит значение s, за которым следует символ новой строки*
`void System.out.println()`	*выводит символ новой строки*

Примечание: аргументом может быть значение любого типа данных (которое автоматически преобразуется в* `String`*).

Библиотечные методы Java для вывода строк на терминал

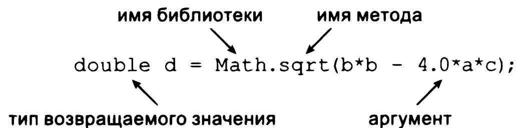
Такая сводка называется *интерфейсом прикладного программирования* (API, Application Programming Interface). Каждый метод описывается строкой с информацией, которую необходимо знать для его использования. Форма записи в этих таблицах *отличается* от обращений к этим методам в коде программ; она называется *сигнатурой* (или *прототипом*) метода. Сигнатура определяет тип аргументов



Строение сигнатуры метода

(*формальные* параметры), имя метода и тип результата, вычисляемого методом (*возвращаемое значение*).

Чтобы вызвать метод в своем коде, вы указываете его имя, за которым следуют аргументы, заключенные в круглые скобки и разделенные запятыми. Мы говорим, что во время выполнения вашей программы на языке Java метод *вызывается* с заданными значениями аргументов (*фактические* параметры) и *возвращает* значение. Вызов метода является выражением, так что вызовы методов можно использовать точно так же, как вы используете переменные и литералы для построения более сложных выражений, например `Math.sin(x) * Math.cos(y)` и т. д. Аргумент также является выражением, что позволяет писать код вида `Math.sqrt(b*b - 4.0*a*c)`; компилятор Java знает, что имеется в виду, — он вычисляет выражение аргумента и передает полученное значение методу.



Использование библиотечного метода

В таблицах API на следующей странице перечислены некоторые часто используемые методы в библиотеке `Math` языка Java, а также уже знакомые вам методы Java для вывода текста в терминальном окне и для преобразования строк в примитивные типы. В следующей таблице приведены примеры вызовов, использующих библиотечные методы.

Методы на следующей странице (кроме трех) *чистые* — для одних и тех же аргументов они всегда возвращают одно и то же значение, без каких-либо побочных эффектов. Метод `Math.random()` не является чистым, потому что он может возвращать разные значения при каждом вызове; методы `System.out.print()` и `System.out.println()` не являются чистыми, поскольку они имеют побочные эффекты — вывод строк на терминал. В описаниях API для методов с побочными эффектами будут использоваться глагольные конструкции; в остальных случаях для описания возвращаемого значения будет использоваться именная конструкция. Ключевое слово `void` обозначает метод, который не возвращает значения (главная цель такого метода — создание побочных эффектов).

Библиотека `Math` также определяет константы `Math.PI` (число π) и `Math.E` (число e), которые вы можете использовать в своих программах. Например, значение `Math.sin(Math.PI/2)` равно `1.0`, и значение `Math.log(Math.E)` равно `1.0` (потому что `Math.sin()` получает свои аргументы в радианах, а `Math.log()` реализует функцию натурального логарифма).

Эти описания API — типичный пример электронной документации, ставшей стандартом в современном программировании. Обширная электронная документация

вызов метода	библиотека	тип возвращаемого значения	значение
`Integer.parseInt("123")`	`Integer`	`int`	`123`
`Double.parseDouble("1.5")`	`Double`	`double`	`1.5`
`Math.sqrt(5.0*5.0 - 4.0*4.0)`	`Math`	`double`	`3.0`
`Math.log(Math.E)`	`Math`	`double`	`1.0`
`Math.random()`	`Math`	`double`	*random in* [0, 1)
`Math.round(3.14159)`	`Math`	`long`	`3`
`Math.max(1.0, 9.0)`	`Math`	`double`	`9.0`

Примеры типичных вызовов библиотечных методов Java

`public class Math`

`double abs(double a)`	*абсолютное значение (модуль) a*
`double max(double a, double b)`	*большее из значений a и b*
`double min(double a, double b)`	*меньшее из значений a и b*

Примечание 1: abs(), max() и min() также определяются для типов `int`, `long` *и* `float`.

`double sin(double theta)`	*синус* `theta`
`double cos(double theta)`	*косинус* `theta`
`double tan(double theta)`	*тангенс* `theta`

Примечание 2: углы задаются в радианах.
Для преобразования угловых единиц используйте методы `toDegrees()` *и* `toRadians()`.

Примечание 3: обратные тригонометрические функции вычисляются методами `asin()`, `acos()` *и* `atan()`

`double exp(double a)`	*экспонента (e в степени a)*
`double log(double a)`	*натуральный логарифм ($\log_e a$, или ln a)*
`double pow(double a, double b)`	*возведение a в степень b (a^b)*
`long round(double a)`	*округление a до ближайшего целого*
`double random()`	*случайное число в диапазоне [0, 1)*
`double sqrt(double a)`	*квадратный корень из a*
`double E`	*Число e (константа)*
`double PI`	*Число π (константа)*

За информацией о других функциях обращайтесь на сайт книги.

Некоторые методы библиотеки `Math`

`void System.out.print(String s)`	*выводит значение s*
`void System.out.println(String s)`	*выводит значение s, за которым следует символ новой строки*
`void System.out.println()`	*выводит символ новой строки*

Библиотечные методы Java для вывода строк на терминал

`int Integer.parseInt(String s)`	*преобразование s в значение типа* `int`
`double Double.parseDouble(String s)`	*преобразование s в значение типа* `double`
`long Long.parseLong(String s)`	*преобразование s в значение типа* `long`

Библиотечные методы Java для преобразования строк в примитивные типы

по Java API постоянно используется профессиональными программистами, и вы можете загрузить ее с сайта Java или с сайта книги. Чтобы разобраться в коде, приведенном в книге, или написать похожий код, не обязательно обращаться к электронной документации, потому что в тексте книги представлены и описаны все библиотечные методы, используемые в API, а их сводка приводится в завершающем разделе. Что еще важнее, в главах 2 и 3 вы узнаете, как разработать собственные API и реализовать методы для собственных целей.

Преобразование типов

Одно из основных правил современного программирования: разработчик всегда должен учитывать тип данных, обрабатываемых программой[1]. Только зная тип, вы сможете точно определить, какое множество значений может принимать переменная, какие литералы вы сможете использовать и какие операции выполнять. Например, представьте, что вы хотите вычислить среднее арифметическое четырех чисел 1, 2, 3 и 4. Естественно, в голову сразу приходит выражение `(1 + 2 + 3 + 4) / 4`, но из-за правил преобразования типов его результатом является значение 2 типа `int` вместо значения `2.5` типа `double`. Проблема возникает из-за того, что операнды относятся к типу `int`, а результат должен относиться к типу `double`, поэтому в какой-то момент потребуется преобразовать `int` в `double`. В Java такое преобразование может выполняться несколькими способами.

Неявное преобразование типов

Значение `int` может использоваться повсюду, где ожидается значение `double`, потому что Java автоматически преобразует целые числа в `double` там, где это уместно. Например, результат `11*0.25` равен `2,75`, потому что `0.25` относится к типу `double`, а оба операнда должны быть однотипными; поэтому `11` преобразуется в `double`, а результатом деления двух `double` является `double`. Другой пример: результат `Math.sqrt(4)` равен `2.0`, потому что 4 преобразуется в `double`, как того ожидает метод `Math.sqrt()`, возвращающий значение `double`. Такое преобразование называется ***автоматическим повышением***. Автоматическое повышение в данном случае допустимо, потому что ваша цель ясна, а преобразование не приводит к потере информации. Напротив, преобразование, которое может привести к потере информации (например, присваивание значения `double` переменной типа `int`), приведет к ошибке компиляции.

Явное преобразование

В Java существуют встроенные правила преобразования для примитивных типов, которые можно использовать, когда вы знаете о возможной потере информации.

[1] В ряде языков явная типизация данных не предусмотрена или не является обязательной. Однако, по большому счету, учитывать фактический тип данных и его преобразования необходимо всегда. — *Примеч. науч. ред.*

Листинг 1.2.5. Преобразование типа для получения случайного целого числа

```
public class RandomInt
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      double r = Math.random();   // Равномерное распределение от 0.0 до 1.0
      int value = (int) (r * n);  // Равномерное распределение от 0 до n - 1
      System.out.println(value);
   }
}
```

Метод Java Math.random() генерирует случайное число r в диапазоне от 0.0 (включительно) до 1.0 (без включения). Затем r умножается на аргумент командной строки n для получения случайного числа, большего или равного 0 и меньшего n; далее преобразование типа усекает результат до целого значения в диапазоне от 0 до n – 1.

```
% javac RandomInt.java
% java RandomInt 1000
548
% java RandomInt 1000
141
% java RandomInt 1000000
135032
```

Вы должны явно обозначить свои намерения, воспользовавшись механизмом *явного преобразования типов*. Чтобы преобразовать выражение к другому примитивному типу, поставьте перед ним имя нужного типа в круглых скобках. Например, выражение `(int) 2.71828` преобразует `double` в `int`, а его результатом является `int` со значением 2. Методы, определенные для явных преобразований, отбрасывают лишнюю информацию наиболее разумным способом (за полной информацией обращайтесь на сайт книги). Например, при преобразовании числа с плавающей точкой в целое число теряется дробная часть, для чего производится округление по направлению к нулю. Программа `RandomInt` (листинг 1.2.5) дает пример использования преобразования для практических целей.

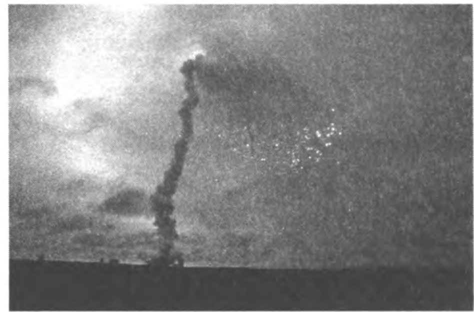
Явное преобразование типа имеет более высокий приоритет, чем арифметические операции, — любое преобразование применяется к значению, следующему сразу же после него. Например, в записи `int value = (int) 11 * 0.25` преобразование бесполезно: литерал 11 уже является целым числом, так что преобразование `(int)` ни на что не влияет. В этом случае компилятор выдает ошибку возможной потери точности, потому что преобразование полученного значения (2.75) в `int` для присваивания приводит к потере информации. Сообщение помогает найти ошибку, потому что, скорее всего, в этом коде имеется в виду вычисление `(int) (11 * 0.25)`, значение которого равно `2`, а не `2.75`.

выражение	тип выражения	значение выражения
`(1 + 2 + 3 + 4) / 4.0`	`double`	`2.5`
`Math.sqrt(4)`	`double`	`2.0`
`"1234" + 99`	`String`	`"123499"`
`11 * 0.25`	`double`	`2.75`
`(int) 11 * 0.25`	`double`	`2.75`
`11 * (int) 0.25`	`int`	`0`
`(int) (11 * 0.25)`	`int`	`2`
`(int) 2.71828`	`int`	`2`
`Math.round(2.71828)`	`long`	`3`
`(int) Math.round(2.71828)`	`int`	`3`
`Integer.parseInt("1234")`	`int`	`1234`

Типичные преобразования типа

Методы для явного преобразования типов

Также можно вызвать метод, который получает аргумент одного типа (преобразуемое значение) и создает результат другого типа. Мы уже использовали библиотечные методы `Integer.parseInt()` и `Double.parseDouble()` для преобразования значений `String` в значения типов `int` и `double` соответственно[1]. Существует много других методов для преобразования между другими типами. Например, библиотечный метод `Math.round()` получает аргумент типа `double` и возвращает результат `long`: целое число, ближайшее к аргументу. Таким образом, например, результаты `Math.round(3.14159)` и `Math.round(2.71828)` относятся к типу `long` и имеют одинаковое значение (`3`). Если вы захотите привести результат `Math.round()` к типу `int`, придется использовать явное преобразование.



Взрыв ракеты «Ариан-5»

Новичков преобразования типов часто раздражают, но опытные программисты знают, что внимание к типам данных — ключ к успеху в программировании. А иногда оно способно предотвратить катастрофу: например, в знаменитой аварии 1985 года французская ракета взорвалась в воздухе из-за ошибки преобразования типов. Возможно, ошибка в вашей программе не приведет к взрыву, и все же не жалейте времени на то, чтобы разобраться с преобразованием типов. Написав несколько программ, вы увидите, что хорошее понимание типов данных не только способствует созданию более компактного кода, но и позволяет более четко выразить намерения и избежать коварных ошибок в ваших программах.

---

[1] Фактически эти методы не преобразуют тип, а извлекают числовые данные из текста (строки); это подчеркивается наличием «parse» в их именах. — *Примеч. науч. ред.*

Выводы

Тип данных представляет собой множество значений и множество операций с этими значениями. В языке Java восемь примитивных типов данных: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` и `double`. В коде Java для выполнения операций, связанных с каждым типом, используются операторы и выражения вроде тех, которые знакомы нам по математическим выражениям. Тип `boolean` предназначен для вычислений с логическими значениями `true` и `false`; тип `char` представляет символ, вводимый с клавиатуры; шесть других числовых типов используются для обработки числовых данных. В этой книге чаще всего используются `boolean`, `int` и `double`; типы `short` и `float` практически не встречаются. Другой часто используемый тип данных, `String`, не относится к примитивным, но в Java существуют встроенные средства для работы со `String`, похожие на аналогичные средства для примитивных типов.

Программируя на Java, вы должны учитывать, что каждая операция определена только в контексте своего типа данных (поэтому может возникнуть необходимость в преобразовании типов), а все типы имеют конечное число допустимых значений (а значит, иногда приходится обходиться неточными результатами).

Тип `boolean` и его операции `&&`, `||` и `!` в сочетании с операторами сравнения `==`, `!=`, `<`, `>`, `<=` и `>=` закладывают основу для принятия решений в программах Java. А конкретнее, выражения `boolean` используются для управления условными (`if`) и циклическими (`for` и `while`) конструкциями Java, которые будут подробно описаны в следующем разделе.

Благодаря числовым типам и библиотекам мы можем использовать Java в режиме калькулятора. В записи арифметических выражений используются операторы `+`, `-`, `*`, `/` и `%`, а также методы Java из библиотеки `Math`. Хотя приведенные в этом разделе программы достаточно примитивны на фоне того, что вы сможете делать после следующего раздела, этот класс программ вполне полезен сам по себе. Примитивные типы и базовые математические функции широко применяются в программировании на Java[1], поэтому время, проведенное за их изучением, наверняка не пропадет даром.

Вопросы и ответы (строки)

В. Как в Java организовано хранение строк?

О. Строки представляют собой последовательности символов в кодировке Юникод (современный стандарт кодирования текста). Юникод содержит более 100 000 разных символов из более чем 100 разных языков, а также математические, музыкальные и другие символы.

[1] Как, вообще говоря, и на практически любом другом языке. — *Примеч. науч. ред.*

В. Можно ли использовать операторы < и > для сравнения значений `String`?

О. Нет. Эти операторы определены только для значений примитивных типов.

В. А как насчет `==` и `!=` ?

О. Да, но результат может быть не тем, на который вы рассчитываете, из-за особенностей этих операторов для непримитивных типов. Например, переменная `String` и ее значение — не одно и то же. Выражение `"abc" == "ab" + x` ложно, если `x` относится к типу `String` и имеет значение `"c"`, потому что два операнда хранятся в разных местах памяти (хотя их значения и совпадают). Эти различия очень важны, как вы узнаете, когда мы будем рассматривать их более подробно в разделе 3.1.

В. Как сравнить две строки подобно тому, как мы делаем это при поиске слова в алфавитном указателе или словаре?

О. Обсуждение типа `String` и связанных с ним методов откладывается до раздела 3.1, когда мы займемся объектно-ориентированным программированием. А до того времени придется ограничиться операцией конкатенации строк.

В. Как задать строковый литерал, который не помещается в одной строке?

О. Никак. Вместо этого придется разделить строковый литерал на независимые строковые литералы и объединить их посредством конкатенации, как в следующем примере:

```
String dna = "ATGCGCCCACAGCTGCGTCTAAACCGGACTCTG" +
             "AAGTCCGGAAATTACACCTGTTAG";
```

Вопросы и ответы (целые числа)

В. Как в Java организовано хранение целых чисел?

О. Простейшее представление используется для небольших положительных чисел. Для хранения любого такого числа в блоке памяти фиксированного размера используется *двоичная система счисления*.

В. Что такое «двоичная система счисления»?

О. В двоичной системе целое число представляется последовательностью двоичных разрядов — *битов*. Бит — отдельная двоичная цифра, 0 или 1, — лежит в основе всей системы представления информации в компьютерах. В данном случае биты являются коэффициентами степеней 2. А именно, последовательность битов $b_n b_{n-1} \ldots b_2 b_1 b_0$ представляет целое число

$$b_n 2^n + b_{n-1} 2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

Например, последовательность 1100011 представляет целое число

$$99 = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1.$$

Более привычная *десятичная система счисления* работает, в общем-то, так же, только в ней используются цифры от 0 до 9 со степенями 10. Преобразование числа в двоичную форму — интересная вычислительная задача, которая будет рассмотрена в следующем разделе. В Java значения `int` 32-разрядные. Например, десятичное число 99 может быть представлено 32 битами `00000000000000000000000001100011`.

В. А как же отрицательные числа?

О. Отрицательные числа записываются по схеме, известной как *дополнительный код*, которую нам незачем рассматривать подробно. Из-за этого значения `int` в Java лежат в диапазоне от `-2147483648` (-2^{31}) до `2147483647` ($2^{31} - 1$). Одно удивительное последствие применения этой записи заключается в том, что при переполнении и выходе очень больших значений `int` за верхнюю границу (`2147483647`) значение становится отрицательным — происходит так называемое заворачивание. Если вы еще не знакомы с этим явлением, обратитесь к упражнению 1.2.10. В таких случаях безопасно использовать тип `int`, только если вы уверены в том, что целые значения содержат менее 10 цифр, и тип `long` — если целые значения содержат 10 и более цифр[1].

В. Значит, Java просто допускает переполнение и появление некорректных значений? Почему язык Java не использует автоматическую проверку переполнения?

О. Да, этот вопрос довольно часто возникает у программистов. Дело в том, что отсутствие такой проверки — одна из причин, по которым такие типы называются *примитивными*. Внимательность поможет вам предотвратить подобные проблемы. Еще раз напомним, что тип `int` может использоваться для малых чисел, но для миллиардных значений он не подойдет.

В. Чему равно значение `Math.abs(-2147483648)`?

О. –2147483648. Этот странный результат — типичный пример последствий целочисленного переполнения и представления в дополнительном коде.

В. Какой результат будет получен при вычислении выражений `1 / 0` и `1 % 0` в Java?

О. В обоих случаях генерируется исключение времени выполнения (деление на нуль).

В. Как вычисляется результат деления и остаток от деления для отрицательных чисел?

О. Частное `a / b` округляется в направлении 0; остаток `a % b` определяется так, чтобы `(a / b) * b + a % b` всегда было равно `a`. Например, результаты выражений `-14 / 3` и `14 / -3` равны `-4`, но `-14 % 3` равно `-2`, а `14 % -3` равно `2`. В некоторых других языках (включая Python) при делении отрицательных чисел применяются другие правила.

---

[1] Строго говоря, для 64-разрядных чисел «заворачивание» точно так же возможно, но намного больший диапазон значений делает это проблему не столь актуальной. — *Примеч. науч. ред.*

В. Почему значение `10 ^ 6` равно не 1 000 000, а 12?

О. Оператор `^` не выполняет возведение в степень, как можно было бы подумать. Это оператор поразрядного XOR («исключающего ИЛИ»), который вы вряд ли будете часто использовать. Вместо этого следует использовать литерал `1e6`. Также можно использовать вызов `Math.pow(10, 6)`, но при возведении 10 в заранее известную фиксированную степень такое решение нерационально.

Вопросы и ответы (числа с плавающей точкой)

В. Почему тип для представления вещественных чисел назван `float`?

О. Точка может «плавать» по разрядам, образующим вещественное число. Напротив, в целых числах точка (условная) располагается за последней значащей цифрой.

В. Как Java хранит вещественные числа в своем внутреннем представлении?

О. Java использует стандарт IEEE 754, который на большинстве современных компьютеров поддерживается аппаратно. По этому стандарту представление числа с плавающей точкой состоит из трех полей: знака, мантиссы и порядка. Если вас интересует эта тема, обращайтесь за подробностями к сайту книги. Стандарт IEEE 754 также указывает, как должны обрабатываться особые значения с плавающей точкой: положительный нуль, отрицательный нуль, положительная бесконечность, отрицательная бесконечность и «нечисло» (`NaN`). В частности, вычисления с плавающей точкой никогда не приводят к исключениям времени выполнения. Например, выражение `-0.0/3.0` дает результат `-0.0`, выражение `1.0/0.0` дает положительную бесконечность, а при вычислении `Math.sqrt(-2.0)` будет получен результат `NaN`.

В. Пятнадцати цифр для чисел с плавающей точкой вполне достаточно. Мне действительно нужно беспокоиться о точности?

О. Да, потому что вы привыкли к математике, основанной на вещественных числах, тогда как компьютеры всегда имеют дело с приближенными представлениями. Например, выражение `(0.1 + 0.1 == 0.2)` дает результат `true`, но выражение `(0.1 + 0.1 + 0.1 == 0.3)` дает результат `false`! Такие ловушки не так редко встречаются в научных вычислениях. Начинающим программистам лучше избегать сравнения двух чисел с плавающей точкой на строгое равенство.

В. Как инициализировать переменную типа `double` значением `NaN` или бесконечностью?

О. В Java для этой цели существуют встроенные константы: `Double.NaN`, `Double.POSITIVE_INFINITY` и `Double.NEGATIVE_INFINITY`.

В. Существуют ли в библиотеке Java Math функции для других тригонометрических функций: косеканса, секанса, котангенса?

О. Нет, но они легко вычисляются с использованием Math.sin(), Math.cos() и Math.tan(). При выборе функций, включаемых в API, приходится искать компромисс между удобством включения всех функций, которые могут понадобиться, и неудобством от поиска одной нужной функции в длинном списке. Удовлетворить всех не может ни один вариант, а создателям Java приходилось думать об интересах многих пользователей. Обратите внимание: даже в приведенных нами API существует некоторая избыточность. Например, вместо Math.tan(x) можно использовать выражение Math.sin(x)/Math.cos(x).

В. Такое количество цифр при выводе double только мешает. Можно ли приказать System.out.println() выводить только две или три цифры после точки?

О. Для задач такого рода необходимо повнимательнее присмотреться к методу, используемому для преобразования double в String. Библиотечная функция Java System.out.printf() предоставляет один из способов решения этой задачи. Она напоминает основные методы вывода в языке программирования C и многих других современных языках, как будет показано в разделе 1.5. Но до тех пор нам придется жить с лишними цифрами (что не так уж плохо, потому что это помогает привыкнуть к разным примитивным числовым типам).

Вопросы и ответы (переменные и выражения)

В. Что произойдет, если я забуду объявить переменную?

О. Компилятор пожалуется, когда вы будете ссылаться на эту переменную в выражении. Например, программа IntOpsBad почти совпадает с программой 1.2.2, но переменная p не объявлена (с типом int):

```
% javac IntOpsBad.java
IntOpsBad.java:7: error: cannot find symbol
      p = a * b;
      ^
  symbol:   variable p
  location: class IntOpsBad
IntOpsBad.java:10: error: cannot find symbol
      System.out.println(a + " * " + b + " = " + p);
                                                 ^
  symbol:   variable p
  location: class IntOpsBad
2 errors
```

Компилятор сообщает о двух ошибках, но на самом деле ошибка всего одна: в программе пропущено объявление p. Если вы забудете объявить часто используемую переменную, сообщений об ошибке может быть довольно много. В таких ситуациях

стоит исправить первую ошибку и проверить результат, прежде чем заниматься остальными.

В. Что произойдет, если я забуду инициализировать переменную?

О. Если вы попытаетесь использовать в выражении переменную, которая не была инициализирована, компилятор проверяет это условие и сообщает об ошибке.

В. Операторы `=` и `==` чем-то отличаются?

О. Да, и очень сильно! Первый — оператор присваивания, изменяющий значение переменной, а второй — оператор сравнения, который возвращает результат типа `boolean`. Если вы поняли этот ответ, это верный признак того, что вы поняли материал этого раздела. Подумайте, как бы вы объяснили различия своему другу.

В. Можно ли сравнить `double` с `int`?

О. Только с преобразованием типа, но помните, что Java обычно выполняет преобразование типа автоматически. Например, если `x` — значение типа `int` со значением 3, то выражение `(x < 3.1)` истинно — Java преобразует `x` в `double` (так как `3.1` — литерал типа `double`) перед выполнением сравнения.

В. Присвоит ли команда `a = b = c = 17;` значение 17 трем целочисленным переменным `a`, `b` или `c`?

О. Да. Решение работает, потому что команда присваивания в Java также является выражением (результат которого определяется значением в правой части), а оператор присваивания обладает правосторонней ассоциативностью. Ради хорошего стиля программирования мы не будем использовать *цепочечное присваивание* в этой книге.

В. Будет ли выражение `(a < b < c)` проверять, что значения трех целочисленных переменных `a`, `b` и `c` упорядочены строго по возрастанию?

О. Нет, это выражение не скомпилируется, потому что выражение `a < b` производит результат `boolean`, который затем сравнивается со значением `int`. Java не поддерживает цепочечные сравнения. Вместо этого следует использовать запись `(a < b && b < c)`.

В. Почему используется запись `(a && b)`, а не `(a & b)`?

О. В Java также существует оператор `&`, который может вам встретиться в учебном курсе программирования более высокого уровня.

В. Чему равно значение `Math.round(6.022e23)`?

О. Обзаведитесь полезной привычкой: напишите крошечную программу на Java, чтобы отвечать на такие вопросы самостоятельно (а заодно попробуйте понять, почему программа выдает именно такой результат).

В. Java называют языком со *статической типизацией*. Что это означает?

О. Термин «статическая типизация» означает, что типы всех переменных и всех выражений известны во время компиляции[1]. Java также проверяет и следит за соблюдением ограничений типов во время компиляции; например, программа не будет компилироваться, если вы попытаетесь сохранить значение типа `double` в переменной типа `int` или вызовете `Math.sqrt()` с аргументом `String`.

Упражнения

1.2.1. Предположим, `a` и `b` — переменные типа `int`. Что делает следующая последовательность команд?

```
int t = a; b = t; a = b;
```

1.2.2. Напишите программу, которая использует `Math.sin()` и `Math.cos()` и проверяет, что значение $\cos^2\theta + \sin^2\theta$ равно приблизительно 1 для любого значения θ, переданного в аргументе командной строки. Просто выведите значение. Почему оно не всегда в точности равно 1?

1.2.3. Предположим, `a` и `b` — переменные типа `boolean`. Покажите, что результат выражения `(!(a && b) && (a || b)) || ((a && b) || !(a || b))` равен `true`.

1.2.4. Предположим, `a` и `b` — переменные типа `int`. Упростите следующее выражение: `(!(a < b) && !(a > b))`.

1.2.5. Оператор исключающего OR `^` для операндов типа `boolean` по определению возвращает `true`, если операнды различны, и возвращает `false`, если они совпадают. Приведите таблицу истинности для этой функции.

1.2.6. Почему `10/3` возвращает `3`, а не `3.333333333`?

Решение. Так как оба значения, 10 и 3, являются целочисленными литералами, Java не видит необходимости в преобразовании типов и использует целочисленное деление. Если вы хотите, чтобы числа интерпретировались как значения `double`, используйте запись `10.0/3.0`. Если вы напишете `10/3.0` или `10.0/3`, Java выполнит неявное преобразование для получения того же результата.

1.2.7. Какой результат выводит каждая из следующих строк кода?

```
a. System.out.println(2 + "bc");
b. System.out.println(2 + 3 + "bc");
c. System.out.println((2+3) + "bc");
d. System.out.println("bc" + (2+3));
e. System.out.println("bc" + 2 + 3);
```

Объясните каждый результат.

---

[1] Альтернативой является *динамическая* типизация, при которой типы данных определяются (и автоматически преобразуются) непосредственно при обращении к ним. — *Примеч. науч. ред.*

1.2.8. Объясните, как использовать программу из листинга 1.2.3 для нахождения квадратного корня числа.

1.2.9. Какой результат выводит каждая из следующих команд?

```
a. System.out.println('b');
b. System.out.println('b' + 'c');
c. System.out.println((char) ('a' + 4));
```

Объясните каждый результат.

1.2.10. Допустим, переменная a объявляется командой `int a = 2147483647` (или эквивалентной константой `Integer.MAX_VALUE`). Какой результат выводит каждая из следующих команд?

```
a. System.out.println(a);
b. System.out.println(a+1);
c. System.out.println(2-a);
d. System.out.println(-2-a);
e. System.out.println(2*a);
f. System.out.println(4*a);
```

Объясните каждый результат.

1.2.11. Предположим, переменная a объявляется командой `double a = 3.14159`. Какой результат выводит каждая из следующих команд?

```
a. System.out.println(a);
b. System.out.println(a+1);
c. System.out.println(8/(int) a);
d. System.out.println(8/a);
e. System.out.println((int) (8/a));
```

Объясните каждый результат.

1.2.12. Опишите, что произойдет, если заменить `Math.sqrt` в программе 1.2.3 на `sqrt`.

1.2.13. Определите результат выражения `(Math.sqrt(2) * Math.sqrt(2) == 2)`.

1.2.14. Напишите программу, которая получает два положительных числа в аргументах командной строки и выводит `true`, если одно из этих чисел нацело делится на другое.

1.2.15. Напишите программу, которая получает три положительных числа в аргументах командной строки и выводит `false`, если одно из них больше или равно сумме других, или `true` в противном случае. (Примечание: программа проверяет, могут ли числа представлять длины сторон некоторого треугольника.)

1.2.16. Студент-физик получает неожиданные результаты при использовании кода

```
double force = G * mass1 * mass2 / r * r;
```

для вычисления значений по формуле $F = Gm_1m_2/r^2$. Объясните суть проблемы и исправьте ошибку.

1.2.17. Присвойте значение переменной a после выполнения каждой из следующих серий команд:

```
int a = 1;     boolean a = true;    int a = 2;
a = a + a;     a = !a;              a = a * a;
a = a + a;     a = !a;              a = a * a;
a = a + a;     a = !a;              a = a * a;
```

1.2.18. Напишите программу, которая получает два числа с плавающей точкой x и y в аргументах командной строки и выводит евклидово расстояние от точки (x, y) до точки (0, 0).

1.2.19. Напишите программу, которая получает два целых числа a и b в аргументах командной строки и выводит случайное целое число из диапазона от a до b включительно.

1.2.20. Напишите программу, которая выводит сумму двух случайных чисел от 1 до 6 (такие результаты могут быть получены при броске кубиков).

1.2.21. Напишите программу, которая получает число t в аргументе командной строки и выводит значение $\sin(2t) + \sin(3t)$.

1.2.22. Напишите программу, которая получает числа типа double x_0, v_0 и t в аргументах командной строки и выводит значение $x_0 + v_0t - g\,t^2/2$, где g — константа 9,80665. (*Примечание*: значение определяет путь в метрах, пройденный за t секунд брошенным вертикально вверх объектом, из исходной позиции x_0 со скоростью v_0 метров в секунду.)

1.2.23. Напишите программу, которая получает два целых числа m и d в аргументах командной строки и выводит true, если день d месяца m лежит в диапазоне от 20/03 до 20/06, или false в противном случае.

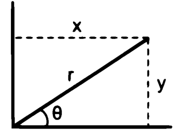
1.2.24. *Капитализация процентов.* Напишите программу, которая вычисляет и выводит сумму, которую вы получите через t лет при размещении P долларов под ежегодный процент r (непрерывно начисляемый). Искомое значение вычисляется по формуле Pe^{rt}.

1.2.25. *Охлаждение под действием ветра.* Для заданной температуры T (по шкале Фаренгейта) и скорости ветра v (в милях в час) Национальная метеорологическая служба вычисляет фактическую температуру (охлаждение под действием ветра) по формуле:

$$w = 35.74 + 0.6215\,T + (0.4275\,T - 35.75)\,v^{0.16}.$$

Напишите программу, которая получает два числа типа double temperature и velocity в аргументах командной строки и выводит величину охлаждения. Для вычисления a^b используйте Math.pow(a, b). *Примечание*: формула недействительна, если абсолютное значение T больше 50 или если v больше120 или меньше 3 (предполагается, что полученные значения лежат в этом диапазоне).

1.2.26. *Полярные координаты.* Напишите программу, которая преобразует декартовы координаты в полярные. Ваша программа должна получать два числа типа `double` в аргументах командной строки и выводить полярные координаты `r` и θ. Используйте метод `Math.atan2(y, x)` для вычисления арктангенса величины `y/x` в диапазоне от $-\pi$ до π.



Полярные координаты

1.2.27. *Случайные числа с гауссовским (нормальным) распределением.* Напишите программу `RandomGaussian` для вывода случайного числа `r` из *гауссовского распределения.* Для этого можно воспользоваться формулой Бокса — Мюллера

$$r = \sin(2\pi v)\,(-2\ln u)^{1/2},$$

где u и v — вещественные числа в диапазоне от 0 до 1, сгенерированные методом `Math.random()`.

1.2.28. *Проверка упорядоченности.* Напишите программу, которая получает три числа типа `double` `x`, `y` и `z` в аргументах командной строки и выводит `true`, если значения расположены строго по возрастанию или убыванию ($x < y < z$ или $x > y > z$), или `false` в противном случае.

1.2.29. *День недели.* Напишите программу, которая получает дату и выводит день недели, на который приходится эта дата. Ваша программа должна получать три значения типа `int` в аргументах командной строки: `m` (месяц), `d` (день) и `y` (год). Значение `m` равно 1 для января, 2 — для февраля и т. д. Программа выводит 0 для воскресенья, 1 — для понедельника, 2 — для вторника и т. д. Вычисление выполняется по следующим формулам из григорианского календаря:

$$y_0 = y - (14 - m) / 12$$

$$x = y_0 + y_0 / 4 - y_0 / 100 + y_0 / 400$$

$$m_0 = m + 12 \times ((14 - m) / 12) - 2$$

$$d_0 = (d + x + (31 \times m_0) / 12) \% 7$$

Пример: на какой день недели приходится 14 февраля 2000 года?

$$y_0 = 2000 - 1 = 1999$$

$$x = 1999 + 1999 / 4 - 1999 / 100 + 1999 / 400 = 2483$$

$$m_0 = 2 + 12 \times 1 - 2 = 12$$

$$d_0 = (14 + 2483 + (31 \times 12) / 12) \% 7 = 2500 \% 7 = 1$$

Ответ: понедельник.

1.2.30. *Случайные числа с равномерным распределением.* Напишите программу, которая выводит пять случайных чисел с равномерным распределением в диапазоне от 0 до 1, их среднее арифметическое, минимальное и максимальное значения. Используйте методы `Math.random()`, `Math.min()` и `Math.max()`.

1.2.31. *Проекция Меркатора.* Проекция Меркатора представляет собой равноугольную (сохраняющую углы между направлениями) картографическую проекцию, отображающую широту ϕ и долготу λ в прямоугольные координаты (x, y). Проекция Меркатора широко применяется — например, в навигационных картах и тех картах, которые вы распечатываете из Интернета. Проекция определяется формулами $x = \lambda - \lambda_0$ и $y = 1/2 \ln((1 + \sin \phi) / (1 - \sin \phi))$, где λ_0 — долгота точки в центре карты. Напишите программу, которая получает в командной строке λ_0, а также широту и долготу точки и выводит проекцию точки.

1.2.32. *Преобразование цветов.* Для представления цвета используются несколько разных форматов. Например, *RGB* — основной формат для ЖК-экранов, цифровых камер и веб-страниц — задает интенсивность красной (R), зеленой (G) и синей (B) цветовой составляющей в диапазоне от 0 до 255. В полиграфии чаще всего используется формат *CMYK*, определяющий уровень голубого (C), малинового (M), желтого (Y) и черного (K) цветов по вещественной шкале от 0,0 до 1,0. Напишите программу `RGBtoCMYK`, преобразующую значение RGB в CMYK. Программа получает в командной строке три целых числа `r`, `g` и `b` и выводит эквивалентные значения в формате CMYK. Если все компоненты RGB равны 0, то все значения CMY равны 0, а значение K равно 1; в остальных случаях используются следующие формулы:

```
w = max ( r / 255, g / 255, b / 255 )
c = (w - (  r / 255)) / w
m = (w - (  g / 255)) / w
y = (w - ( b / 255)) / w
k = 1 - w
```

1.2.33. *Дуга большого круга.* Напишите программу `GreatCircle`, которая получает четыре значения типа `double` `x1`, `y1`, `x2` и `y2` в аргументах командной строки (широта и долгота в градусах двух точек земной поверхности) и выводит расстояние дуги большого круга между этими точками. Расстояние дуги большого круга (в морских милях) задается следующей формулой:

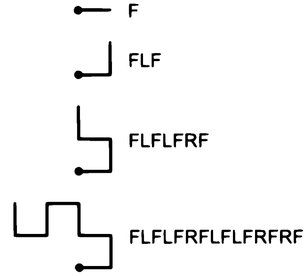
$$d = 60 \arccos(\sin(x_1) \sin(x_2) + \cos(x_1) \cos(x_2) \cos(y_1 - y_2)).$$

Обратите внимание: в формуле используются градусы, а тригонометрические функции Java работают с радианами. Используйте методы `Math.toRadians()` и `Math.toDegrees()` для преобразования между угловыми единицами. Используйте свою программу для вычисления расстояния по дуге большого круга между Парижем (48,87° N и –2,33° W) и Сан-Франциско (37,8° N и 122,4° W).

1.2.34. *Сортировка троек.* Напишите программу, которая получает в аргументах командной строки три целочисленных значения и выводит их упорядоченными по возрастанию. Используйте методы `Math.min()` и `Math.max()`.

1.2.35. *Кривая дракона.* Напишите программу, которая выводит инструкции для рисования кривых дракона разного порядка, от 0 до 5. Инструкции выглядят как

цепочки символов F, L и R, где F означает «нарисовать линию на 1 единицу вперед», L — «повернуть налево», а R — «повернуть направо». Чтобы получить кривую дракона порядка *n*, сложите полоску бумаги пополам *n* раз, а затем разверните с прямыми углами. Чтобы решить эту задачу, заметим, что кривая порядка *n* является кривой порядка *n* – 1, за которой следует инструкция L и кривая порядка *n* – 1, обходимая в обратном порядке (далее аналогичное описание строится для обратной кривой).



Кривая дракона порядков 0, 1, 2 и 3

1.3. Условные переходы и циклы

В программах, которые мы до сих пор рассматривали, каждая строка выполнялась ровно один раз в заданном порядке. Обычно программы имеют более сложную структуру, а последовательность выполнения операций и количество выполнений каждой операции могут изменяться. Для обозначения последовательности выполнения команд в программах мы будем использовать термин «*программная логика*». В этом разделе будут описаны конструкции, позволяющие изменять программную логику в зависимости от значений переменных программы. Эта возможность является важнейшей составляющей программирования.

А если конкретнее, мы рассмотрим *условные переходы* (*ветвления*) в Java, когда некоторые операции могут выполняться или не выполняться в зависимости от условий, и *циклы*, в которых некоторые операции могут выполняться многократно (также в зависимости от условий). Как будет показано в этом разделе, условные конструкции и циклы позволяют в полной мере использовать возможности компьютера и решать множество разнообразных задач, решение которых без применения компьютера попросту нереально.

Конструкции if

Во многих вычислениях для разных вариантов входных данных должны выполняться разные действия. Для выражения таких различий на языке Java может использоваться конструкция `if`:

```
if (<выражение boolean>) { <операции> }
```

В этом описании представлен *шаблон* — формальная запись, используемая для определения формата конструкций Java. Уже определенные конструкции заключаются в угловые скобки (< >); они показывают, что в указанном месте может использоваться любой экземпляр этой конструкции. В данном случае компонент <*выражение boolean*> обозначает выражение, в результате вычисления которого

будет получен результат типа `boolean` (например, выражение с оператором сравнения), а компонент *<операции>* обозначает программный блок (последовательность строк программы Java). Последняя конструкция вам уже знакома: тело `main()` является такой последовательностью. Если последовательность состоит всего из одной команды, фигурные скобки необязательны. В принципе, можно было бы привести формальные определения *<выражение boolean>* и *<операции>*, но мы не будем опускаться на такой уровень детализации. Смысл конструкции `if` достаточно очевиден: операции в блоке выполняются в том (и только в том) случае, если выражение истинно.

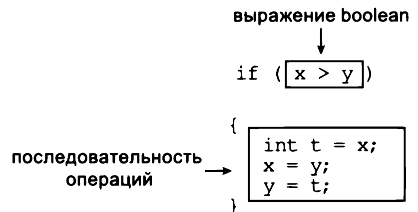
Простой пример: допустим, вы хотите вычислить абсолютное значение (модуль) переменной `x` типа `int`. Задача решается следующей командой:

```
if (x < 0) x = -x;
```

(Если говорить точнее, текущее значение `x` заменяется абсолютным значением `x`.) Рассмотрим еще один простой пример:

```
if (x > y)
{
   int t = x;
   x = y;
   y = t;
}
```

Этот код сохраняет меньшее из двух значений типа `int` в `x`, а большее — в `y`, меняя местами значения двух переменных при необходимости.



Строение конструкции `if`

В конструкцию `if` также можно добавить секцию `else`, которая выражает концепцию выполнения либо одной операции (последовательности операций), либо другой в зависимости от истинности или ложности логического условия, как в следующем шаблоне:

```
if (<выражение boolean>) <операции T>
else                     <операции F>
```

Возьмем следующий код, в котором большее из двух значений типа `int` присваивается переменной `max`:

```
if (x > y) max = x;
else       max = y;
```

Чтобы лучше понять программную логику, можно воспользоваться специальной графической формой, которая называется *блок-схемой*. Пути на блок-схеме соответствуют путям потока операций в программе. На ранней стадии развития программирования, когда программисты использовали низкоуровневые языки

и запутанную программную логику, блок-схемы были важной частью работы программиста. С современными языками блок-схемы используются разве что для пояснения базовых структурных элементов, таких как конструкция `if`[1].

Примеры блок-схем (конструкции `if`)

В следующей таблице приведены примеры использования условных переходов `if` и `if-else`. Эти примеры типичны для простых вычислений, которые могут встретиться в написанных вами программах. Условные переходы являются важнейшей частью программирования. Поскольку по своей *семантике* (смыслу, проще говоря) они близки к их аналогам в естественных языках, вы быстро привыкнете к ним.

абсолютное значение
```
if (x < 0) x = -x;
```

сохранение меньшего из двух значений в x, а большего в y
```
if (x > y)
{
  int t = x;
  x = y;
  y = t;
}
```

максимум из x и y
```
if (x > y) max = x;
else       max = y;
```

проверка ошибок для операции деления
```
if (den == 0) System.out.println("Division by zero");
else          System.out.println("Quotient = " + num/den);
```

проверка ошибок для решения квадратного уравнения
```
double discriminant = b*b - 4.0*c;
if (discriminant < 0.0)
{
   System.out.println("No real roots");
}
else
{
   System.out.println((-b + Math.sqrt(discriminant))/2.0);
   System.out.println((-b - Math.sqrt(discriminant))/2.0);
}
```

Типичные примеры использования конструкций if и if-else

[1] Блок-схемы (и подобные им) сохраняют актуальность и сейчас, однако чаще используются более высокоуровневые специализированные нотации, например на основе языка графического описания объектов UML. — *Примеч. науч. ред.*

Листинг 1.3.1. Бросок монетки

```
public class Flip
{
   public static void main(String[] args)
   {  // Моделирование броска монетки.
      if (Math.random() < 0.5) System.out.println("Heads");
      else                     System.out.println("Tails");
   }
}
```

Программа использует метод `Math.random()` *для моделирования броска монетки. При каждом запуске она выводит либо* `Heads` *(«орел»), либо* `Tails` *(«решка»). Последовательность бросков может обладать многими свойствами, которыми обладает последовательность обычных бросков, но действительно случайной такая последовательность не является.*

```
% java Flip
Heads
% java Flip
Tails
% java Flip
Tails
```

В листинге 1.3.1 приведен еще один пример использования `if-else`, на этот раз для моделирования броска монетки. Тело программы состоит из одной строки, похожей на операции из таблицы, однако оно заслуживает особого внимания, потому что поднимает интересный философский вопрос, над которым стоит задуматься: а может ли компьютерная программа генерировать действительно *случайные* значения? Нет, конечно, но программа может выдавать числа, которые обладают многими свойствами случайных чисел.

Циклы while

Многие вычисления по своей природе ориентированы на многократное повторение однотипных действия. Базовая конструкция Java для проведения таких вычислений имеет следующий формат:

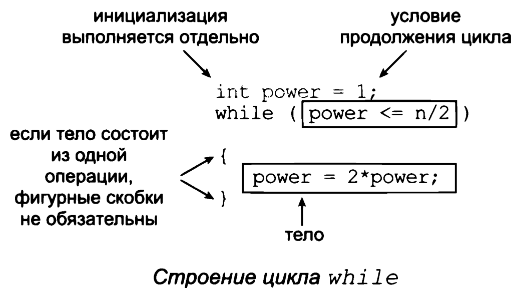
```
while (<выражение boolean>) { <операции> }
```

Конструкция `while` устроена практически так же, как конструкция `if` (единственное отличие — использование ключевого слова `while` вместо `if`), но смысл у нее совершенно другой. Фактически она приказывает компьютеру действовать по следующей схеме: если логическое условие ложно, не делать ничего; если условие истинно, выполнить последовательность операций (как и в случае с командой `if`), а затем снова проверить выражение, в случае его истинности снова выполнить последовательность операций... И *продолжать* так, пока выражение остается истинным. Блок операций (строк кода), повторяемых в цикле, называется *телом цикла.*

Как и в случае с `if`, если тело цикла состоит всего из одной операции, фигурные скобки необязательны. Команда `while` эквивалентна следующей последовательности одинаковых команд `if`[1]:

```
if (<выражение boolean>) { <операции> }
if (<выражение boolean>) { <операции> }
if (<выражение boolean>) { <операции> }
...
```

В какой-то момент код одной из строк должен что-то изменить (например, значение некоторой переменной, задействованной в выражении `boolean`). Условие становится ложным, и последовательность выполнения нарушается.

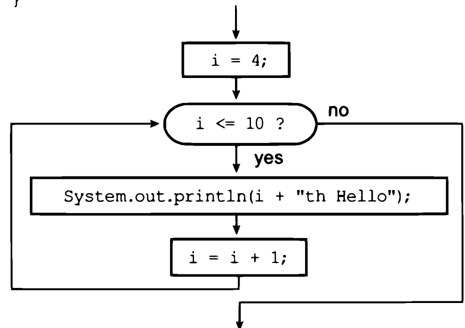


Строение цикла `while`

Одна из типичных ситуаций в программах — наличие целочисленной переменной, в которой хранится количество выполнений цикла. Переменной присваивается некоторое исходное значение, которое затем увеличивается на 1 при каждом прохождении цикла. Чтобы принять решение о продолжении, программа проверяет, не достигло ли значение переменной некоего заранее определенного максимума. Программа `TenHellos` (листинг 1.3.2) дает простой пример парадигмы, использующей команду `while`. Ключевую роль в вычислениях играет операция

```
i = i + 1;
```

```
int i = 4;
while (i <= 10)
{
   System.out.println(i + "th Hello");
   i = i + 1;
}
```



Блок-схема (цикл `while`*)*

[1] Еще удобнее было бы смоделировать цикл `while` с помощью одного ветвления `if` и одного *безусловного перехода*. Однако операции безусловного перехода обычно считаются нежелательными с точки зрения хорошего стиля *структурного* программирования. — *Примеч. науч. ред.*

Листинг 1.3.2. Первый цикл while

```
public class TenHellos
{
   public static void main(String[] args)
   {  // Вывести "Hello" 10 раз.
      System.out.println("1st Hello");
      System.out.println("2nd Hello");
      System.out.println("3rd Hello");
      int i = 4;
      while (i <= 10)
      {  // Вывод i-го приветствия
         System.out.println(i + "th Hello");
         i = i + 1;
      }
   }
}
```

В этой программе цикл `while` *используется для выполнения простой, монотонной работы – вывода результата, приведенного ниже. После третьего экземпляра выводимые строки отличаются только значением индекса в тексте, поэтому мы определяем переменную i для хранения этого индекса. После инициализации i значением 4 программа входит в цикл while, в котором значение i передается* `System.out.println()` *и увеличивается при каждом прохождении цикла. После того, как программа выведет "Hello" в 10 раз, значение i увеличивается до 11, и цикл прекращается.*

```
% java TenHellos
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

i	i <= 10	вывод
4	true	4th Hello
5	true	5th Hello
6	true	6th Hello
7	true	7th Hello
8	true	8th Hello
9	true	9th Hello
10	true	10th Hello
11	false	

Трассировка для команды java TenHellos

Если рассматривать ее как математическое уравнение, она бессмысленна, но как операция присваивания Java она имеет вполне определенный смысл: она приказывает вычислить значение `i + 1` и присвоить результат переменной `i`. Если значение `i` до выполнения этой операции было равно 4, то после него оно становится равным 5; если оно было равно 5, то становится равным 6 и т. д. В `TenHellos` установлено начальное условие, согласно которому значение `i` начинается с 4; в этом случае блок до нарушения условия будет выполнен 7 раз, когда значение `i` станет равно 11.

Для такой простой задачи применение цикла `while` вряд ли оправданно, но вскоре вы начнете решать задачи, в которых количество повторений слишком велико, чтобы его можно было организовать без циклов. Между программами с командами `while` и программами без них существуют принципиальные различия: команды `while` по-

зволяют определить любое количество повторяющихся действий, которые должны выполняться в программе. В частности, цикл `while` позволяет выполнять в коротких программах невероятно длинные последовательности действий, которые было бы немыслимо выполнить без компьютера. Впрочем, удобство не дается даром: чем эффективнее становятся ваши программы, тем труднее в них разобраться.

i	power	i <= n
0	1	true
1	2	true
2	4	true
3	8	true
4	16	true
5	32	true
6	64	true
7	128	true
8	256	true
9	512	true
10	1024	true
11	2048	true
12	4096	true
13	8192	true
14	16384	true
15	32768	true
16	65536	true
17	131072	true
18	262144	true
19	524288	true
20	1048576	true
21	2097152	true
22	4194304	true
23	8388608	true
24	16777216	true
25	33554432	true
26	67108864	true
27	134217728	true
28	268435456	true
29	536870912	true
30	1073741824	false

Трассировка для команды `java PowersOfTwo`

Программа `PowersOfTwo` (листинг 1.3.3) использует цикл `while` для вывода таблицы степеней 2. Кроме управляющей переменной цикла `i`, программа создает переменную `power` для хранения вычисляемых степеней 2. Тело цикла состоит из трех операций: одна выводит текущую степень 2, другая вычисляет следующую (умножением текущего значения на 2), а третья увеличивает управляющую переменную цикла (*счетчик цикла*).

В информатике часто встречаются ситуации, в которых полезно знать степени 2. Вы должны знать по крайней мере 10 первых значений из этой таблицы, а также примерно представлять, что 2^{10} — чуть более тысячи, 2^{20} — более миллиона, а 2^{30} — более миллиарда.

Программа `PowersOfTwo` может стать прототипом для многих полезных вычислений. Изменяя формулу, по которой рассчитывается накапливаемое значение, и правило увеличения управляющей переменной цикла, можно выводить таблицы значений множества разных функций (см. упражнение 1.3.12).

Тщательно проанализируйте поведение программ, использующих циклы, изучив трассировку программы. Например, в трассировке программы `PowersOfTwo` значение переменной и значение выражения `boolean`, управляющего циклом, выводятся в начале каждой итерации цикла. Трассировка циклов выглядит весьма однообразно, но часто оказывается очень полезной, потому что она ясно показывает, что происходит в программе.

Программа `PowersOfTwo` практически не нуждается в трассировке, потому что она выводит значения своих переменных при каждом проходе цикла. Разумеется, подобную «самотрассировку» можно включить в любую программу, добавив соответствующие команды `System.out.println()`. В современных средах разработки предусмотрены и другие мощные средства отладки, но этот проверенный метод прост и эффективен. Конечно, на первых порах вам стоит добавлять команды вывода в написанные вами циклы — это поможет вам убедиться в том, что циклы работают именно так, как вы ожидаете.

В программе `PowersOfTwo` скрыта ловушка: самое большое целое число, которое может храниться в типе данных `int` языка Java, равно $2^{31} - 1$, но программа не проверяет эту возможность. Если запустить ее командой `java PowersOfTwo 31`, последняя строка вывода может вас удивить:

```
...
1073741824
-2147483648
```

Переменная `power` становится слишком большой и принимает отрицательное значение, потому что в Java целые числа представляются именно так. Наибольшее значение `int`, которое может использоваться в программе, равно `Integer.MAX_VALUE`. Более правильная по сравнению с листингом 1.3.3 версия программы должна, используя это значение, обнаружить переполнение и вывести сообщение об ошибке, если пользователь введет слишком большой аргумент, хотя обеспечить правильную работу этой программы для всех вариантов входных данных сложнее, чем может показаться. (Похожая задача представлена в упражнении 1.3.16.)

Листинг 1.3.3. Вычисление степеней 2

```
public class PowersOfTwo
{
   public static void main(String[] args)
   {  // Вывод первых n степеней 2.
      int n = Integer.parseInt(args[0]);
      int power = 1;
      int i = 0;
      while (i <= n)
      {  // Вывод i-й степени 2.
         System.out.println(i + " " + power);
         power = 2 * power;
         i = i + 1;
      }
   }
}
```

n	пороговое значение, завершающее цикл
i	управляющая переменная цикла
power	текущая степень 2

Программа получает целое число в аргументе командной строки и выводит таблицу степеней 2 вплоть до 2^n*. При каждом проходе цикла программа увеличивает значение i и удваивает значение power. Мы приводим только три первых и три последних строки таблицы; программа выводит n+1 строк.*

```
% java PowersOfTwo 5
0 1
1 2
2 4
3 8
4 16
5 32
```

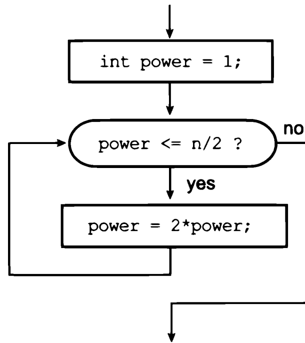
```
% java PowersOfTwo 29
0 1
1 2
2 4
...
27 134217728
28 268435456
29 536870912
```

Рассмотрим более сложный пример: допустим, вы хотите вычислить самую большую степень 2, меньшую или равную заданному положительному числу n. Если n равно 13, программа должна выводить результат 8; если n равно 1000 — результат 512; если n равно 64 — результат 64 и т. д. Необходимые вычисления легко выполняются в цикле while:

```
int power = 1;
while (power <= n/2)
   power = 2*power;
```

Возможно, вам понадобится какое-то время, чтобы удостовериться в том, что этот простой блок кода приводит к желаемому результату. Для этого можно отметить следующее.

- Значение power всегда содержит степень 2.
- Значение power никогда не превышает n.



Блок-схема для последовательности операций

```
int power = 1;
while (power <= n/2)
   power = 2*power;
```

- Значение `power` увеличивается при каждом выполнении цикла, поэтому цикл в какой-то момент должен завершиться.
- После завершения цикла `2*power` больше `n`.

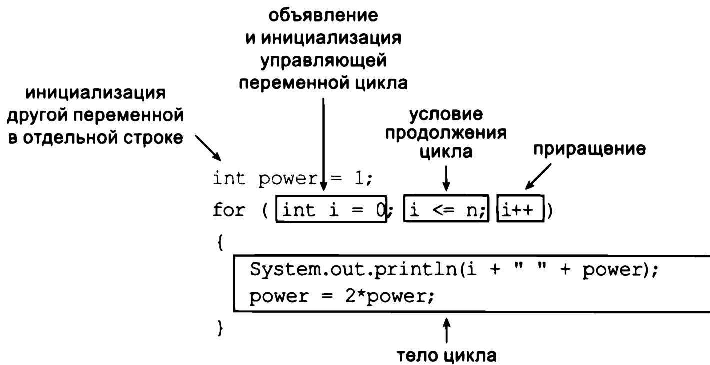
Рассуждения такого рода часто важны для понимания того, как работает цикл `while`. И хотя многие циклы, которые вы напишете, будут гораздо проще этого, постарайтесь проследить за тем, чтобы все написанные вами циклы работали так, как вы ожидали.

Логика, на которой базируются такие рассуждения, остается неизменной, сколько бы раз ни выполнялся цикл — несколько раз, как в программе `TenHellos`, несколько десятков раз, как в программе `PowersOfTwo`, или миллионы раз, как в примерах, которые будут рассмотрены ниже. Этот переход от нескольких итераций к огромным вычислениям исключительно важен. При написании циклов очень важно понять, как значения переменных изменяются на каждой итерации цикла, — и проверить это понимание, добавив команды для трассировки значений и выполнив цикл с небольшим количеством итераций. Когда это будет сделано, вы сможете с уверенностью удалить «страховку» и в полной мере использовать все возможности компьютера.

Циклы for

Как вы увидите, циклы `while` позволяют писать программы для самых разных задач. Но прежде чем переходить к новым примерам, мы рассмотрим альтернативную конструкцию Java, которая обладает еще большей гибкостью при написании программ с циклами. Этот альтернативный синтаксис принципиально не отличается от базового цикла `while`, но широко применяется на практике благодаря тому, что

программы с ним получаются более компактными и понятными, чем при использовании только циклов `while`.



Строение цикла for (для вывода степеней 2)

Синтаксис for

Многие циклы строятся по одной схеме: индексная переменная инициализируется некоторым значением, после чего в цикле `while` проверяется условие продолжения цикла с участием индексной переменной. Такие циклы можно выражать напрямую в синтаксисе Java `for`:

```
for (<инициализация>; <выражение boolean>; <приращение>)
{
   <операции>
}
```

Этот код за незначительными исключениями эквивалентен следующему:

```
<инициализация>;
while (<выражение boolean>)
{
   <операции>
   <приращение>;
}
```

Ваш компилятор Java даже может сгенерировать одинаковые результаты для двух циклов. Секции *<инициализация>* и *<приращение>* могут быть составными, но мы почти всегда используем циклы `for` для поддержки типичной идиомы программирования «инициализация+приращение». Например, следующие две строки кода эквивалентны соответствующим строкам кода в `TenHellos` (листинг 1.3.2):

```
for (int i = 4; i <= 10; i = i + 1)
   System.out.println(i + "th Hello");
```

Как правило, в программах используется чуть более компактная версия этого кода, которая рассматривается в следующем разделе.

Совмещенное присваивание

Операция изменения значения переменной настолько часто встречается в программировании, что в Java для этой цели существуют различные варианты сокращенной записи. Например, следующие четыре операции увеличивают значение переменной `i` на 1:

```
i = i+1;   i++;   ++i;   i += 1;
```

Также можно использовать запись `i--`, `--i`, `i -= 1` или `i = i-1` для уменьшения значения `i` на 1. Многие программисты используют в циклах `for` запись `i++` или `i--`, хотя подойдет и любая другая сокращенная форма. Конструкции `++` и `--` обычно используются для целых чисел, но конструкции составного присваивания могут использоваться с любым арифметическим оператором для любого примитивного числового типа — например, `power *= 2` или `power += power` вместо `power = 2*power`. Все эти так называемые идиомы существуют исключительно для удобства записи, ничего более[1]. Они получили широкое распространение после языка программирования C в 1970-х годах и с тех пор стали фактически стандартными. Они прошли проверку временем, потому что с ними создаются компактные, элегантные и понятные программы. Научившись писать (и читать) программы, в которых эти конструкции используются, вы сможете перенести свои навыки в область программирования на множестве современных языков, не только на языке Java.

Область видимости

Областью видимости переменной называется часть программы, которая может обращаться к этой переменной по имени. Как правило, область видимости переменной состоит из строк кода, следующих за объявлением переменной в том же блоке, в котором находится объявление. В этом контексте код заголовка цикла `for` считается находящимся в одном блоке с телом цикла `for`. Следовательно, циклы `while` и `for` не полностью эквивалентны: в типичном цикле `for` переменная цикла объявляется в самом цикле и поэтому недоступна для использования в дальнейшем коде, а в аналогичном цикле `while` она доступна. Из-за этого различия в программах нередко используется цикл `while` вместо цикла `for`.

Выбор между разными формулировками одной операции нередко определяется личными предпочтениями программиста — по аналогии с тем, как писатель выбирает между несколькими синонимами или между активным и пассивным залогом при составлении предложения. Непреложных правил относительно того, как писать программы, не существует, как и непреложных правил по написанию прозы. Постарайтесь найти стиль, который подходит лично вам, решает необходимую задачу и будет понятен другим.

---

[1] Конкретно в Java совмещенное присваивание, в отличие от присваивания результата выражения, сопровождается автоматическим приведением типа. Также, в зависимости от реализации компилятора, может генерироваться более эффективный исполняемый код. — *Примеч. науч. ред.*

В следующей таблице приведены фрагменты кода с типичными примерами циклов, используемых в коде Java. Некоторые из них относятся к коду, который вы уже видели; в других представлен достаточно прямолинейный код. Чтобы закрепить ваше понимание циклов в языке Java, напишите циклы для похожих вычислений вашего собственного изобретения или выполните первые упражнения в конце этого раздела. Ничто не заменит практического опыта выполнения написанного вами кода, поэтому вы должны развивать собственное понимание того, как писать на Java с использованием циклов.

вычисление наибольшей степени 2, меньшей или равной n	`int power = 1;`
	`while (power <= n/2)`
	`   power = 2*power;`
	`System.out.println(power);`

вычисление суммы членов конечной арифметической прогрессии (1 + 2 + ... + n)	`int sum = 0;`
	`for (int i = 1; i <= n; i++)`
	`   sum += i;`
	`System.out.println(sum);`

вычисление факториала (1 × 2 × ... × n)	`int product = 1;`
	`for (int i = 1; i <= n; i++)`
	`   product *= i;`
	`System.out.println(product);`

вывод таблицы значений функции	`for (int i = 0; i <= n; i++)` `   System.out.println(i + " " + 2*Math.PI*i/n);`

вычисление размеров делений на линейке (см. листинг 1.2.1)	`String ruler = "1";`
	`for (int i = 2; i <= n; i++)`
	`   ruler = ruler + " " + i + " " + ruler;`
	`System.out.println(ruler);`

Типичные примеры использования циклов for и while

Вложенные конструкции

Операции `if`, `while` и `for` обладают таким же статусом, как операции присваивания или любые другие в Java; иначе говоря, мы можем использовать их повсюду, где должна использоваться операция. В частности, одна или несколько таких операций могут использоваться в теле другой операции; так формируются *составные операции*. В первом примере `DivisorPattern` (листинг 1.3.4) используется цикл `for`, тело которого состоит из другого цикла `for` (содержащего команду `if-else`) и команды вывода. Он выводит узор, `i`-я строка которого содержит звездочку в каждой позиции, соответствующей делителям `i` (для всех столбцов).

Для выделения вложенных конструкций в программном коде используются отступы. Мы будем называть цикл с переменной `i` *внешним циклом*, а цикл с переменной `j` — *внутренним*. Внутренний цикл заново выполняет все свои итерации при каждой итерации внешнего цикла. Как обычно, чтобы понять новую программную конструкцию, лучше всего изучить трассировку.

Листинг 1.3.4. Первая программа с вложенным циклом

```
public class DivisorPattern
{
   public static void main(String[] args)
   {  // Вывод визуального представления делителей.
      int n = Integer.parseInt(args[0]);
      for (int i = 1; i <= n; i++)
      {  // Вывод i-й строки.
         for (int j = 1; j <= n; j++)
         {  // Вывод j-го элемента в i-й строке.
            if ((i % j == 0) || (j % i == 0))
               System.out.print("* ");
            else
               System.out.print("  ");
         }
         System.out.println(i);
      }
   }
}
```

n	количество строк и столбцов
i	индекс строки
j	индекс столбца

Программа получает целочисленный аргумент n в командной строке и использует вложенные циклы for для вывода таблицы n × n, в которой на пересечении строки i и столбца j стоит звездочка, если i делится на j или j делится на i. Переменные цикла i и j управляют ходом вычислений.

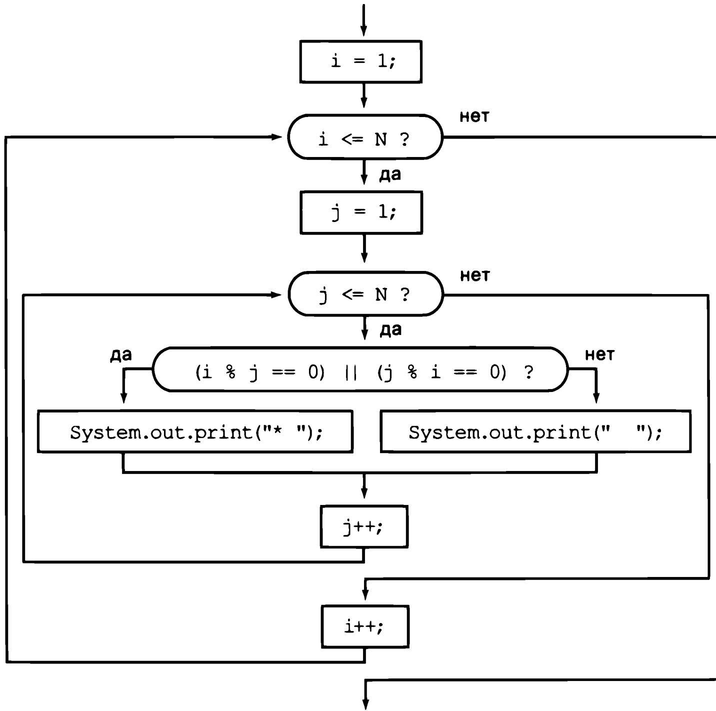
```
% java DivisorPattern 3
* * * 1
* *   2
*   * 3
% java DivisorPattern 12
* * * * * * * * * * * * 1
* *   *   *   *   *   * 2
*   *     *     *     * 3
* *   *       *       * 4
*       *         *     5
* * *     *           * 6
*           *           7
* *   *       *         8
*   *           *       9
* *     *         *     10
*                   *   11
* * * *   *           * 12
```

i	j	i % j	j % i	вывод
1	1	0	0	*
1	2	1	0	*
1	3	1	0	*
				1
2	1	0	1	*
2	2	0	0	*
2	3	2	1	
				2
3	1	0	1	*
3	2	1	2	
3	3	0	0	*
				3

Трассировка для команды java DivisorPattern 3

В программе `DivisorPattern` используется достаточно сложная программная логика, как видно из блок-схемы. Такие блок-схемы показывают, как важно использовать при программировании небольшое количество простых управляющих структур. Благодаря вложению из циклов и ветвлений можно строить простые и понятные программы, даже если они базируются на сложной программной логи-

ке. Очень многие полезные вычисления реализуются с одним или двумя уровнями вложения. Например, во многих программах в книге используется та же общая структура, что и `DivisorPattern`.



Блок-схема программы DivisorPattern

Второй пример вложения: рассмотрим следующий фрагмент программы, который может использоваться в финансовой программе для вычисления ставки подоходного налога:

```
if      (income <      0) rate = 0.00;
else if (income <   8925) rate = 0.10;
else if (income <  36250) rate = 0.15;
else if (income <  87850) rate = 0.23;
else if (income < 183250) rate = 0.28;
else if (income < 398350) rate = 0.33;
else if (income < 400000) rate = 0.35;
else                      rate = 0.396;
```

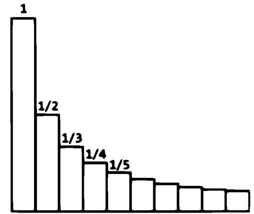
В этом примере несколько вложенных команд `if` обеспечивают проверку нескольких взаимоисключающих условий. Мы будем часто использовать эту конструкцию в книге. В других случаях при вложении команд `if` следует использовать фигурные скобки, чтобы избежать неоднозначной интерпретации. Эта проблема (вместе с другими примерами) рассматривается в разделах «Вопросы и ответы» и «Упражнения».

Примеры

Применение циклов в программировании открывает перед вами целый мир новых возможностей. Чтобы убедить вас в этом, мы рассмотрим ряд примеров. Во всех этих примерах используются типы данных, которые рассматривались в разделе 1.2, но можете не сомневаться: те же механизмы работают в любом вычислительном приложении. Все примеры были тщательно проработаны, и их изучение подготовит вас к написанию собственных программ с циклами.

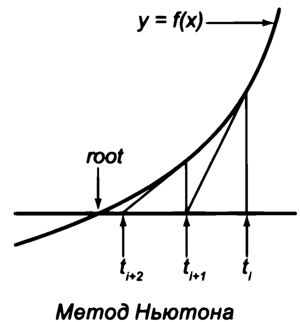
Примеры этого раздела основаны на вычислениях с числовыми данными. В некоторых примерах упоминаются задачи, которыми занимались математики и ученые на протяжении нескольких веков. И хотя компьютеры существуют всего лишь около 70 лет, многие вычислительные методы, используемые в примерах, основаны на богатой математической традиции, уходящей корнями в Античность.

Сумма конечного ряда



Подход, использованный в программе `PowersOfTwo`, часто будет встречаться в ваших программах: использование двух переменных, одна из которых управляет циклом, а в другой накапливается результат вычислений. В программе `HarmonicNumber` (листинг 1.3.5) этот же подход используется для вычисления суммы $H_n = 1 + 1/2 + 1/3 + ... + 1/n$. Эти числа, называемые *гармоническими*, часто встречаются в дискретной математике. Гармонические числа также являются дискретным аналогом логарифма и аппроксимируют площадь под кривой $y = 1/x$. Программа из листинга 1.3.5 может послужить моделью для вычисления значений других конечных сумм (см. упражнение 1.3.18).

Вычисление квадратного корня



Метод Ньютона

Как реализованы функции библиотеки Java `Math`, такие как `Math.sqrt()`? В программе `Sqrt` (листинг 1.3.6) продемонстрирован один из приемов. Для вычисления квадратного корня из положительного числа используются итерационные вычисления, известные еще в Древнем Вавилоне более 4000 лет назад. Также пример является частным случаем вычислительного метода, который был разработан в XVII веке Исааком Ньютоном и Джозефом Рафсоном и известен под названием *метода Ньютона*. Если заданная функция $f(x)$ соответствует определенным критериям, то метод Ньютона позволяет эффективно найти ее корни (значения x, для которых функция равна 0). Поиск начинается с исходной точки приближения t_0. Имея приближение t_i, для вычисления следующего приближения проводим касательную к кривой $y = f(x)$ в точке $(t_i, f(t_i))$ и в качестве t_{i+1} выбираем координату x точки пересечения касательной с осью x. Повторение этого процесса приближает найденную точку к точному значению корня.

Листинг 1.3.5. Гармонические числа

```
public class HarmonicNumber
{
   public static void main(String[] args)
   {  // Вычисление n-го гармонического числа.
      int n = Integer.parseInt(args[0]);
      double sum = 0.0;
      for (int i = 1; i <= n; i++)
      {  // Прибавление i-го слагаемого к сумме.
         sum += 1.0/i;
      }
      System.out.println(sum);
   }
}
```

n	количество слагаемых в сумме
i	управляющая переменная цикла
sum	накапливаемая сумма

Программа получает целочисленный аргумент n в командной строке и вычисляет значение n-го гармонического числа. Из математического анализа известно, что это значение равно приблизительно ln(n) + 0,57721 для больших n. Например, ln(1 000 000) + 0,57721 ≈ ≈ 14,39272.

```
% java HarmonicNumber 2
1.5
% java HarmonicNumber 10
2.9289682539682538
```

```
% java HarmonicNumber 1000
7.485470860550343
% java HarmonicNumber 1000000
14.392726722864989
```

Вычисление квадратного корня положительного числа c эквивалентно поиску положительного корня функции $f(x) = x^2 - c$. В этом частном случае метод Ньютона приводит к процессу, реализованному в `Sqrt` (см. упражнение 1.3.19). Вычисления начинаются с приближения `t = c`. Если `t` равно `c/t`, то `t` равно квадратному корню из `c`, и вычисления завершаются. В противном случае текущее приближение уточняется — `t` заменяется средним арифметическим `t` и `c / t`. При использовании метода Ньютона значение квадратного корня из 2 вычисляется с точностью до 15 знаков всего за 5 итераций[1].

Метод Ньютона играет важную роль в научных вычислениях, потому что тот же итерационный метод позволяет эффективно находить корни широкого класса функций, в том числе и тех, для которых аналитические решения неизвестны (и для которых библиотека Java `Math` не поможет). Сейчас мы воспринимаем возможность нахождения любых значений математических функций как нечто само собой разумеющееся, но до появления компьютеров ученым и инженерам приходилось пользоваться таблицами или вычислять значения вручную. Вычислительные методы, разработанные для проведения ручных вычислений, должны были быть очень эффективными; неудивительно, что многие из этих методов эффективно работают и на компьютерах. Метод Ньютона — классический пример такого рода. Другой

[1] Метод Ньютона относится к быстросходящимся: каждая итерация удваивает количество точных цифр текущего приближенного результата. — *Примеч. науч. ред.*

Листинг 1.3.6. Метод Ньютона

```
public class Sqrt
{
   public static void main(String[] args)
   {
      double c = Double.parseDouble(args[0]);
      double EPSILON = 1e-15;
      double t = c;
      while (Math.abs(t - c/t) > EPSILON * t)
      {  // Заменить t средним арифметическим t и c/t.
         t = (c/t + t) / 2.0;
      }
      System.out.println(t);
   }
}
```

c	аргумент
EPSILON	погрешность
t	приближенное значение квадратного корня из c

Программа получает положительное число с плавающей точкой c *в командной строке и вычисляет квадратный корень из* c *с точностью до 15 знаков по методу Ньютона (см. в тексте).*

```
% java Sqrt 2.0
1.414213562373095
% java Sqrt 2544545
1595.1630010754388
```

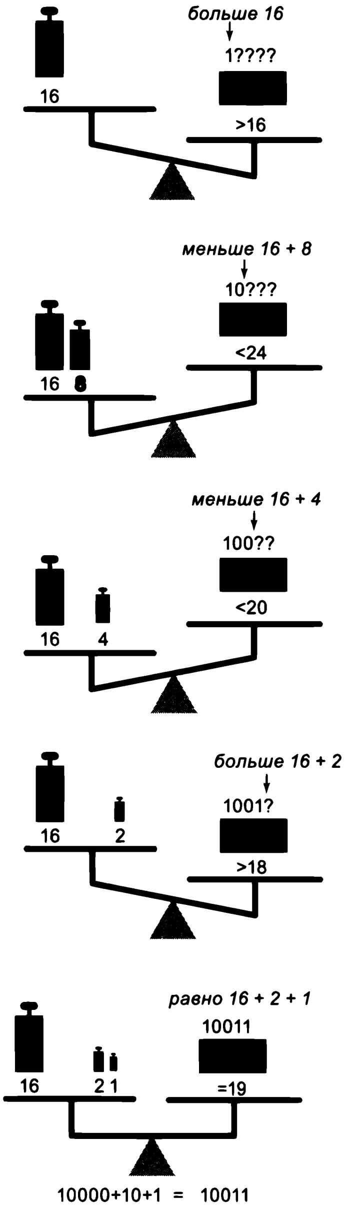
Итерация	t	c/t
	2.0000000000000000	1.0
1	1.5000000000000000	1.3333333333333333
2	1.4166666666666665	1.4117647058823530
3	1.4142156862745097	1.4142114384748700
4	1.4142135623746899	1.4142135623715002
5	1.4142135623730950	1.4142135623730951

Трассировка для команды java Sqrt 2.0

полезный метод вычислений математических функций основан на использовании разложения в ряд Тейлора (см. упражнения 1.3.37 и 1.3.38).

Преобразование чисел

Программа Binary (листинг 1.3.7) выводит двоичное (по основанию 2) представление десятичного числа, переданного в аргументе командной строки. Работа программы основана на разложении числа на сумму степеней 2. Например, двоичное представление 19 равно 10011; фактически это то же самое, что 19 = 16 + 2 + 1. Чтобы вычислить двоичное представление n, мы рассматриваем степени 2, меньшие или равные n, в порядке убывания, чтобы определить, какие из них присутствуют в двоичном разложении (и, следовательно, соответствуют единичным разрядам в двоичном представлении). Процесс в точности соответствует взвешиванию предмета на весах с гирями, вес которых определяется степенями 2. Сначала находится самая большая гиря, не превышающая веса объекта. Затем, рассматривая гири по убыванию веса, мы ставим на весы каждую гирю и смотрим, не превысил ли



Преобразование в двоичную систему методом весов

суммарный вес гирь вес объекта. В случае превышения гиря снимается; в противном случае проверяется следующая гиря. Каждая гиря соответствует биту в двоичном представлении веса объекта; гири, остающиеся на весах, соответствуют 1 в двоичном представлении веса объекта, а снятые гири соответствуют 0 в двоичном представлении веса.

В программе `Binary` переменная `power` соответствует текущей проверяемой гире, а переменная `n` отвечает за избыточную (неизвестную) часть веса предмета (чтобы смоделировать груз, оставшийся на весах, мы просто вычитаем его вес из `n`). Значение `power` перебирает степени 2 по убыванию. Если значение больше `n`, программа выводит `0`; в противном случае она выводит `1` и вычитает `power` из `n`. Как обычно, трассировка (значения `n`, `power`, `n < power` и выводимого разряда для каждой итерации цикла) сильно поможет в понимании логики программы. Читая выходные данные сверху вниз в крайнем правом столбце вывода, мы получаем результат `10011` — двоичное представление числа 19.

Тема преобразования данных между системами счисления довольно часто встречается в программировании. Она подчеркивает различия между абстракцией (целое число, представляющее количество часов в сутках) и представлением этой абстракции (24 или 11000). Своего рода ирония состоит в том, что сам компьютер использует двоичное представление чисел.

Имитационное моделирование

Следующий пример отличается от тех, которые рассматривались ранее, но он характерен для распространенной ситуации с моделированием на компьютере явлений, происходящих в реальном мире, для принятия обоснованных решений. Конкретный пример, который мы рассмотрим сейчас, относится к хорошо изученному классу задач о *разорении игрока*. Предположим, игрок делает серию ставок в $1, начиная с некоторой заданной исходной суммы. Рано или поздно игрок все

Листинг 1.3.7. Преобразование в двоичную форму

```
public class Binary
{
   public static void main(String[] args)
   {  // Вывод двоичного представления числа n.
      int n = Integer.parseInt(args[0]);
      int power = 1;
      while (power <= n/2)
         power = 2*power;
      // Теперь power содержит наибольшую степень 2 <= n.
      while (power > 0)
      {  // Вывод степеней 2 по убыванию.
         if (n < power) { System.out.print(0);             }
         else           { System.out.print(1); n -= power; }
         power /= 2;
      }
      System.out.println();
   }
}
```

n	преобразуемое целое число
power	текущая степень 2

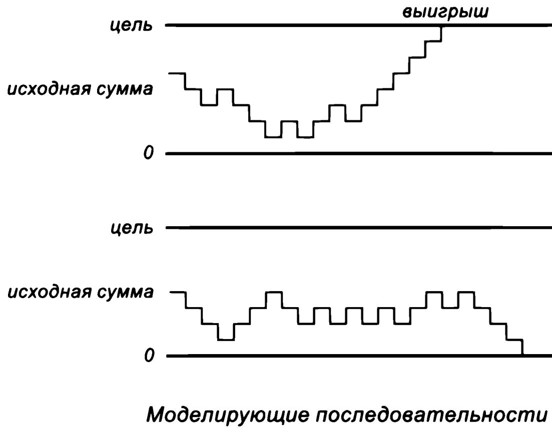
Программа получает положительное целое число n в аргументе командной строки и выводит двоичное представление n методом проверки степеней 2 в порядке убывания (см. текст).

```
% java Binary 19
10011
% java Binary 100000000
101111101011110000100000000
```

равно разорится, но при задании других игровых ограничений возникают различные вопросы. Предположим, игрок заранее решил покинуть игру после достижения определенной цели. Какова вероятность того, что игрок выиграет? Сколько ставок ему может понадобиться для выигрыша или проигрыша? Какая максимальная сумма окажется на руках у игрока в ходе игры?

n	двоичное представление	power	power > 0	двоичное представление	n < power	вывод
19	10011	16	true	10000	false	1
3	0011	8	true	1000	true	0
3	011	4	true	100	true	0
3	01	2	true	10	false	1
1	1	1	true	1	false	1
0		0	false			

Трассировка цикла с исключением степеней 2 для команды java Binary 19



Моделирующие последовательности

Программа Gambler (листинг 1.3.8) представляет собой модель для получения ответов на такие вопросы. Она проводит серию «опытов» (розыгрышей), используя метод `Math.random()` для моделирования последовательности ставок, которая продолжается либо до разорения игрока, либо до достижения им цели. При этом программа следит за тем, сколько раз игрок достиг цели и сколько ставок было сделано. После проведения заданного количества испытаний программа вычисляет средние результаты и выводит их. Попробуйте запустить эту программу с разными значениями аргументов командной строки — не для того, чтобы спланировать следующий поход в казино, а для того, чтобы вам было проще думать над следующими вопросами: насколько точно модель отражает то, что происходит в реальном мире? Сколько испытаний потребовалось для получения точного ответа? Какие вычислительные ограничения действуют для такого моделирования? Подобные модели широко применяются в экономике, науке и технике, и вопросы такого рода играют важную роль при любом моделировании.

Программа `Gambler` проверяет классические результаты, полученные методами теории вероятностей, которая утверждает, что *вероятность успеха определяется делением исходной суммы на целевую*, а *ожидаемое количество ставок равно произведению исходной суммы и ожидаемой прибыли* (разности между целевой суммой и исходной суммой). Например, если вы поедете в Монте-Карло, чтобы превратить $500 в $2500, вероятность успеха не так уж мала (20%), но вам придется сделать около миллиона ставок по $1! А если вы попытаетесь превратить $1 в $1000, вероятность успеха равна всего 0,1%, и к этому успеху (а скорее к разорению) можно прийти приблизительно за 999 ставок.

Моделирование и анализ идут рука об руку, проверяя результаты друг друга. На практике ценность имитационного моделирования заключается в том, что оно может дать ответы на вопросы, которые слишком трудно разрешить аналитическими методами. Представьте, что игрок понимает, что сделать миллион ставок все равно не удастся, решает заранее установить верхний предел для количества ставок. На какой заработок он может рассчитывать в этом случае? Чтобы получить

Листинг 1.3.8. Моделирование задачи о разорении игрока

```
public class Gambler
{
   public static void main(String[] args)
   {  // Проведение trials экспериментов, которые начинаются
      // с исходной суммы $stake и завершаются результатом $0 или $goal.
      int stake  = Integer.parseInt(args[0]);
      int goal   = Integer.parseInt(args[1]);
      int trials = Integer.parseInt(args[2]);
      int bets = 0;
      int wins = 0;
      for (int t = 0; t < trials; t++)
      {  // Проведение одного опыта.
         int cash = stake;
         while (cash > 0 && cash < goal)
         {  // Моделирование одной ставки
            bets++;
            if (Math.random() < 0.5) cash++;
            else                     cash--;
         }  // Сумма достигает 0 (разорение) или $goal (победа).
         if (cash == goal) wins++;
      }
      System.out.println(100*wins/trials + "% wins");
      System.out.println("Avg # bets: " + bets/trials);
   }
}
```

stake	исходная сумма
goal	целевая сумма
trials	количество опытов
bets	количество ставок
wins	количество побед
cash	текущая сумма

Программа получает в аргументах командной строки три целых числа: исходную сумму, цель и количество испытаний. Внутренний цикл while моделирует игрока с исходной суммой $stake, который совершает серию ставок по $1 и продолжает игру, пока не разорится или не достигнет $goal. Время выполнения этой программы пропорционально произведению количества испытаний и среднего количества ставок. Например, при третьем из приведенных ниже запусков программа генерирует около 100 миллионов случайных чисел.

```
% java Gambler 10 20 1000
50% wins
Avg # bets: 100
% java Gambler 10 20 1000
51% wins
Avg # bets: 98
```

```
% java Gambler 50 250 100
19% wins
Avg # bets: 11050
% java Gambler 500 2500 100
21% wins
Avg # bets: 998071
```

ответ на этот вопрос, достаточно внести простое изменение в программу 1.3.8 (см. упражнение 1.3.26), но решить такую задачу иными математическими методами намного сложнее.

Факторизация

Простым числом называется целое число, большее 1 и имеющее только 2 натуральных делителя: единицу и само себя. *Разложение на простые множители целого числа*, или *факторизация*, представляет собой мультимножество про-

factor	n	*Вывод*
2	3757208	2 2 2
3	469651	
4	469651	
5	469651	
6	469651	
7	469651	7
8	67093	
9	67093	
10	67093	
11	67093	
12	67093	
13	67093	13 13
14	397	
15	397	
16	397	
17	397	
18	397	
19	397	
20	397	
		397

Трассировка для команды
java Factors 3757208

стых чисел, произведением которых является целое число. Например, 3 757 208 = 2 × 2 × 2 × × 7 × 13 × 13 × 397. Программа `Factors` (листинг 1.3.9) вычисляет разложение на простые множители для любого заданного положительного целого числа. В отличие от многих других программ, рассмотренных нами (которые можно было воспроизвести за несколько минут с калькулятором или даже с карандашом и бумагой), провести такие вычисления без компьютера нереально. Как разложить на простые множители число вида 287994837222311? Возможно, множитель 17 вы найдете быстро, но даже с калькулятором потребуется немало времени для нахождения множителя 1739347.

Несмотря на компактность программы `Factors`, не столь очевидно, что она возвращает желаемый результат для любого целого числа. Как обычно, трассировка со значениями переменных в начале каждой итерации внешнего цикла `for` помогает понять логику вычислений. Если значение `n` равно 3757208, внутренний цикл `while` выполняется три раза для значения `factor`, равного 2, и исключает три множителя 2; затем нуль раз для `factor`, равного 3, 4, 5 и 6, так как 469651 ни на одно из этих чисел не делится, и т. д. Трассировка программы для нескольких примеров входных данных раскрывает принцип ее действия. Чтобы убедиться в том, что программа ведет себя как задумано для всех вариантов ввода, следует понять, что делает каждый из циклов. Цикл `while` выводит и исключает из `n` все множители `factor`; но логика программы становится по-настоящему понятной только после осознания следующего факта: в начале каждой итерации цикла `for` `n` не имеет множителей в диапазоне от `2` до `factor-1`. Другими словами, если значение `factor` не является простым числом, оно не может быть множителем `n`; если же `factor` — простое число, цикл `while` выполняет свою работу. А если мы знаем, что `n` не имеет множителей, меньших или равных `factor`, это означает, что у него нет множителей, больших `n/factor`, поэтому если `factor` больше `n/factor`, то дальнейший поиск не нужен.

В наивной реализации можно было бы просто использовать для завершения цикла `for` условие `(factor < n)`. Но даже с учетом запредельных скоростей современных компьютеров такое решение серьезно отразится на размере чисел, которые можно разложить на множители. Упражнение 1.3.28 предлагает вам поэкспериментировать с программой, чтобы оценить эффективность этого простого изменения. На компьютере, способном выполнять миллиарды операций в секунду, числа порядка 10^9

Листинг 1.3.9. Разложение целых чисел на простые множители

```
public class Factors
{
   public static void main(String[] args)
   {  // Вывод разложения n на простые множители.
      long n = Long.parseLong(args[0]);
      for (long factor = 2; factor <= n/factor; factor++)
      {  // Проверка потенциального множителя.
         while (n % factor == 0)
         {  // Исключение и вывод множителя factor.
            n /= factor;
            System.out.print(factor + " ");
         }  // Любой множитель n должен быть больше factor.
      }
      if (n > 1) System.out.print(n);
      System.out.println();
   }
}
```

n	часть, не разложенная на множители
factor	потенциальный множитель

Программа получает положительное целое число n в аргументе командной строки и выводит его разложение на простые множители. Код простой, но его правильность не столь очевидна (см. текст).

```
% java Factors 3757208
2 2 2 7 13 13 397
```

```
% java Factors 287994837222311
17 1739347 9739789
```

раскладываются на простые множители за несколько секунд; с условием (`factor <= n/factor`) за сравнимый промежуток времени раскладываются числа порядка 10^{18}. Циклы открывают возможность решения сложных задач, но они также открывают возможность писать программы простые, но медленные, поэтому вы должны постоянно помнить об эффективности.

В современных криптографических приложениях часто возникают ситуации, требующие разложения на простые множители действительно огромных чисел (из сотен и тысяч цифр). Такие вычисления могут оказаться непреодолимо сложными даже *с использованием* компьютера.

Другие условные и циклические конструкции

Чтобы наше описание языка Java было более полным, мы рассмотрим еще четыре конструкции управления программной логикой. Это вовсе не означает, что эти конструкции нужно использовать во всех написанных вами программах — они встречаются намного реже, чем `if`, `while` и `for`. И конечно, вам вообще можно не беспокоиться об этих конструкциях, пока вы не освоите `if`, `while` и `for`. Какие-то из них могут встретиться вам в учебниках или в Интернете, но многие программисты ими не пользуются. В книге вы почти не встретите их вне этого раздела.

Операция break

В некоторых ситуациях бывает нужно немедленно завершить цикл, не давая ему отработать до завершения. В языке Java для этого существует операция `break`. Например, следующий фрагмент позволяет эффективно проверить, является ли заданное целое число `n > 1` простым:

```
int factor;
for (factor = 2; factor <= n/factor; factor++)
   if (n % factor == 0) break;
if (factor > n/factor)
   System.out.println(n + " is prime");
```

Выход из этого цикла возможен по двум причинам: либо из-за выполнения команды `break` (`n` делится на `factor`, а значит, не является простым числом), либо из-за нарушения условия продолжения цикла (не найден множитель с `factor <= n/factor`, на который делится `n`, из чего следует, что `n` — простое число.) Обратите внимание: переменная `factor` должна объявляться за пределами цикла `for`, а не в секции инициализации, чтобы ее область видимости не ограничивалась циклом.

Операция continue

Язык Java также позволяет сразу перейти к следующей итерации цикла. При выполнении операции `continue` в теле цикла `for` управление сразу передается на операцию приращения для следующей итерации.

Операция switch

Команды `if` и `if-else` предоставляют всего одну или две альтернативы в управлении программной логикой. Иногда сама суть вычислений предполагает существование нескольких взаимоисключающих альтернатив. В такой ситуации можно использовать последовательные, или вложенные, конструкции `if-else` (как в примере с вычислением ставки налога, приведенном ранее в этом разделе), но команда `switch` предоставляет более прямолинейное решение. Давайте сразу рассмотрим типичный пример. Вместо того чтобы выводить значение переменной `day` типа `int` в программе, работающей с днями недели (как в решении упражнения 1.2.29), можно воспользоваться командой `switch`:

```
switch (day)
{
   case 0: System.out.println("Sun"); break;
   case 1: System.out.println("Mon"); break;
   case 2: System.out.println("Tue"); break;
   case 3: System.out.println("Wed"); break;
   case 4: System.out.println("Thu"); break;
   case 5: System.out.println("Fri"); break;
   case 6: System.out.println("Sat"); break;
}
```

Если вы пишете программу, в которой вроде бы нужно выполнить длинную серию однообразных конструкций `if`, попробуйте свериться с сайтом книги и использовать `switch` или воспользуйтесь альтернативным решением, описанным в разделе 1.4.

Цикл do–while[1]

Также существует другой вариант записи циклов с использованием шаблона

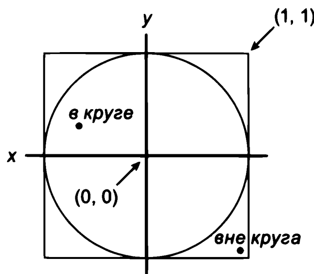
```
do { <операции> } while (<выражение boolean>);
```

Эта конструкция почти эквивалентна циклу

```
while (<выражение boolean>) { <операции> }
```

если не считать того, что первая проверка условия опускается. Если условие выполняется в начале цикла, ничего не изменяется. В какой ситуации цикл `do-while` мог бы пригодиться? Возьмем задачу генерирования точек, равномерно распределенных в единичном круге. Метод `Math.random()` позволяет независимо генерировать координаты x и y, распределенные в квадрате 2×2 с центром в начале координат. Большинство точек попадает в единичный круг, поэтому нужно просто отбросить точки, выходящие за его границы. Хотя бы одна точка должна генерироваться всегда, поэтому цикл `do-while` идеально подходит для данного случая. Следующий код присваивает переменным `x` и `y` такие значения, чтобы точка `(x, y)` была случайным образом распределена в единичном круге:

```
do
{  // x и y размещаются случайным образом в интервале (-1, 1).
   x = 2.0*Math.random() - 1.0;
   y = 2.0*Math.random() - 1.0;
} while (x*x + y*y > 1.0);
```



Так как площадь диска равна π, а площадь описанного вокруг него квадрата равна 4, ожидаемое количество итераций равно $4/\pi$ (приблизительно 1,27).

[1] Непопулярность этой конструкции еще в языке C объясняется, возможно, также и тем, что ошибочно вставленная перед `while` точка с запятой превращала цикл в бесконечный, причем компилятор эту ошибку не обнаруживал. — *Примеч. науч. ред.*

Бесконечные циклы

Прежде чем писать программу с циклом, необходимо хорошенько задуматься над одним вопросом: что произойдет, если условие продолжения цикла `while` всегда будет истинным? С теми командами, которые рассматривались до настоящего момента, возможна одна из двух неприятных ситуаций. Вы должны уметь справиться с обеими.

Во-первых, предположим, что в таком цикле вызывается `System.out.println()`. Например, если бы в программе `TenHellos` условие продолжения цикла имело вид `(i > 3)` вместо `(i <= 10)`, оно всегда было бы истинным. Что произойдет? Результат вывода бесконечного количества строк в терминальном окне зависит от особенностей среды выполнения. Если в вашей системе вывод производится на печать, у вас может кончиться бумага или вам придется отключить принтер, а в терминальном окне будет необходимо прервать операцию вывода. Прежде чем запускать программы с циклами, убедитесь в том, что вы умеете прерывать бесконечные циклы с вызовами `System.out.println()`, и проверьте свою стратегию — внесите указанное выше изменение в `TenHellos` и попробуйте прервать программу. В большинстве систем текущая программа прерывается комбинацией клавиш Ctrl+C.

```
public class BadHellos
...
int i = 4;
while (i > 3)
{
   System.out.println
      (i + "th Hello");
   i = i + 1;
}
...

% java BadHellos
1st Hello
2nd Hello
3rd Hello
5th Hello
6th Hello
7th Hello
...
```

Бесконечный цикл

Во-вторых, может не происходить вообще *ничего*. Если бесконечный цикл в программе не производит никакого вывода, программа «зациклится» и никаких результатов вы не увидите. Если вы окажетесь в такой ситуации, проверьте циклы и убедитесь в том, что условие выхода из цикла всегда выполняется, но проблему не всегда так легко обнаружить. Один из способов обнаружить такую ошибку — вставить вызовы `System.out.println()` для вывода данных трассировки. Если вызовы попадают в бесконечный цикл, эта стратегия сводит проблему к случаю из предыдущего абзаца, но результаты могут подсказать, что делать. А может быть, вы не знаете, является цикл бесконечным или просто очень длинным (или это несущественно). Например, даже программа `BadHellos` может завершиться через какое-то время после вывода более 1 миллиарда записей из-за целочисленного переполнения. Но если запустить программу 1.3.8 с аргументами `java Gambler 100000 200000 100`, дожидаться ответа придется слишком долго. Вскоре вы начнете учитывать это обстоятельство и оценивать примерное время выполнения ваших программ.

Почему Java не обнаруживает бесконечные циклы и не предупреждает нас о них? Как ни удивительно, в общем случае это невозможно. Этот неочевидный факт является одним из фундаментальных результатов компьютерной теории.

Выводы

В следующей таблице перечислены программы, которые рассматривались в этом разделе. Они характерны для задач, которые решаются короткими программами с конструкциями `if`, `while` и `for`, работающими со встроенными типами данных. Такие программы помогают освоить базовые конструкции управления логикой выполнения в языке Java.

Программа	*Описание*
`Flip`	моделирование броска монетки
`TenHellos`	первая программа с циклом
`PowersOfTwo`	вычисление и вывод таблицы значений
`DivisorPattern`	первая программа с вложенным циклом
`Harmonic`	вычисление суммы конечного ряда
`Sqrt`	классический итерационный алгоритм
`Binary`	преобразование чисел в другую систему счисления
`Gambler`	имитационное моделирование с использованием вложенных циклов
`Factors`	цикл while в цикле for

Сводка программ этого раздела

Чтобы научиться пользоваться условными и циклическими конструкциями, вы должны практиковаться в написании и отладке программ с командами `if`, `while` и `for`. Упражнения в конце этого раздела помогут вам начать практику. Для каждого упражнения вы напишете программу на языке Java, запустите и протестируете ее. Все программисты знают, что программы редко работают идеально с первого запуска, поэтому вы должны понимать свою программу и представлять, что она должна делать. На первых порах используйте трассировку для проверки ваших ожиданий. Со временем у вас появится опыт, и вы начнете думать о том, какие результаты выведет трассировка, еще при написании цикла. Попробуйте ответить на несколько вопросов: какими будут значения переменных после первой итерации цикла? После второй? Последней? Может ли программа каким-то образом зациклиться?

Освоив циклические и условные конструкции, вы сделаете огромный шаг вперед: команды `if`, `while` и `for` означают переход от простых и прямолинейных программ к логике произвольной сложности. В нескольких ближайших главах мы сделаем еще несколько таких же огромных шагов, которые позволят вам обрабатывать большие объемы входных данных, а также определять и обрабатывать данные других типов, кроме обычных числовых. Конструкции `if`, `while` и `for` из этого раздела будут играть важную роль в программах, которые мы будем рассматривать, делая эти шаги.

Вопросы и ответы

В. Чем отличаются операторы = и ==?

О. Мы повторим это, чтобы напомнить: не путайте операторы = и == в логических выражениях. Выражение (x = y) присваивает значение y переменной x, тогда как выражение (x == y) сравнивает текущие значения двух переменных. В некоторых языках программирования это различие способно устроить хаос и породить коварные ошибки, но в Java на помощь обычно приходит проверка безопасности типов. Например, если по ошибке написать (cash = goal) вместо (cash == goal) в программе 1.3.8, компилятор обнаружит ошибку:

```
javac Gambler.java
Gambler.java:18: incompatible types
found : int
required: boolean
if (cash = goal) wins++;
    ^
1 error
```

Будьте особенно внимательны с условиями (x = y), когда переменные x и y относятся к типу boolean; такая команда будет интерпретирована как присваивание значения y переменной x, а ее результатом станет логическое значение y. Например, следует использовать запись if (!isPrime) вместо (isPrime = false).

В. Итак, при написании условий для циклов и ветвлений нужно помнить, что вместо = используется ==. Еще что-нибудь в этом же роде?

О. Другая распространенная ошибка — пропущенные фигурные скобки в цикле или ветвлении с телом, состоящим из нескольких строк кода. Например, возьмем следующую версию кода из Gambler:

```
for (int t = 0; t < trials; t++)
   for (cash = stake; cash > 0 && cash < goal; bets++)
      if (Math.random() < 0.5) cash++;
      else                     cash--;
   if (cash == goal) wins++;
```

Этот код кажется правильным, но он не работает, потому что вторая операция if находится вне обоих циклов for и выполняется только один раз. Многие программисты всегда заключают тело цикла в фигурные скобки именно для того, чтобы избежать таких коварных ошибок.

В. Что-нибудь еще?

О. Третья классическая ловушка — неоднозначная интерпретация вложенных команд if:

```
if <выражение1> if <выражение2> <командаA> else <командаB>
```

В языке Java эта команда эквивалентна следующей:

```
if <выражение1> { if <выражение2> <командаA> else <командаB> }
```

даже если вы имели в виду

```
if <выражение1> { if <выражение2> <командаA> } else <командаB>
```

Еще раз: лучший способ предотвращения этой ловушки — явное использование фигурных скобок.

В. Есть ли ситуации, в которых нужно использовать обязательно цикл `for`, но не `while`, или наоборот?

О. Нет. В общем случае цикл `for` используется там, где есть инициализация, приращение и условие продолжения цикла (если вы не планируете использовать управляющую переменную цикла за пределами цикла). Но и в этом случае эквивалентный цикл `while` вполне применим.

В. Какие правила устанавливаются для объявлений управляющих переменных цикла?

О. Есть разные точки зрения. В традиционных языках программирования все переменные должны были объявляться в начале блока; многие программисты привыкли к этому, и в значительной части нового кода соблюдается это же правило. Конечно, вполне разумно объявлять переменные при первом использовании, особенно в циклах `for` (если переменная, как это часто бывает, не используется за пределами цикла). С другой стороны, не так уж редко управляющую переменную цикла приходится проверять (а следовательно, и объявлять) вне цикла, как в коде проверки простых чисел, рассмотренном в примере команды `break`.

В. Чем отличается запись `++i` и `i++`?

О. Если рассматривать их как операции — различий нет. В выражении оба варианта записи увеличивают `i`, но результат операции `++i` определяется *после* увеличения `i`, а значение `i++` — *до* увеличения[1]. В этой книге мы избегаем строк вида `x = ++i`, обладающих побочным эффектом изменения значений переменных. Следовательно, можно не беспокоиться о различиях и просто использовать запись `i++` в циклах `for` и в других командах. Используя `++i` в книге, мы обязательно обратим на это ваше внимание и укажем, почему это делается.

В. В цикле `for` секции <*инициализация*> и <*приращение*> могут содержать более сложные действия, не ограничивающиеся простым объявлением, инициализацией и обновлением управляющей переменной цикла. Как использовать эту возможность?

О. Секции <*инициализация*> и <*приращение*> могут содержать *последовательности* операций, разделенных запятыми. Данная форма записи позволяет писать код, который инициализирует и изменяет также и другие переменные, а не только

[1] Здесь разъясняется различие между префиксной и постфиксной записью инкремента, которое состоит в результате, возвращаемом самой операцией, в то время как «побочный эффект» (приращение переменной) совершенно одинаков. То же справедливо и для декремента `--`. — *Примеч. науч. ред.*

управляющую переменную цикла. В некоторых случаях код становится более компактным. Например, следующие две строки могут заменить последние восемь строк метода `main()` в программе `PowersOfTwo` (листинг 1.3.3):

```
for (int i = 0, power = 1; i <= n; i++, power *= 2)
   System.out.println(i + " " + power);
```

Такой код редко бывает действительно необходимым. Лучше избегать его, особенно новичкам.[1]

В. Можно ли использовать переменную типа `double` в качестве управляющей переменной цикла `for`?

О. Можно, но обычно так поступать не рекомендуется. Возьмем следующий цикл:

```
for (double x = 0.0; x <= 1.0; x += 0.1)
   System.out.println(x + " " + Math.sin(x));
```

Сколько раз он будет выполняться? Количество итераций зависит от проверки равенства между переменными типа `double`, которая не всегда дает ожидаемый результат из-за точности операций с плавающей точкой.

В. У циклов есть еще какие-нибудь нюансы?

О. Некоторые части цикла `for` могут не содержать кода. Команда инициализации, условие цикла, операция приращения, тело цикла — любые из этих компонентов могут быть опущены. Каноны хорошего стиля программирования рекомендуют использовать цикл `while` вместо цикла `for` с пустыми командами. В примерах книги мы стараемся не использовать *пустые операции*.

```
int power = 1;
while (power <= n/2)
   power *= 2;
for (int power = 1; power <= n/2; )
   power *= 2;
for (int power = 1; power <= n/2; power *= 2)
   ;
```

```
int power = 1;
while (power <= n/2)
   power *= 2;

for (int power = 1; power <= n/2; )
   power *= 2;

for (int power = 1; power <= n/2; power *= 2)
   ;
```

пустая операция приращения (указывает на пустое место перед `)` во втором цикле)

пустое тело цикла (указывает на `;` в третьем цикле)

Три эквивалентных цикла

[1] Злоупотребление подобными возможностями «уплотнения» нередко делает исходный текст малопригодным для понимания и сопровождения. — *Примеч. науч. ред.*

Упражнения

1.3.1. Напишите программу, которая получает три целых числа как аргументы командной строки и выводит сообщение "`equal`", если все три числа равны, или "`not equal`" в противном случае.

1.3.2. Напишите более универсальную и надежную версию программы `Quadratic` (листинг 1.2.3), которая выводит корни многочлена $ax^2 + bx + c$, либо соответствующее сообщение, если дискриминант отрицателен, при этом избегая деления на нуль, если значение a равно нулю.

1.3.3. Какая ошибка допущена в каждой из следующих команд (а может, ошибки нет)?

```
a.   if (a > b) then c = 0;
b.   if a > b { c = 0; }
c.   if (a > b) c = 0;
d.   if (a > b) c = 0 else b = 0;
```

1.3.4. Напишите фрагмент кода, который выводит сообщение "`true`", если переменные `x` и `y` типа `double` лежат строго в диапазоне от 0 до 1, или "`false`" в противном случае.

1.3.5. Напишите программу `RollLoadedDie`, которая моделирует результат броска «смещенного» игрального кубика. Грани 1, 2, 3, 4 и 5 выпадают с вероятностью 1/8, а грань 6 — с вероятностью 3/8.

1.3.6. Усовершенствуйте свое решение упражнения 1.2.25: добавьте код, который проверяет, что значения аргументов командной строки лежат в диапазонах допустимых значений формулы, а также код для вывода сообщения об ошибке при нарушении ограничений.

1.3.7. Допустим, переменные `i` и `j` относятся к типу `int`. Какое значение принимает переменная `j` после выполнения каждой из следующих команд?

```
a.   for (i = 0, j = 0; i < 10; i++)  j += i;
b.   for (i = 0, j = 1; i < 10; i++)  j += j;
c.   for (j = 0; j < 10; j++) j += j;
d.   for (i = 0, j = 0; i < 10; i++) j += j++;
```

1.3.8. Перепишите программу `TenHellos` и создайте программу `Hellos`, которая получает в аргументе командной строки количество повторений. Можно предположить, что аргумент меньше 1000. Подсказка: используйте выражения `i % 10` и `i % 100`, чтобы определить, какой из суффиксов числительных — `st`, `nd`, `rd` или `th` —должен использоваться при выводе i-го сообщения "`Hello`".

1.3.9. Напишите программу с одним циклом `for` и одной командой `if`, которая выводит целые числа от 1000 до 2000, по 5 чисел на строку. Подсказка: используйте оператор `%`.

1.3.10. Напишите программу, которая получает целое число `n` как аргумент командной строки, использует `Math.random()` для вывода `n` случайных значений с равно-

мерным распределением в интервале от 0 до 1, после чего выводит их среднее значение (см. упражнение 1.2.30).

1.3.11. Опишите, что произойдет при попытке вызвать функцию расчета делений на линейке (см. листинг на с. 42) со слишком большим значением `n` — например, 100.

1.3.12. Напишите программу `FunctionGrowth` для вывода таблицы значений `log n`, `n`, `n log n`, n^2, n^3 и 2^n для `n` = 16, 32, 64, ..., 2048. Используйте символы табуляции (символы `\t`) для выравнивания столбцов.

1.3.13. Какие значения принимают переменные `m` и `n` после выполнения следующего кода?

```
int n = 123456789;
int m = 0;
while (n != 0)
{
   m = (10 * m) + (n % 10);
   n = n / 10;
}
```

1.3.14. Что выводит следующий фрагмент кода?

```
int f = 0, g = 1;
for (int i = 0; i <= 15; i++)
{
   System.out.println(f);
   f = f + g;
   g = f - g;
}
```

Решение. Даже опытный программист скажет вам, что такую программу можно понять только одним способом — просмотреть трассировку. Из данных трассировки видно, что программа выводит значения `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, 233, 377` и `610`. Это первые 16 чисел из знаменитой последовательности *чисел Фибоначчи*, определяемой следующими формулами: $F_0 = 0$, $F_1 = 1$, и $F_n = F_{n-1} + F_{n-2}$ для $n > 1$.

1.3.15. Сколько строк выведет следующий фрагмент кода?

```
for (int i = 0; i < 999; i++);
{  System.out.println("Hello");  }
```

Решение. Одну. Обратите внимание на неприметную точку с запятой в конце первой строки.

1.3.16. Напишите программу, которая получает целое число `n` как аргумент командной строки и выводит все положительные степени 2, меньшие или равные `n`. Убедитесь в том, что ваша программа правильно работает для всех значений `n`.

1.3.17. Доработайте свое решение упражнения 1.2.24, чтобы программа выводила общую накопленную сумму и текущие проценты после каждого ежемесячного перерасчета суммы вклада.

1.3.18. В отличие от гармонических чисел, сумма $1/1^2 + 1/2^2 + ... + 1/n^2$ сходится к константе при стремлении n к бесконечности. (Эта константа равна $\pi^2/6$, поэтому формула может использоваться для вычисления приближенного значения π.) Какой из следующих циклов `for` вычисляет эту сумму? Предполагается, что `n` — переменная типа `int`, инициализированная значением `1000000`, а `sum` — переменная типа `double`, инициализированная значением `0.0`.

```
a.  for (int i = 1; i <= n; i++) sum += 1 / (i*i);
b.  for (int i = 1; i <= n; i++) sum += 1.0 / i*i;
c.  for (int i = 1; i <= n; i++) sum += 1.0 / (i*i);
d.  for (int i = 1; i <= n; i++) sum += 1 / (1.0*i*i);
```

1.3.19. Покажите, что листинг 1.3.6 реализует метод Ньютона для поиска квадратного корня c. Подсказка: воспользуйтесь тем фактом, что угол наклона касательной дифференцируемой функции $f(x)$ в точке $x = t$ равен $f'(t)$, для нахождения уравнения касательной, после чего воспользуйтесь уравнением для определения точки пересечения касательной с осью x. Покажите, что метод Ньютона может использоваться для нахождения корня произвольной функции, следующим образом: при каждой итерации замените очередную точку приближения t значением $t - f(t)/f'(t)$.

1.3.20. Используя метод Ньютона, разработайте программу, которая получает два целых числа `n` и `k` как аргументы командной строки и выводит корень `k`-й степени из `n`. (*Подсказка*: см. упражнение 1.3.19.)

1.3.21. На основе программы `Binary` создайте программу `Kary`, которая получает два целых числа `i` и `k` в аргументах командной строки и преобразует `i` в систему счисления с основанием `k`. Предполагается, что `i` — целое число с типом данных Java `long`, а `k` — целое число в диапазоне от 2 до 16. В системах счисления с основанием больше 10 для представления цифр, соответствующих десятичным значениям с 111 до 16, используйте буквы A–F.

1.3.22. Напишите фрагмент кода, который переводит двоичное представление положительного целого числа `n` в переменную `s` типа `String`.

Решение. В Java существует встроенный метод `Integer.toBinaryString(n)` для решения этой задачи, но суть этого упражнения — проработка возможной реализации этого метода. Взяв за основу листинг 1.3.7, мы получаем решение:

```
String s = "";
int power = 1;
while (power <= n/2) power *= 2;
while (power > 0)
{
   if (n < power) { s += 0;             }
   else           { s += 1; n -= power; }
   power /= 2;
}
```

Более простое решение основано на обработке справа налево:

```
String s = "";
for (int i = n; i > 0; i /= 2)
   s = (i % 2) + s;
```

Оба метода заслуживают тщательного изучения.

1.3.23. Напишите версию программы Gambler, использующую два вложенных цикла while или два вложенных цикла for (вместо цикла while в цикле for).

1.3.24. Напишите программу GamblerPlot, которая трассирует модель задачи о разорении игрока и выводит после каждой ставки строку, в которой для каждого доллара на руках у игрока выводится звездочка.

1.3.25. Внесите изменения в программу Gambler, чтобы она получала дополнительный аргумент командной строки с фиксированным значением вероятности выигрыша каждой ставки. При помощи своей программы попробуйте выяснить, как эта вероятность влияет на вероятность выигрыша и ожидаемое количество ставок. Опробуйте значения p, близкие к 0,5 (например, 0.48).

1.3.26. Внесите изменения в программу Gambler, чтобы она получала дополнительный аргумент командной строки с количеством ставок, которые готов сделать игрок, чтобы у игры было три возможных исхода: игрок выигрывает, игрок проигрывает или у игрока кончается время. Добавьте в вывод ожидаемую сумму денег, которая будет на руках у игрока при завершении игры. Задание для самостоятельной работы: используйте программу для планирования поездки в Монте-Карло.

1.3.27. Внесите изменения в программу Factors, чтобы каждый простой множитель выводился только в одном экземпляре.

1.3.28. Экспериментальным путем определите, как влияет использование в программе Factors (листинг 1.3.9) условия завершения (factor <= n/factor) вместо (factor < n). Для каждого метода найдите наибольшее значение n, с которым при вводе числа из n цифр программа гарантированно завершается за 10 секунд.

1.3.29. Напишите программу Checkerboard, которая получает целое число n как аргумент командной строки и использует вложенные циклы для вывода двумерного узора «шахматной доски» из чередующихся пробелов и звездочек.

1.3.30. Напишите программу GreatestCommonDivisor для нахождения наибольшего общего делителя (НОД) двух чисел по алгоритму Евклида. Этот итерационный алгоритм основан на следующем наблюдении: если x больше y, то если x делится на y, то НОД x и y равен y; в противном случае НОД x и y равен НОД x % y и y.

1.3.31. Напишите программу RelativelyPrime, которая получает целое число n как аргумент командной строки и выводит таблицу n × n, в которой в строке i и столбце j выводится *, если НОД i и j равен 1 (i и j — взаимно простые числа), или пробел в противном случае.

1.3.32. Напишите программу PowersOfK, которая получает целое число k как аргумент командной строки и выводит все положительные степени k из типа данных

Java `long`. *Примечание*: наибольшее значение типа `long` задается константой `Long.MAX_VALUE`.

1.3.33. Напишите программу для вывода координат случайной точки `(a, b, c)` на поверхности сферы. Для генерирования точек используйте *метод Марсальи*: начните с выбора случайной точки (x, y) на единичном круге (способ описан в конце раздела), затем присвойте a значение $2x\sqrt{1-x^2-y^2}$, b — значение $2\sqrt{1-x^2-y^2}$ и c — значение $1 - 2(x^2 + y^2)$.

1.3.34. *Такси Рамануджана.* Сриниваса Рамануджан — индийский математик, прославившийся своей интуицией в отношении чисел. Когда английский математик Годфри Харди однажды был у него в гостях, то заметил, что номер такси, на котором он приехал, — 1729 — скучное число. На это Рамануджан ответил: «Нет-нет, Харди! Это очень интересное число. Это наименьшее число, которое может быть выражено в виде суммы двух кубов двумя разными способами». Проверьте это утверждение — напишите программу, которая получает целое число `n` в командной строке и выводит все целые числа, меньшие или равные `n`, которые могут быть выражены в виде суммы двух кубов двумя разными способами. Иначе говоря, найдите положительные числа a, b, c и d, для которых $a^3 + b^3 = c^3 + d^3$. В программе используйте четыре вложенных цикла `for`.

1.3.35. *Контрольная сумма.* Код ISBN (International Standard Book Number) представляет собой код из 10 цифр, который однозначно определяет книгу. Последняя цифра образует контрольную сумму и однозначно вычисляется по 9 другим цифрам из условия, согласно которому значение $d_1 + 2d_2 + 3d_3 + ... + 10d_{10}$ должно быть кратно 11 (здесь d_i — i-я цифра справа). Контрольная цифра d_1 может иметь любое значение от 0 до 10. По правилам ISBN для обозначения 10 используется символ 'X'. Например, контрольная цифра для последовательности 020131452 равна 5, потому что 5 — единственное значение x от 0 до 10, для которого

$$10\cdot0 + 9\cdot2 + 8\cdot0 + 7\cdot1 + 6\cdot3 + 5\cdot1 + 4\cdot4 + 3\cdot5 + 2\cdot2 + 1\cdot x$$

кратно 11. Напишите программу, которая получает в аргументе командной строки целое число из 9 цифр, вычисляет контрольную сумму и выводит код ISBN.

1.3.36. *Подсчет простых чисел.* Напишите программу `PrimeCounter`, которая получает целое число `n` как аргумент командной строки и выводит количество простых чисел, меньших или равных `n`. Используйте программу для вывода количества простых чисел, меньших или равных 10 000 000. *Примечание*: если действовать бездумно, ваша программа вряд ли завершится за разумное время!

1.3.37. *Случайное блуждание на плоскости.* Случайное блуждание моделирует поведение частицы, двигающейся по сетке из точек. На каждом шаге блуждающая частица выбирает перемещение на север, на юг, на восток или на запад с вероятностью 1/4 независимо от предыдущих перемещений. Напишите программу `RandomWalker`, которая получает целое число `n` в аргументе командной строки и оценивает, сколько перемещений потребуется для достижения границы квадрата `2n` × `2n` с центром в начальной точке.

1.3.38. *Экспоненциальная функция.* Пусть x — положительная переменная типа double. Напишите фрагмент кода, использующий разложение в ряд Тейлора для вычисления значения sum, равного $e^x = 1 + x + x^2/2! + x^3/3! + ...$

Решение. Цель этого упражнения — дать представление о том, как библиотечная функция Math.exp() могла бы быть реализована на уровне элементарных операций. Попробуйте решить эту задачу, а затем сравните свое решение с приведенным ниже.

Начнем с задачи вычисления одного слагаемого. Допустим, x и term — переменные типа double, а n — переменная типа int. Следующий фрагмент кода присваивает term значение $x^n/n!$ путем прямого вычисления, где один цикл используется для числителя, а другой — для знаменателя, с последующим делением результатов:

```
double num = 1.0, den = 1.0;
for (int i = 1; i <= n; i++) num *= x;
for (int i = 1; i <= n; i++) den *= i;
double term = num/den;
```

В такой ситуации лучше использовать один цикл for:

```
double term = 1.0;
for (i = 1; i <= n; i++) term *= x/i;
```

Кроме компактности и элегантности, последнее решение предпочтительно еще и из-за отсутствия погрешностей, возникающих при вычислениях с большими вещественными числами. Например, решение с двумя циклами перестает работать с x = 10 и n = 100, потому что значение 100! слишком велико для представления в формате double.

Для вычисления e^x следует вложить этот цикл for в другой цикл for:

```
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; n++)
{
   sum += term;
   term = 1.0;
   for (int i = 1; i <= n; i++) term *= x/i;
}
```

Количество итераций цикла зависит от соотношения между следующим слагаемым и накопленной суммой. После того как значение sum перестанет изменяться, цикл завершается. (Эта стратегия эффективнее, чем условие продолжения цикла (term > 0), потому что она предотвращает большое количество итераций, не изменяющих величину sum.) Этот код работает, но в целом он неэффективен, потому что внутренний цикл for пересчитывает все значения, вычисленные им на предыдущей итерации внешнего цикла for. Вместо этого можно воспользоваться слагаемым, добавленным на предыдущей итерации, и решить задачу с одним циклом for:

```
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; n++)
{
   sum += term;
   term *= x/n;
}
```

1.3.39. *Тригонометрические функции.* Напишите две программы `Sin` и `Cos`, которые вычисляют значения функций синуса и косинуса с использованием разложения в ряд Тейлора:

$$\sin x = x - x^3/3! + x^5/5! - \ldots \text{ и } \cos x = 1 - x^2/2! + x^4/4! - \ldots .$$

1.3.40. *Экспериментальный анализ.* Экспериментальным путем оцените относительную трудоемкость `Math.exp()` и методов из упражнения 1.3.38 для вычисления e^x: прямого решения с вложенными циклами `for`, усовершенствования с одним циклом `for` и решения с условием продолжения цикла `(term > 0)`. Используя аргумент командной строки, определите опытным путем, сколько раз ваш компьютер сможет выполнить каждое вычисление за 10 секунд.

1.3.41. *Задача Пеписа.* В 1693 году Сэмюэл Пепис спросил у Исаака Ньютона, что более вероятно: получить 1 хотя бы один раз при броске 6 правильных игральных костей или получить 1 хотя бы дважды при броске 12 костей. Напишите программу, с помощью которой Ньютон мог бы быстро получить ответ.

1.3.42. *Модель игры.* В телевизионной игре «Let's Make a Deal» участник оказывается перед тремя дверями. За одной из дверей находится ценный приз. После того как участник выберет дверь, ведущий открывает одну из двух других дверей (естественно, никогда не открывая приз), и участнику предоставляется возможность выбрать другую неоткрытую дверь. Стоит ли участнику поступить так? На интуитивном уровне может показаться, что приз может с равной вероятностью находиться за дверью, изначально выбранной участником, и другой неоткрытой дверью, так что причин для изменения выбора нет. Напишите программу `MonteHall` для проверки интуитивного представления посредством моделирования. Ваша программа должна получать аргумент командной строки `n`, проводить игру `n` раз с использованием каждой из двух стратегий (менять или не менять выбор) и выводить вероятность успеха для каждой из двух стратегий.

1.3.43. *Медиана 5 чисел.* Напишите программу, которая получает в аргументах командной строки пять разных целых чисел и выводит их медиану (то из этих чисел, которое меньше двух из оставшихся четырех и больше других двух из них). Задание повышенной сложности: решите задачу при помощи программы, которая сравнивает значения менее 7 раз для любого заданного ввода.

1.3.44. Сортировка 3 чисел. Допустим, переменные `a`, `b`, `c` и `t` относятся к типу `int`. Объясните, почему следующий код упорядочивает `a`, `b` и `c` по возрастанию:

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

1.3.45. *Хаос.* Напишите программу для анализа простой модели роста численности популяции: рыб в пруду, бактерий в пробирке или любой другой аналогичной ситуации. Предполагается, что численность популяции изменяется в диапазоне от 0 (вымирание) до 1 (максимальная жизнеспособная численность.) Если в момент времени `t` численность популяции составляет `x`, предполагается, что в момент `t+1` она составит `rx(1-x)`, где аргумент `r`, называемый *плодовитостью*, определяет скорость размножения. Начните с малой популяции (допустим, `x = 0.01`) и изучите результат моделирования для разных значений `r`. Для каких значений `r` популяция стабилизируется на уровне `x = 1 - 1/r`? Можно ли сказать что-нибудь о популяции, если `r` равно 3.5? 3.8? 5?

1.3.46. *Гипотеза Эйлера.* В 1769 году Леонард Эйлер сформулировал обобщенную версию так называемой большой теоремы Ферма, которая предполагала, что для получения суммы, являющейся n-й степенью, требуется не менее n слагаемых n-й степени (для $n > 2$). Напишите программу для опровержения гипотезы Эйлера (которая продержалась до 1967 года); используя пятикратно вложенный цикл для нахождения 4 положительных целых чисел, сумма 5-х степеней которых дает 5-ю степень другого положительного целого числа. Иначе говоря, найдите такие a, b, c, d и e, что $a^5 + b^5 + c^5 + d^5 = e^5$. Используйте тип данных `long`.

1.4. Массивы

В этом разделе будет представлена сама концепция структурированных данных, а также ваша первая структура данных — *массив*. Массивы создаются прежде всего для хранения и обработки больших объемов данных. Массивы играют важную роль во многих задачах обработки данных. Они также соответствуют векторам и матрицам, широко применяемым в науке и научном программировании. Мы рассмотрим основные свойства массивов в языке Java, а многочисленные примеры наглядно продемонстрируют их полезность.

a
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]

Массив

Структура данных представляет собой способ организации данных в компьютере (обычно для экономии времени или памяти.) Структуры данных играют важную роль в программировании, — поэтому глава 4 этой книги посвящена изучению различных классических структур данных.

Одномерный массив (или просто *массив*) — структура данных для хранения последовательности значений, относящихся к одному типу. Компоненты массива называются его *элементами*. Доступ к элементам в массиве осуществляется по их

индексам: если массив состоит из *n* элементов, мы считаем, что эти элементы пронумерованы от `0` до `n-1`[1], что позволяет однозначно идентифицировать элемент по его целочисленному индексу в этом диапазоне.

Двумерный массив может рассматриваться как массив одномерных массивов. Если элементы одномерного массива индексируются одним целым числом, элементы двумерного массива индексируются двумя целыми числами: первый индекс задает строку, а второй — столбец.

При обработке больших объемов данных все данные часто заранее помещаются в один или несколько массивов. Затем индексы используются для обращения к отдельным элементам и обработки данных: результатов экзаменов, биржевых котировок, нуклеотидов в цепи ДНК, символов в книге и т. д. В каждом из этих примеров используется большое количество однотипных значений. Такие приложения будут представлены при рассмотрении ввода/вывода в разделе 1.5 и в примере из раздела 1.6. А в этом разделе мы представим основные свойства массивов на примерах, в которых наши программы сначала заполняют массивы, а затем обрабатывают их.

Массивы в Java

Создание массива в программе на Java состоит из трех шагов:

- объявление массива;
- создание массива;
- инициализация элементов массива.

Чтобы *объявить* массив, необходимо указать имя массива и тип содержащихся в нем данных. Чтобы *создать* массив, необходимо задать его длину (количество элементов). Наконец, *инициализация* подразумевает присваивание значения каждому из элементов массива. Например, следующий код создает массив из *n* элементов, каждый из которых относится к типу `double`, и инициализирует каждый элемент значением `0.0`:

```
double[] a;                    // Объявление массива
a = new double[n];             // Создание массива
for (int i = 0; i < n; i++)    // Инициализация массива
   a[i] = 0.0;
```

Первая строка содержит *объявление массива*. Оно почти не отличается от объявления переменной соответствующего примитивного типа, если не считать квадратных скобок за именем типа; скобки указывают, что объявляется именно массив. Вторая строка *создает* массив; ключевое слово `new` выделяет память для хранения задан-

[1] Справедливо для C, Java и ряда других языков программирования. Однако существуют языки, где индексация может начинаться с 1 или даже с произвольно выбранного значения. — *Примеч. науч. ред.*

ного количества элементов[1]. Для переменных примитивного типа это действие необязательно, но оно необходимо для всех остальных типов данных в Java (см. раздел 3.1). Цикл `for` присваивает значение `0.0` каждому из `n` элементов массива. Чтобы обозначить конкретный элемент массива, укажите его индекс в квадратных скобках за именем массива: конструкция `a[i]` обозначает `i`-й элемент массива `a[]`. (В тексте книги мы используем запись `a[]`, чтобы указать, что переменная `a` является массивом, но в коде Java запись `a[]` не используется.)

К очевидным преимуществам массивов относится возможность определения множества переменных (элементов данных) без явного указания их имен. Например, если вы хотите работать с 8 переменными типа `double`, их можно объявить командой:

```
double a0, a1, a2, a3, a4, a5, a6, a7;
```

а затем ссылаться на них в коде `a0`, `a1`, `a2` и т. д. С другой стороны, присваивать имена десяткам отдельных переменных неудобно, а миллионам — нереально. Вместо этого можно объявить `n` переменных в массиве командой `double[] a = new double[n]`, а затем обращаться к переменным по именам `a[0]`, `a[1]`, `a[2]` и т. д. Теперь ничто не мешает вам определять десятки или миллионы переменных. Более того, так как в качестве индекса массива часто используется переменная (или другое выражение, вычисляемое во время выполнения программы), вы можете обработать сколько угодно элементов в одном цикле, как это сделано выше. Элемент массива рассматривается как отдельная переменная, обращение к которой можно поместить в выражении или в левой части команды присваивания.

В нашем первом примере массивы будут использоваться для представления *векторов*. Векторы подробно рассматриваются в разделе 3.3; пока считайте, что вектор — это последовательность вещественных чисел. *Скалярное произведение* двух векторов (одинаковой длины) определяется как сумма произведений соответствующих им элементов. Так, скалярное произведение двух векторов, представленных одномерными массивами `x[]` и `y[]`, каждый из которых имеет длину 3, определяется выражением `x[0]*y[0] + x[1]*y[1] + x[2]*y[2]`. Или в более общем виде, если каждый массив имеет длину `n`, скалярное произведение этих массивов вычисляется следующим кодом:

```
double sum = 0.0;
for (int i = 0; i < n; i++)
   sum += x[i]*y[i];
```

Из-за простоты реализации таких вычислений массивы становятся естественным выбором для самых разных применений.

[1] Это *динамическое* создание массива. Далее будет рассмотрено также и *статическое* (во время компиляции) выделение массивов. — *Примеч. науч. ред.*

i	x[i]	y[i]	x[i]*y[i]	sum
				0.00
0	0.30	0.50	0.15	0.15
1	0.60	0.10	0.06	0.21
2	0.10	0.40	0.04	0.25
				0.25

Трассировка вычисления скалярного произведения

В таблице на следующей странице приведены примеры кода, работающего с массивами. Еще больше примеров встретится в книге позднее, потому что массивы играют центральную роль в обработке данных во многих приложениях. Прежде чем браться за более сложные примеры, необходимо отметить некоторые важные особенности программирования с использованием массивов.

Индексы начинаются с нуля

Первый элемент массива `a[]` обозначается `a[0]`, второй — `a[1]` и т. д. Казалось бы, более естественно обозначить первый элемент `a[1]`, второй — `a[2]` и т. д., но индексирование с 0 имеет свои преимущества. Сейчас оно стало фактическим стандартом, применяемым в большинстве современных языков программирования. Неправильное понимание этого правила часто ведет к ошибкам «смещения на 1», которых очень трудно избежать — будьте внимательны!

Длина массива

После того как вы создадите массив в Java, его длина остается фиксированной. Одна из причин, по которой массивы должны явно создаваться во время выполнения, заключается в том, что компилятор Java не всегда знает объем необходимой памяти для массива во время компиляции (потому что его длина может оставаться неизвестной до момента выполнения). С другой стороны, явно выделять память для переменных типа `int` или `double` не нужно, потому что их размер фиксирован и известен во время компиляции. Длина массива `a[]` содержится в `a.length`. Обратите внимание: к последнему элементу массива `a[]` всегда можно обратиться в виде `a[a.length-1]`. Для удобства мы часто сохраняем длину массива в целочисленной переменной `n`.

Инициализация массива по умолчанию

Для более компактной записи кода часто используется механизм *инициализации по умолчанию* языка Java, позволяющий объявить, создать и инициализировать массив в одной строке. Например, следующая строка эквивалентна коду на с. 107:

```
double[] a = new double[n];
```

создание массива со случайными значениями	`double[] a = new double[n];`
	`for (int i = 0; i < n; i++)`
	`   a[i] = Math.random();`

вывод значений из массива по одному в строке	`for (int i = 0; i < n; i++)` `   System.out.println(a[i]);`

определение максимального значения в массиве	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < n; i++)`
	`   if (a[i] > max) max = a[i];`

вычисление среднего арифметического по значениям массива	`double sum = 0.0;`
	`for (int i = 0; i < n; i++)`
	`   sum += a[i];`
	`double average = sum / n;`

обратная перестановка элементов массива	`for (int i = 0; i < n/2; i++)`
	`{`
	`   double temp = a[i];`
	`   a[i] = a[n-1-i];`
	`   a[n-i-1] = temp;`
	`}`

копирование последовательности элементов в другой массив	`double[] b = new double[n];`
	`for (int i = 0; i < n; i++)`
	`   b[i] = a[i];`

Типичный код, работающий с массивами (для массива a[ ], содержащего n значений типа double)[1]

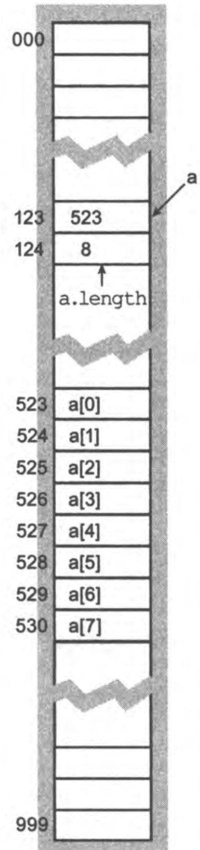
Слева от знака = располагается код объявления; в правой части — код создания. Цикл `for` в данном случае не нужен, потому что Java автоматически инициализирует элементы массива примитивного типа нулями (для числовых типов) или `false` (для типа `boolean`). Java автоматически инициализирует элементы массива типа `String` (и других непримитивных типов) `null` — специальным значением, о котором вы узнаете в главе 3.

Представление в памяти

Массивы относятся к числу фундаментальных структур данных еще и потому, что они напрямую связаны с расположением данных в памяти практически на всех компьютерах. Элементы массива хранятся в памяти последовательно, что позволяет быстро обратиться к любому элементу. Собственно, вся оперативная память может рассматриваться как гигантский массив. На современных компьютерах память реализуется как последовательность ячеек, к которым можно быстро обращаться по соответствующему индексу. При обращении к памяти компьютера индекс ячейки обычно называется *адресом*. Удобно считать, что в имени массива — допустим, `a` — хранится адрес первого элемента массива `a[0]`. Для наглядности предпо-

[1] Можно также обойтись без обращений к константе: первым «кандидатом на максимум» принимается первый элемент массива, а сравнения начинаются со второго его элемента. — *Примеч. науч. ред.*

ложим, что память компьютера может содержать 1000 значений с адресами от 000 до 999. (Эта упрощенная модель игнорирует тот факт, что элементы массива могут занимать разный объем памяти в зависимости от их типа, но сейчас на такие подробности можно не обращать внимания.) Теперь предположим, что массив с 8 элементами хранится в ячейках памяти с 523-го по 530-й. Тогда Java хранит где-то в памяти адрес (индекс) первого элемента массива вместе с длиной массива. Такой адрес называется *указателем*, поскольку он «указывает» на заданную ячейку памяти. Встречая обращение `a[i]`, компилятор генерирует код, который обращается к нужному значению, прибавляя индекс `i` к адресу в памяти массива `a[]`. Например, код Java `a[4]` сгенерирует машинный код, который находит значение в ячейке памяти `523 + 4 = 527`. Обращение к элементу `i` массива выполняется эффективно, потому что для него достаточно просто сложить два целых числа, а затем обратиться к памяти по вычисленному адресу — всего две элементарные операции.



Представление в памяти

Выделение памяти

Когда вы используете ключевое слово `new` для создания массива, Java резервирует память, необходимую для хранения заданного количества элементов. Этот процесс называется выделением памяти. Этот же процесс необходим для всех переменных, встречающихся в программе (однако ключевое слово `new` не используется для переменных примитивных типов, потому что Java знает, сколько памяти для них нужно выделить.) Мы обращаем на это ваше внимание, потому что вы несете ответственность за создание массива перед обращением к любым его элементам. Если это не будет сделано, то во время компиляции вы получите сообщение об ошибке «переменная не инициализирована».

Контроль границ

Как упоминалось ранее, при использовании массивов в программе необходима осторожность. Следите за тем, чтобы при обращении к элементу массива использовались правильные индексы. Если вы создали массив длины `n`, то при использовании индекса, значение которого меньше 0 или больше `n-1`, ваша программа завершится с исключением времени исполнения `ArrayIndexOutOfBoundsException`. (Во многих языках программирования условия переполнения буфера не отслеживаются системой.) Такие *непроверяемые* ошибки часто оборачиваются сущим кошмаром во время отладки, но иногда они остаются незамеченными и в завершенной программе. Такие ошибки могут использоваться хакерами для захвата контроля

над системой (в том числе и вашего персонального компьютера), распространения вирусов, хищения персональной информации или других неблаговидных деяний. Сообщения об ошибках, которые выдает Java, поначалу могут раздражать, но это не большая цена за повышение надежности и безопасности программы.

Присваивание значений элементам массива на стадии компиляции

Если количество значений, хранимых в массиве, относительно невелико и известно заранее, то вы можете объявить, создать и инициализировать массив, перечислив эти значения в фигурных скобках и разделив их запятыми. Например, следующий код может использоваться в программе, моделирующей карточную игру:

```
String[] SUITS = { "Clubs", "Diamonds", "Hearts", "Spades" };

String[] RANKS =
{
   "2", "3", "4", "5", "6", "7", "8", "9", "10",
   "Jack", "Queen", "King", "Ace"
};
```

Теперь эти два массива могут использоваться для вывода названия случайной карты (например, `Queen of Clubs`):

```
int i = (int) (Math.random() * RANKS.length);
int j = (int) (Math.random() * SUITS.length);
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

В этом коде при помощи идиомы, описанной в разделе 1.2, генерируются два случайных индекса, после чего индексы используются для выборки строк из двух массивов. Каждый раз, когда известны значения всех элементов массива (и длина массива не слишком велика), этот метод инициализации оправдан — просто перечислите все значения в фигурных скобках справа от знака равенства в объявлении массива. Такая конструкция подразумевает создание массива, так что ключевое слово `new` необязательно.

Присваивание значений элементам массива во время выполнения

Чаще значения, сохраняемые в массиве, должны вычисляться во время выполнения программы. В таком случае имя массива с индексом используется точно так же, как мы используем имена переменных в левой части оператора присваивания. Например, следующий код инициализирует массив длины 52, представляющий колоду игральных карт, с использованием двух только что определенных массивов:

```
String[] deck = new String[RANKS.length * SUITS.length];
for (int i = 0; i < RANKS.length; i++)
   for (int j = 0; j < SUITS.length; j++)
      deck[SUITS.length*i + j] = RANKS[i] + " of " + SUITS[j];
```

Если после выполнения этого кода вывести содержимое `deck[]` от `deck[0]` до `deck[51]`, результат будет выглядеть так:

```
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
...
Ace of Hearts
Ace of Spades
```

Перестановка двух значений в массиве

На практике часто бывает нужно поменять местами значения двух элементов в массиве. Продолжая пример с игральными картами, следующий код меняет местами карты с индексами `i` и `j` с использованием идиомы, которая была приведена в первом примере команд присваивания из раздела 1.2:

```
String temp = deck[i];
deck[i] = deck[j];
deck[j] = temp;
```

Например, если выполнить этот фрагмент для переменной `i`, равной 1, переменной `j`, равной 4, и массива `deck[]` из предыдущего примера, в `deck[1]` окажется значение `3 of Clubs`, а в `deck[4]` — `2 of Diamonds`. Вы также можете убедиться в том, что при равных значениях `i` и `j` содержимое массива остается неизменным. Таким образом, использование подобного кода гарантирует, что измениться может только *порядок* значений в массиве, но не сам *набор значений*, которые в нем содержатся.

Случайная перестановка элементов массива

Следующий код переставляет значения из массива `deck[]` в случайном порядке:

```
int n = deck.length;
for (int i = 0; i < n; i++)
{
   int r = i + (int) (Math.random() * (n-i));
   String temp = deck[i];
   deck[i] = deck[r];
   deck[r] = temp;
}
```

Перебирая элементы слева направо, мы выбираем случайную карту из подмножества от `deck[i]` до `deck[n-1]` (с равной вероятностью) и меняем ее местами с `deck[i]`. Этот код сложнее, чем может показаться. Во-первых, идиома перестановки двух значений гарантирует, что состав карт в колоде после перестановки останется неизменным. Во-вторых, равномерный случайный выбор среди карт, которые еще не выбирались, гарантирует, что перестановка будет случайной.

Выборка без повторений

Достаточно часто возникает задача формирования случайной выборки таким образом, чтобы каждый элемент исходного множества встречался в выборке не более одного раза. Примером выборки такого рода служит вытаскивание шариков с номерами при розыгрыше лотереи или раздача «руки» из колоды карт. Программа `Sample` (листинг 1.4.1) демонстрирует формирование выборки с применением операции, лежащей в основе случайной перестановки из предыдущего раздела. Программа получает два аргумента командной строки `m` и `n` и создает случайную перестановку длины `n` (для множества целых чисел от 0 до `n-1`); первые `m` элементов образуют случайную выборку. Трассировка содержимого массива `perm[]` в конце каждой итерации основного цикла (для запуска программы со значениями `m` и `n`, равными 6 и 16 соответственно) демонстрирует этот процесс.

i	*r*	*perm[]*															
		0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	9	9	1	2	3	4	5	6	7	8	0	10	11	12	13	14	15
1	5	9	5	2	3	4	1	6	7	8	0	10	11	12	13	14	15
2	13	9	5	13	3	4	1	6	7	8	0	10	11	12	2	14	15
3	5	9	5	13	1	4	3	6	7	8	0	10	11	12	2	14	15
4	11	9	5	13	1	11	3	6	7	8	0	10	4	12	2	14	15
5	8	9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15
		9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15

Трассировка для команды java Sample 6 16

Если значения `r` выбираются так, что все значения в заданном диапазоне равновероятны, то в конце этого процесса элементы массива с `perm[0]` по `perm[m-1]` образуют равномерно распределенную выборку (при том что некоторые значения могут перемещаться многократно), потому что каждому элементу выборки присваивается значение, выбранное равновероятно из значений, которые в выборку еще не включались. Одна из ситуаций, когда может потребоваться такая выборка, — использование ее элементов как индексов для случайной выборки из *произвольного* массива. Часто такое решение лучше перестановки самих элементов массива, потому что исходный порядок элементов нужно по какой-то причине сохранить (допустим, компания хочет получить случайную выборку клиентов из списка, хранящегося в алфавитном порядке).[1]

[1] Второй полезный эффект такого подхода — его экономичность: в манипуляциях участвуют только индексы, а не сами элементы данных, размер которых может быть намного больше. Аналогично работают индексы в базах данных. — *Примеч. науч. ред.*

Листинг 1.4.1. Случайная выборка без замены

```
public class Sample
{
   public static void main(String[] args)
   {  // Вывод случайной выборки из m целых чисел
      // от 0 до n-1 (без повторений).
      int m = Integer.parseInt(args[0]);
      int n = Integer.parseInt(args[1]);
      int[] perm = new int[n];

      // Инициализация perm[].
      for (int j = 0; j < n; j++)
         perm[j] = j;

      // Получение выборки.
      for (int i = 0; i < m; i++)
      {  // perm[i] меняется местами со случайным элементом справа.
         int r = i + (int) (Math.random() * (n-i));
         int t = perm[r];
         perm[r] = perm[i];
         perm[i] = t;
      }

      // Вывод выборки.
      for (int i = 0; i < m; i++)
         System.out.print(perm[i] + " ");
      System.out.println();
   }
}
```

m	размер выборки
n	диапазон
perm[]	перестановка диапазона от 0 до n – 1

Программа получает как аргументы командной строки два числа m и n и строит выборку из m целых чисел из диапазона от 0 до n – 1. Этот процесс может пригодиться не только в лотереях, но и во многих научных приложениях. Если первый аргумент равен второму, то результат представляет собой случайную перестановку целочисленного диапазона от 0 до n – 1. Если первый аргумент больше второго, программа завершается с исключением ArrayOutOfBoundsException.

```
% java Sample 6 16
9 5 13 1 11 8
% java Sample 10 1000
656 488 298 534 811 97 813 156 424 109
% java Sample 20 20
6 12 9 8 13 19 0 2 4 5 18 1 14 16 17 3 7 11 10 15
```

Чтобы понять, как работает этот прием, представьте, что мы хотим набрать случайный расклад карт из массива `deck[]`, построенного только что описанным способом. Мы используем код из программы `Sample` с `n = 52` и `m = 5` и заменяем `perm[i]` на `deck[perm[i]]` в вызове `System.out.print()` (и заменяем его на `println()`). В итоге будет выведен следующий результат:

```
3 of Clubs
Jack of Hearts
6 of Spades
Ace of Clubs
10 of Diamonds
```

Подобные выборки широко используются на практике; они закладывают основу для статистических исследований в опросах, научных проектах и многих других приложениях — в любых ситуациях, в которых мы хотим прийти к неким заключениям относительно большого набора данных по результатам анализа малой случайной выборки.

Предварительно вычисленные значения

Массивы часто используются для временного хранения значений, вычисленных для последующего использования. Например, представьте, что вы пишете программу для вычислений с гармоническими числами (см. листинг 1.3.5). Эффективное решение основано на сохранении вычисленных данных в массиве:

```
double[] harmonic = new double[n];
for (int i = 1; i < n; i++)
   harmonic[i] = harmonic[i-1] + 1.0/i;
```

В дальнейшем для получения `i`-го гармонического числа достаточно обратиться к `harmonic[i]`. Подобные предварительные вычисления — типичный пример компромисса между затратами памяти и времени: расходуя память (для сохранения значений), мы экономим время (потому что их не приходится вычислять заново). Этот метод особенно эффективен, если вы собираетесь многократно использовать предварительно вычисленные значения для относительно малых `n`.

Упрощение повторяющегося кода

Рассмотрим еще один пример простого применения массивов: следующий фрагмент кода выводит название месяца по номеру (1 — январь, 2 — февраль и т. д.):

```
if      (m ==  1) System.out.println("Jan");
else if (m ==  2) System.out.println("Feb");
else if (m ==  3) System.out.println("Mar");
else if (m ==  4) System.out.println("Apr");
else if (m ==  5) System.out.println("May");
else if (m ==  6) System.out.println("Jun");
else if (m ==  7) System.out.println("Jul");
else if (m ==  8) System.out.println("Aug");
else if (m ==  9) System.out.println("Sep");
else if (m == 10) System.out.println("Oct");
else if (m == 11) System.out.println("Nov");
else if (m == 12) System.out.println("Dec");
```

В такой ситуации также можно использовать команду `switch`, но существует другое, гораздо более компактное решение — использовать массив строк с названиями месяцев:

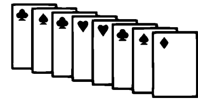
```
String[] MONTHS =
{
   "", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
       "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
System.out.println(MONTHS[m]);
```

Этот прием особенно эффективен для получения названий месяцев по номеру многократно, в разных местах программы. Обратите внимание: мы сознательно оставляем незаполненным один элемент массива (элемент 0), чтобы элемент MONTHS[1] соответствовал январю, как требуется по условиям задачи.

Разобравшись с базовыми определениями и примерами, теперь мы можем рассмотреть пару примеров, которые решают интересные классические задачи и демонстрируют фундаментальную роль массивов для эффективной организации вычислений. В обоих случаях идея использования выражений в качестве индексов массивов очень важна — она позволяет организовать вычисления, которые в других случаях были бы невозможны.

Собиратель купонов

Допустим, имеется колода карт и вы случайным образом переворачиваете карты одну за другой. Сколько карт нужно перевернуть, чтобы была открыта хотя бы одна карта каждой масти? Сколько карт нужно перевернуть, чтобы каждый номинал был открыт хотя бы один раз? Эти вопросы — частные случаи знаменитой задачи о собирателе купонов. В более общей постановке задачи представьте, что выпускается n разновидностей карт («купонов»). Считая, что карты каждого вида встречаются равновероятно, сколько карт необходимо получить, чтобы в наборе оказались все n разновидностей?



Набор карт («купонов»)

Задача о собирателе купонов вовсе не искусственна. Например, ученым часто бывает нужно узнать, обладает ли последовательность, встречающаяся в реальном мире, такими же характеристиками, как и случайная последовательность. Если так, этот факт может представлять интерес, а если нет, возможно, потребуются дополнительные исследования для выявления важных закономерностей. Например, такие проверки используются учеными для принятия решения о том, какие части геномов заслуживают изучения. Для проверки того, является ли последовательность действительно случайной, эффективно используется тест собирателя купонов: количество элементов, которые необходимо проверить для обнаружения всех значений, сравнивается с соответствующим числом для равномерно распределенной последовательности. Программа CouponCollector (листинг 1.4.2) моделирует этот процесс и демонстрирует пользу массивов. Она получает аргумент командной строки n и генерирует последовательность случайных целых чисел от 0 до n-1 с использованием конструкции (int) (Math.random() * n) — см. листинг 1.2.5. Каждое целое число изображает карточку: для каждой карточки нужно определить,

встречалась ли она ранее. Для хранения этой информации используется массив isCollected[], в котором карточка используется в качестве индекса; значение isCollected[i] равно true, если карта i уже встречалась, и равно false в противном случае. Получая новую карточку, представленную целым числом r, мы проверяем, встречалась ли она ранее, обращаясь к элементу isCollected[r]. Вычисления состоят из подсчета разных встречавшихся карточек и количества проверенных карточек и вывода последнего, когда первое достигает n.

r	*isCollected[]*						*distinct*	*count*
	0	*1*	*2*	*3*	*4*	*5*		
	F	F	F	F	F	F	0	0
2	F	F	T	F	F	F	1	1
0	T	F	T	F	F	F	2	2
4	T	F	T	F	T	F	3	3
0	T	F	T	F	T	F	3	4
1	T	T	T	F	T	F	4	5
2	T	T	T	F	T	F	4	6
5	T	T	T	F	T	T	5	7
0	T	T	T	F	T	T	5	8
1	T	T	T	F	T	T	5	9
3	T	T	T	T	T	T	6	10

Трассировка для типичного выполнения команды *java CouponCollector 6*

Как обычно, лучший способ понять логику программы — рассмотреть трассировку значений ее переменных для типичного запуска. Мы можем легко добавить в CouponCollector код для трассировки, сообщающий значения переменных в конце цикла while. В таблице выше для удобства значение false обозначено буквой F, а значение true — буквой T. Трассировка программ, использующих большие массивы, может создать проблемы: если в вашей программе используется массив длины n, он содержит n элеметов, и все их нужно вывести. Трассировка программ с Math.random() тоже создает проблемы, потому что вы получаете разные значения при каждом запуске. Следовательно, мы должны тщательно проверить взаимосвязь между переменными. Заметьте, что значение distinct всегда равно количеству значений true в isCollected[].

Без массивов мы бы не смогли смоделировать процесс сбора купонов для больших n; с массивами это совсем несложно. В книге вы встретите еще немало примеров такого рода.

Листинг 1.4.2. Моделирование задачи о собирателе купонов

```
public class CouponCollector
{
   public static void main(String[] args)
   {
      // Генерирование случайных значений [0..n) до того,
      // как будет найдено каждое значение.
      int n = Integer.parseInt(args[0]);
      boolean[] isCollected = new boolean[n];
      int count = 0;
      int distinct = 0;

      while (distinct < n)
      {
         // Генерирование очередного купона.
         int r = (int) (Math.random() * n);
         count++;
         if (!isCollected[r])
         {
            distinct++;
            isCollected[r] = true;
         }
      } // Найдено n разных купонов.

      System.out.println(count);
   }
}
```

n	значения купонов (от 0 до n – 1)
isCollected[i]	купон i был найден?
count	количество купонов
distinct	количество разных купонов
r	случайный купон

Программа получает целое число n как аргумент командной строки и моделирует процесс собирания купонов, генерируя случайные числа от 0 до n – 1, пока не будут получены все возможные значения.

```
% java CouponCollector 1000
6583
% java CouponCollector 1000
6477
% java CouponCollector 1000000
12782673
```

Решето Эратосфена

Простые числа играют важную роль в математике и вычислительных дисциплинах, включая криптографию. *Простым числом* называется целое число, большее 1 и имеющее только 2 натуральных делителя: единицу и само себя. Функция распределения простых чисел $\pi(n)$ возвращает количество простых чисел, меньших или равных n. Например, $\pi(25) = 9$, потому что первые девять простых чисел — 2, 3, 5, 7, 11, 13, 17, 19 и 23. Эта функция играет центральную роль в теории чисел.

Один из способов подсчета простых чисел основан на использовании таких программ, как `Factors` (листинг 1.3.9). А точнее, мы можем изменить код `Factors` для присваивания переменной `boolean` значения `true`, если заданное число простое, или `false` в противном случае (вместо вывода множителей), а затем заключить этот код в цикл с увеличением счетчика для каждого простого числа. Этот алгоритм эффективен для малых `n`, но очень сильно замедляется с ростом `n`. Программа `PrimeSieve` (листинг 1.4.3) получает целое число `n` как аргумент командной строки и вычисляет распределение целых чисел с использованием алгоритма, называемого *решетом Эратосфена*. Программа использует логический массив `isPrime[]` для хранения информации о том, какие целые числа являются простыми. Если целое число `i` является простым, `isPrime[i]` содержит `true`, а если нет — `false`. Решето Эратосфена работает следующим образом: изначально всем элементам массива присваивается `true`; это означает, что ни один множитель еще не был найден. Затем следующие шаги повторяются, пока выполняется условие `i <= n/i`.

- Найти следующее наименьшее целое число `i`, для которого еще не найдены множители.
- Оставить `isPrime[i]` значение `true`, так как `i` не раскладывается на меньшие множители.
- Присвоить значение `false` всем элементам `isPrime[]` с индексами, кратными `i`.

После завершения вложенного цикла `for` элемент `isPrime[i]` равен `true` в том и только в том случае, если целое число `i` — простое.

Еще один проход по массиву подсчитывает количество простых чисел, меньших или равных `n`.

Как обычно, вы можете легко добавить в программу код трассировки. Для таких программ, как `PrimeSieve`, нужно действовать осторожно — она содержит вложенную конструкцию `for-if-for`, и вы должны следить за расположением фигурных скобок, чтобы правильно разместить код вывода. Цикл завершается по условию `i > n/i`, как и в программе `Factors`.

i	isPrime[]																							
	2	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*
	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
3	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
5	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F
	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F

Трассировка для команды java PrimeSieve 25

Листинг 1.4.3. Решето Эратосфена

```
public class PrimeSieve
{
   public static void main(String[] args)
   {  // Вывод количества простых чисел <= n.
      int n = Integer.parseInt(args[0]);
      boolean[] isPrime = new boolean[n+1];
      for (int i = 2; i <= n; i++)
         isPrime[i] = true;
      for (int i = 2; i <= n/i; i++)
      {  if (isPrime[i])
         {  // Числа, кратные i, помечаются как непростые.
            for (int j = i; j <= n/i; j++)
               isPrime[i * j] = false;
         }
      }

      // Подсчет простых чисел.
      int primes = 0;
      for (int i = 2; i <= n; i++)
         if (isPrime[i]) primes++;
      System.out.println(primes);
   }
}
```

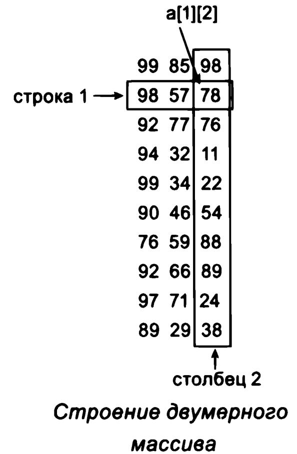
`n`	аргумент
`isPrime[i]`	i простое число?
`primes`	счетчик простых чисел

Программа получает целое число n как аргумент командной строки и вычисляет количество простых чисел, меньших или равных n. Для этого программа присваивает элементам логического массива isPrime[i] значение true, если число i простое, и false в противном случае. Сначала всем элементам массива присваивается значение true; это означает, что изначально среди чисел нет ни одного, которое бы заведомо не было простым. Затем значение false присваивается элементам массива, индексы которых не являются простыми (то есть кратны известным простым числам). Если a[i] сохраняет значение true после того, как всем числам, кратным меньшим простым числам, было присвоено значение false, из этого следует, что i является простым числом. Во втором цикле for используется условие завершения i <= n/i вместо наивного i <= n, потому что любое число, не имеющее множителей, меньших n/i, не имеет множителей, больших n/i, и такие множители рассматривать не нужно. Это усовершенствование позволяет эффективнее выполнять программу для больших n.

```
% java PrimeSieve 25
9
% java PrimeSieve 100
25
% java PrimeSieve 1000000000
50847534
```

С программой `PrimeSieve` можно вычислить $\pi(n)$ для больших n; ограничения определяются прежде всего максимальной длиной массива, поддерживаемой Java. Это еще один пример компромисса между затратами памяти и времени. Такие программы, как `PrimeSieve`, помогают математикам развивать теорию чисел, которая находит много важных практических применений.

Двумерные массивы



Строение двумерного массива

Во многих приложениях для хранения информации удобно использовать числовую таблицу прямоугольной формы и указывать номера строк и столбцов таблицы. Например, преподаватель может создать таблицу, строки которой соответствуют студентам, а столбцы — экзаменам; ученый может создать таблицу экспериментальных данных, строки которой соответствуют экспериментам, а столбцы — различным результатам; или программист может подготовить изображение для вывода на экран, заполняя таблицу пикселов различными цветами или градациями серого.

В среде математических абстракций аналогом таких таблиц является *матрица*, а из конструкций Java — *двумерный массив*. Скорее всего, вы уже сталкивались с примерами использования матриц и двумерных массивов и наверняка еще неоднократно встретитесь с ними в науке, технике и обработке данных. Как и в случае с векторами и одномерными массивами, многие применения сопряжены с обработкой больших объемов данных, и рассмотрение таких приложений откладывается до знакомства с механизмами ввода и вывода (раздел 1.5).

Расширение конструкций массивов Java для обработки двумерных массивов достаточно прямолинейно. Для обращения к элементу, находящемуся на пересечении строки `i` и столбца `j` двумерного массива `a[][]`, используется запись `a[i][j]`; для объявления двумерного массива добавляется еще одна пара квадратных скобок; и для создания массива указывается имя типа, за которым следует количество строк и количество столбцов (и то и другое заключается в квадратные скобки):

```
double[][] a = new double[m][n];
```

Такие массивы называются *массивами* `m` × `n`. По действующим соглашениям в первых скобках указывается количество строк, а во втором — количество столбцов. Как и в случае с одномерными массивами, Java по умолчанию инициализирует все элементы числовых массивов нулями, а элементы массивов `boolean` — значением `false`.

Инициализация по умолчанию

Инициализация по умолчанию для двумерных массивов полезна, потому что за ней скрывается больший объем кода, чем для одномерных массивов. Следующий код эквивалентен однострочной идиоме «создания/инициализации», описанной в предыдущем разделе:

```
double[][] a;
a = new double[m][n];
for (int i = 0; i < m; i++)
{  // Инициализация i-й строки.
```

```
   for (int j = 0; j < n; j++)
      a[i][j] = 0.0;
}
```

Для инициализации элементов двумерного массива нулями этот код избыточен, но для инициализации элементов другим(-и) значением(-ями) необходимы вложенные циклы `for`. Как вы увидите, по этому образцу пишется код, используемый для обращения к каждому элементу двумерного массива или его изменения.

Вывод

Вложенные циклы `for` используются во многих операциях обработки двумерных массивов. Например, для вывода массива `m` × `n` в табличном формате используется следующий код:

```
for (int i = 0; i < m; i++)
{  // Вывод i-й строки.
   for (int j = 0; j < n; j++)
      System.out.print(a[i][j] + " ");
   System.out.println();
}
```

При желании можно включить в выходные данные индексы строк и столбцов (см. упражнение 1.4.6). Программисты Java обычно считают, что в двумерных массивах индексы строк увеличиваются сверху вниз, начиная с 0, а индексы столбцов увеличиваются слева направо и тоже начинаются с 0.

Представление в памяти

В Java двумерный массив представляется как массив массивов. Другими словами, двумерный массив из `m` строк и `n` столбцов в действительности представляет собой массив длины `m`, каждый элемент которого является одномерным массивом длины `n`. В двумерном массиве Java `a[][]` конструкция `a[i]` может использоваться для обозначения строки `i` (которая представляет собой одномерный массив), но аналогичного способа для обозначения столбца `j` не предусмотрено.

a[][]

	a[0][0]	a[0][1]	a[0][2]
	a[1][0]	a[1][1]	a[1][2]
	a[2][0]	a[2][1]	a[2][2]
	a[3][0]	a[3][1]	a[3][2]
	a[4][0]	a[4][1]	a[4][2]
a[5] →	a[5][0]	a[5][1]	a[5][2]
	a[6][0]	a[6][1]	a[6][2]
	a[7][0]	a[7][1]	a[7][2]
	a[8][0]	a[8][1]	a[8][2]
	a[9][0]	a[9][1]	a[9][2]

Массив 10 × 3

Присваивание значений во время компиляции

Метод инициализации значений массивов следует непосредственно из представления. Двумерный массив является массивом строк, каждая из которых инициализируется как одномерный массив. Чтобы инициализировать двумерный массив, мы заключаем в фигурные скобки список инициализаторов

строк, разделенных запятыми. Каждый элемент списка сам по себе является списком: он состоит из значений элементов массива, входящих в строку, которые заключаются в фигурные скобки и разделяются запятыми.

```
double[][] a =
{
   { 99.0, 85.0, 98.0,  0.0 },
   { 98.0, 57.0, 79.0,  0.0 },
   { 92.0, 77.0, 74.0,  0.0 },
   { 94.0, 62.0, 81.0,  0.0 },
   { 99.0, 94.0, 92.0,  0.0 },
   { 80.0, 76.5, 67.0,  0.0 },
   { 76.0, 58.5, 90.5,  0.0 },
   { 92.0, 66.0, 91.0,  0.0 },
   { 97.0, 70.5, 66.5,  0.0 },
   { 89.0, 89.5, 81.0,  0.0 },
   {  0.0,  0.0,  0.0,  0.0 }
};
```

Инициализация массива 10×3 с элементами `double` *на стадии компиляции*

Электронные таблицы

Одно из типичных применений двумерных массивов — электронная таблица с числовыми данными. Например, преподаватель с `m` студентами и `n` оценками контрольных для каждого студента может создать массив $(m + 1) \times (n + 1)$, зарезервировав последний столбец для средней оценки каждого студента и последнюю строку — для средних оценок по каждой контрольной. И хотя обычно такие вычис-

Вычисление средних значений строк

```
for (int i = 0; i < m; i++)
{  // Вычисление среднего значения
      по строке i
   double sum = 0.0;
   for (int j = 0; j < n; j++)
      sum += a[i][j];
   a[i][n] = sum / n;
}
```

Вычисление средних значений столбцов

```
for (int j = 0; j < n; j++)
{  // Вычисление среднего значения
      по столбцу j
   double sum = 0.0;
   for (int i = 0; i < m; i++)
      sum += a[i][j];
   a[m][j] = sum / m;
}
```

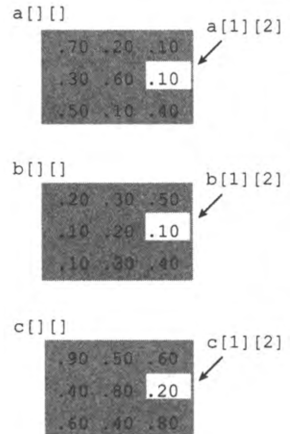
Типичные вычисления для электронных таблиц

ления проводятся в специализированных приложениях, полезно проанализировать используемый код для лучшего понимания обработки массивов. Чтобы вычислить среднюю оценку для каждого студента (средние значения по строкам), следует просуммировать элементы каждой строки и разделить сумму на n. Аналогичным образом для вычисления средней оценки по каждой контрольной (среднее значение по каждому столбцу) следует просуммировать элементы каждого столбца и разделить результат на m.

Операции с матрицами

В типичных приложениях из области науки и техники матрицы представляются двумерными массивами, а затем в программе реализуются различные математические операции с матричными операндами. И хотя такая обработка данных часто встречается в специализированных приложениях, вам будет полезно понимать суть задействованных в ней вычислений. Например, *суммирование* двух матриц $n \times n$ выполняется следующим образом:

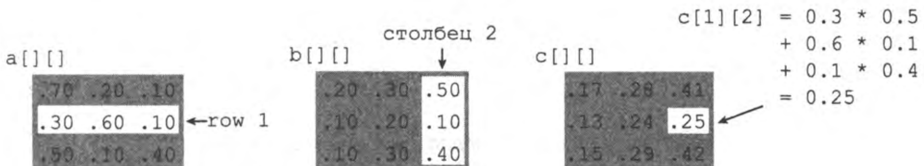
```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      c[i][j] = a[i][j] + b[i][j];
```



Суммирование матриц

Аналогично, матрицы можно *умножать*. Возможно, вы уже изучали эту операцию, но если успели забыть — приведенный ниже код Java для вычисления произведения двух квадратных матриц фактически повторяет свое математическое определение. Каждый элемент `c[i][j]` в произведении `a[][]` и `b[][]` вычисляется как скалярное произведение строки `i` матрицы `a[][]` на столбец `j` матрицы `b[][]`.

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++)
{
   for (int j = 0; j < n; j++)
   {
      // Скалярное произведение строки i и столбца j.
      for (int k = 0; k < n; k++)
         c[i][j] += a[i][k]*b[k][j];
   }
}
```



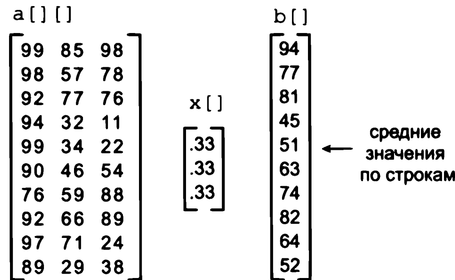
Умножение матриц

Особые случаи матричного умножения

Существуют два важных особых случая матричного умножения. Они возникают тогда, когда одно измерение одной из матриц равно 1, так что матрица может рассматриваться как вектор. В таком случае происходит *матрично-векторное умножение*, при котором матрица $m \times n$ умножается на вектор-столбец (матрица $n \times 1$) для получения результирующего вектора $m \times 1$ (каждый элемент результата равен скалярному произведению соответствующей строки матрицы и векторного опе-

Матрично-векторное умножение `a[][]*x[] = b[]`

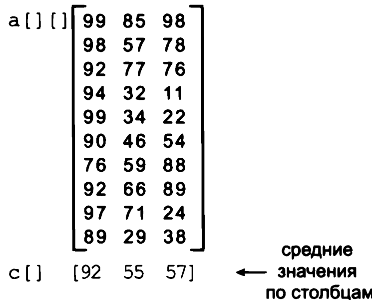
```
for (int i = 0; i < m; i++)
{  // Скалярное произведение строки i и x[].
   for (int j = 0; j < n; j++)
      b[i] += a[i][j]*x[j];
}
```



Векторно-матричное умножение `y[]*a[][] = c[]`

```
for (int j = 0; j < n; j++)
{  // Скалярное произведение y[] и столбца j.
   for (int i = 0; i < m; i++)
      c[j] += y[i]*a[i][j];
}
```

y[] [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1]



Матрично-векторное и векторно-матричное умножение

ранда.) Во втором случае — при *векторно-матричном умножении* — вектор-строка (матрица $1 \times m$) умножается на матрицу $m \times n$ для получения результирующего вектора $1 \times n$ (каждый элемент результата равен скалярному произведению векторного операнда и соответствующего столбца матрицы). Такие операции предоставляют компактную запись для выражения разнообразных матричных вычислений. Например, вычисление средних значений по строкам для электронной таблицы из m строк и n столбцов эквивалентно матрично-векторному умножению, при котором вектор содержит n элементов, каждый из которых равен $1/n$. Аналогичным образом вычисление средних значений по столбцам в такой таблице эквивалентно векторно-матричному умножению, при котором вектор содержит m элементов, каждый из которых равен $1/m$. Мы вернемся к векторно-матричному умножению в контексте приложения, рассмотренного в конце этой главы.

Ступенчатые массивы

Строки двумерного массива не обязаны иметь одинаковую длину — массив со строками разной длины называется *ступенчатым массивом* (пример приведен в упражнении 1.4.34). Использование ступенчатых массивов требует большей аккуратности в написании кода, работающего с массивами. Например, следующий фрагмент выводит содержимое ступенчатого массива:

```
for (int i = 0; i < a.length; i++)
{
   for (int j = 0; j < a[i].length; j++)
      System.out.print(a[i][j] + " ");
   System.out.println();
}
```

Этот код проверяет ваше понимание работы с массивами в Java; не пожалейте времени на его изучение. В этой книге мы обычно используем квадратные или прямоугольные массивы, размеры которых задаются переменной `m` или `n`. Код, использующий `a[i].length` подобным образом, четко сообщает о том, что массив является ступенчатым (невыровненным).

Многомерные массивы

Синтаксис двумерных массивов расширяется для массивов с любым количеством измерений. Например, трехмерный массив можно объявить и инициализировать следующим образом:

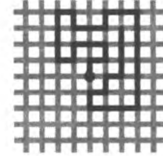
```
double[][][] a = new double[n][n][n];
```

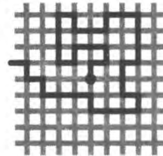
В дальнейшем вы сможете обращаться к элементам в коде вида `a[i][j][k]` и т. д.

Двумерные массивы обеспечивают естественное представление матриц, повсеместно встречающихся в науке, математике и технике. Также они представляют собой естественный способ организации больших объемов данных — ключевого компонента электронных таблиц и многих других вычислительных приложений. А при использовании декартовых координат двумерные и трехмерные массивы по-

зволяют также моделировать объекты реального мира. В книге будет рассмотрено их применение во всех трех областях.

Пример: случайные блуждания без самопересечений





Случайные блуждания без самопересечений

Представьте собаку, оказавшуюся посреди большого города. Будем считать, что существует *n* улиц, проходящих с севера на юг, и *n* улиц, проходящих с запада на восток; все улицы разделены одинаковыми интервалами и пересекаются во всех точках, образуя прямоугольную *сетку*. Пытаясь покинуть город, собака на каждом перекрестке выбирает случайное направление, но при этом ее чутье заставляет ее избегать перехода к ранее посещенным местам. Однако собака может зайти в тупик, из которого ей неизбежно придется вернуться к какому-либо пересечению. Какова вероятность того, что это случится? Эта занимательная задача — простой пример знаменитой модели *случайных блужданий без самопересечений*, которая нашла важное научное применение в изучении полимеров, статистической механике и ряде других областей. Например, этот процесс может моделировать выращивание цепочек из некоторого вещества до того момента, когда дальнейший рост становится невозможным. Чтобы лучше понимать такие процессы, ученые изучают свойства случайных блужданий без самопересечений.

Вероятность выхода собаки за пределы города очевидно зависит от размера города. В крошечном городе 5 × 5[1] легко убедиться в том, что собака обязательно покинет город. Но какова вероятность этого в большом городе? Также нас интересуют другие параметры: например, какова средняя длина пути собаки? Как часто собака будет находиться в одном квартале от границы? Такие свойства играют важную роль в только что упоминавшихся приложениях.

Программа `SelfAvoidingWalk` (листинг 1.4.4) моделирует ситуацию с использованием двумерного массива типа `boolean`. Каждый элемент массива представляет пересечение улиц. Значение `true` означает, что собака уже посещала это пересечение; значение `false` — что пересечение еще не посещалось. Путь начинается в центре, и собака перемещается в случайном направлении к пересечениям, которые еще не посещались, пока не зайдет в тупик или не выйдет за границу. Для простоты код написан так, что если случайный выбор должен привести к уже посещавшемуся

[1] Исходный текст программы (см. ниже) позволяет понять, что в выбранной модели крайние «улицы» проходят по периметру «города» и достижение их означает выход из «города»: при попадание в крайние ячейки матрицы цикл завершается. Эти крайние улицы на иллюстрациях *не изображены*. Например, рис. на с. 128 соответствует «городу» размером 11×11, но на нем изображены только по 9 «улиц» каждого направления. Аналогично, на рис. с. 130 — «город» 21×21 и 19+19 изображенных «улиц». — *Примеч. науч. ред.*

Листинг 1.4.4. Случайные блуждания без самопересечений

```
public class SelfAvoidingWalk
{
   public static void main(String[] args)
   {  // Моделирование случайных блужданий
      // без самопересечений в сетке nxn.
      int n = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);
      int deadEnds = 0;
      for (int t = 0; t < trials; t++)
      {
         boolean[][] a = new boolean[n][n];
         int x = n/2, y = n/2;
         while (x > 0 && x < n-1 && y > 0 && y < n-1)
         {  // Проверка тупика и случайное перемещение.
            a[x][y] = true;
            if (a[x-1][y] && a[x+1][y] && a[x][y-1] && a[x][y+1])
            {  deadEnds++; break;  }
            double r = Math.random();
            if      (r < 0.25) { if (!a[x+1][y]) x++; }
            else if (r < 0.50) { if (!a[x-1][y]) x--; }
            else if (r < 0.75) { if (!a[x][y+1]) y++; }
            else if (r < 1.00) { if (!a[x][y-1]) y--; }
         }
      }
      System.out.println(100*deadEnds/trials + "% dead ends");
   }
}
```

`n`	размер сетки
`trials`	количество экспериментов
`deadEnds`	количество экспериментов, закончившихся тупиком
`a[][]`	количество посещенных пересечений
`x, y`	текущая позиция
`r`	случайное число в диапазоне (0, 1)

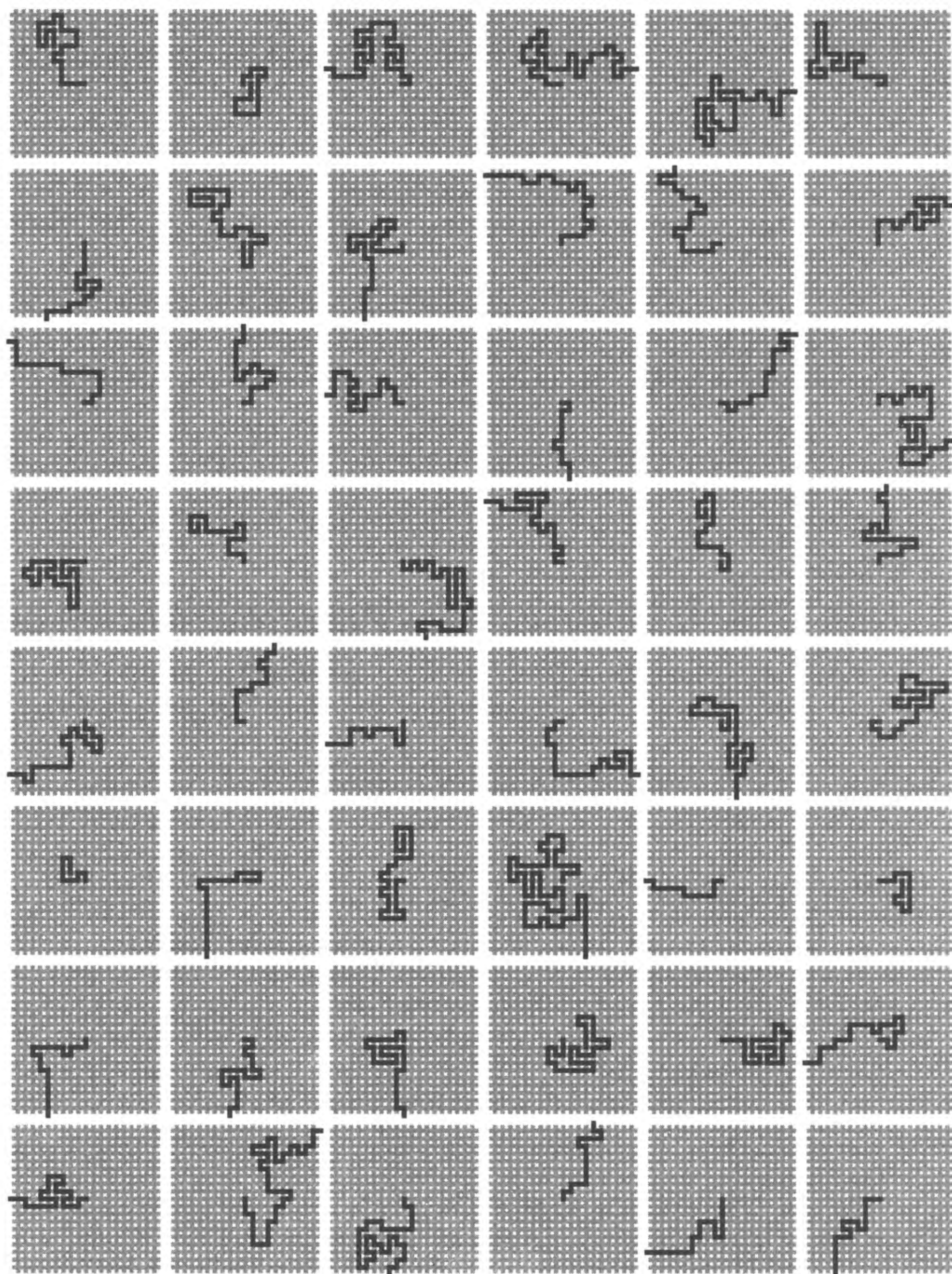
Программа получает аргументы командной строки n и trials и моделирует случайные блуждания без самопересечений на сетке n × n. Для каждого блуждания программа создает массив типа boolean, начинает блуждание в центре и продолжает двигаться до тех пор, пока не окажется в тупике или не достигнет границы. Результатом вычисления является процент блужданий, закончившихся тупиком. Увеличение количества экспериментов повышает точность оценки.

```
% java SelfAvoidingWalk 5 100
0% dead ends
% java SelfAvoidingWalk 20 100
36% dead ends
% java SelfAvoidingWalk 40 100
80% dead ends
% java SelfAvoidingWalk 80 100
98% dead ends
```

```
% java SelfAvoidingWalk 5 1000
0% dead ends
% java SelfAvoidingWalk 20 1000
32% dead ends
% java SelfAvoidingWalk 40 1000
70% dead ends
% java SelfAvoidingWalk 80 1000
95% dead ends
```

пересечению, действие не выполняется; предполагается, что последующий случайный выбор найдет другой вариант (который заведомо существует, потому что программа явно проверяет тупик и завершает цикл в этом случае).

Обратите внимание: работа кода зависит от инициализации всех элементов массива значениями `false` при каждом эксперименте. Также в коде продемонстрирован



Случайные блуждания без самопересечений в сетке 21 х 21

один важный прием: условие продолжения цикла является также и защитой от выполнения недопустимых действий в теле цикла. В данном случае условие продолжения цикла `while` защищает от выхода за границу массива внутри цикла (проверка выхода собаки за границу города). Успешная проверка попадания в тупик в теле цикла приводит к его завершению.

Как видно из примеров на предыдущей странице, в большом городе собака почти наверняка окажется в тупике. Если вам захочется больше узнать о случайных блужданиях, некоторые рекомендации приводятся в упражнениях. Например, в трехмерной версии задачи собака почти наверняка убежит из города. И хотя этот результат интуитивно понятен и подтвержден нашими экспериментами, разработка математической модели, объясняющей поведение случайных блужданий без самопересечений, относится к числу известных нерешенных задач: несмотря на обширные исследования, никому пока не удалось найти компактное математическое выражение для вероятности выхода, средней длины пути или любого другого важного параметра.

Выводы

Массивы — четвертый базовый элемент (после присваивания, условных конструкций и циклов), присутствующий практически в каждом языке программирования. На этом наше знакомство с базовыми конструкциями Java завершается. Как было показано в примерах программ, этих конструкций достаточно для написания самых разнообразных программ.

Массивы играют важную роль во многих программах, и базовые операции, описанные в тексте, пригодятся при решении очень многих задач из области программирования. Даже когда вы не используете массивы явно, при этом они все равно неявно присутствуют, потому что память любого компьютера на концептуальном уровне эквивалентна массиву.

Массивы дополняют наши программы одним фундаментальным качеством: потенциально резким возрастанием размерности *состояния* программы. *Состояние программы* можно определить как совокупность всей информации, необходимой для понимания ее работы. Чтобы определить, что далее произойдет в программе без массивов, обычно достаточно знать значения переменных и следующую выполняемую команду. При выводе трассировки программы вы фактически отслеживаете ее состояние. Но если в программе используются массивы, количество значений (которые к тому же могут изменяться с каждой операцией) может быть слишком велико, чтобы их можно было отслеживать. Из-за этого различия программы с массивами сложнее для написания, чем программы без массивов.

Массивы являются естественной формой представления векторов и матриц, поэтому они постоянно используются в вычислениях, связанных со многими базовыми

задачами в науке и техники. Массивы также позволяют компактно реализовать обработку значительных объемов данных, выполняемую по единой схеме; естественно, они играют важную роль во многих приложениях, связанных с обработкой больших объемов данных.

Вопросы и ответы

В. Некоторые программисты Java используют для объявления массивов конструкцию `int a[]` вместо `int[] a`. Чем они отличаются?

О. В Java обе конструкции допустимы и фактически эквивалентны. Первая соответствует способу объявления массивов в языке С. Вторая считается предпочтительной в Java, потому что тип переменной `int[]` более четко выражает, что создается массив целых чисел.

В. Почему индексы массивов начинаются с 0, а не с 1?

О. Эта традиция пришла из программирования на машинных языках, в которых адрес элемента массива вычислялся суммированием индекса с адресом начала массива. Если бы индексы начинались с 1, это бы привело или к непроизводительной потере памяти в начале массива, или к потере времени на вычитание 1.

В. Что произойдет, если использовать для индексирования массива отрицательное число?

О. То же, что при использовании слишком большого индекса. Каждый раз, когда программа пытается обратиться к элементу с индексом, выходящим за пределы диапазона от 0 до длины массива минус 1, Java инициирует исключение `ArrayIndexOutOfBoundsException`.

В. Должны ли все элементы инициализатора массива быть литералами (константами)?

О. Нет. Элементы инициализатора массива могут быть произвольными выражениями (заданного типа), даже если их значения неизвестны во время компиляции. Например, следующий фрагмент инициализирует двумерный массив значением аргумента командной строки `theta`:

```
double theta = Double.parseDouble(args[0]);
double[][] rotation =
{
   { Math.cos(theta), -Math.sin(theta) },
   { Math.sin(theta),  Math.cos(theta) },
};
```

В. Массив символов чем-то отличается от `String`?

О. Да. Например, вы можете изменять отдельные символы массива `char[]`, а при работе со `String` такой возможности нет. Строки, включая методы для доступа к их содержимому, будут более подробно рассмотрены в разделе 3.1.

В. Что произойдет, если сравнить два массива выражением (a == b)?

О. Результат этого выражения равен true в том и только в том случае, если a[] и b[] обозначают один массив (адрес памяти), но не тогда, когда они хранят одинаковые последовательности значений. К сожалению, такое сравнение редко делает именно то, что вам нужно. Вместо этого вы можете сравнивать соответствующие элементы массивов в цикле.

В. Что произойдет при использовании массива в команде присваивания — например, a = b?

О. Команда присваивания заставит переменную a ссылаться на тот же массив, что и b, — она не копирует значения из массива b в массив a, как можно было бы ожидать. Например, возьмем следующий фрагмент кода:

```
int[] a = { 1, 2, 3, 4 };
int[] b = { 5, 6, 7, 8 };
a = b;
a[0] = 9;
```

После команды присваивания a = b элемент a[0] равен 5, элемент a[1] равен 6 и т. д., как и ожидалось. Таким образом, массивы соответствуют одной последовательности значений. Тем не менее они не являются независимыми массивами. Например, после выполнения последней команды равным 9 оказывается не только элемент a[0], но и элемент b[0]. Это одно из ключевых различий между примитивными типами (такими, как int и double) и непримитивными типами (такими, как массивы). Мы еще вернемся к этому тонкому (но принципиальному) различию, когда будем рассматривать передачу массивов функциям в разделе 2.2 и ссылочные типы в разделе 3.1.

В. Если a[] — массив, то почему вызов System.out.println(a) выводит нечто вроде @f62373 вместо последовательности значений в массиве?

О. Хороший вопрос. Выводится адрес массива в памяти (в виде шестнадцатеричного числа) — к сожалению, обычно это не то, что вам нужно.

В. Чего еще следует остерегаться при работе с массивами?

О. Очень важно помнить, что Java автоматически инициализирует массивы при создании, так что *создание массива занимает время, пропорциональное его длине.*

Упражнения

1.4.1. Напишите программу, которая объявляет, создает и инициализирует массив a[] длины 1000 и обращается к элементу a[1000]. Компилируется ли ваша программа? Что происходит при ее запуске?

1.4.2. Опишите и объясните, что произойдет при попытке откомпилировать программу со следующей командой:

```
int n = 1000;
int[] a = new int[n*n*n*n];
```

1.4.3. Для двух заданных векторов длины n, представленных одномерными массивами, напишите фрагмент кода для вычисления евклидова расстояния между ними (квадратный корень из суммы квадратов разностей между соответствующими элементами).

1.4.4. Напишите фрагмент кода, который переставляет в обратном порядке значения в одномерном строковом массиве. Не создавайте другой массив для хранения результатов. Подсказка: воспользуйтесь кодом, приведенным в тексте, для перестановки значений двух элементов.

1.4.5. Что не так со следующим фрагментом?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

1.4.6. Напишите фрагмент кода для вывода содержимого двумерного массива boolean. В выходных данных символ * должен представлять true, а пробел — false. Включите индексы строк и столбцов.

1.4.7. Какой результат выводит следующий фрагмент кода?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(a[i]);
```

1.4.8. Какие значения запишет в массив a[] следующий код?

```
int n = 10;
int[] a = new int[n];
a[0] = 1;
a[1] = 1;
for (int i = 2; i < n; i++)
   a[i] = a[i-1] + a[i-2];
```

1.4.9. Какой результат выведет следующий фрагмент кода?

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };
System.out.println(a == b);
```

1.4.10. Напишите программу Deal, которая получает целое число n как аргумент командной строки и выводит n покерных раздач (по пять карт каждая) из перетасованной колоды, разделенных пустыми строками.

1.4.11. Напишите программу HowMany, которая получает переменное количество аргументов командной строки и сообщает, сколько аргументов было передано.

1.4.12. Напишите программу `DiscreteDistribution`, которая получает переменное количество целочисленных аргументов командной строки и выводит целое число `i` с вероятностью, пропорциональной `i`-му аргументу командной строки.

1.4.13. Напишите фрагменты кода для создания двумерного массива `b[][]`, который является копией существующего двумерного массива `a[][]` при каждом из следующих предположений.

a. Массив `a[][]` — квадратный.

b. Массив `a[][]` — прямоугольный.

c. Массив `a[][]` может быть невыровненным (ступенчатым).

Ваше решение для пункта *b* должно работать также и для пункта *a*, а ваше решение для *c* должно работать и для пунктов *b* и *a*. Код при этом последовательно усложняется.

1.4.14. Напишите фрагмент кода для вывода транспонированного квадратного двумерного массива (то есть полученного перестановкой строк и столбцов). Для примера таблицы, приведенного в тексте, ваш код должен выводить следующий результат:

```
99  98  92  94  99  90  76  92  97  89
85  57  77  32  34  46  59  66  71  29
98  78  76  11  22  54  88  89  24  38
```

1.4.15. Напишите фрагмент кода для транспонирования квадратного двумерного массива «на месте» без создания второго массива.

1.4.16. Напишите программу, которая получает целое число `n` как аргумент командной строки и создает массив `n` × `n` `a[][]` с элементами `boolean`. Элемент `a[i][j]` равен `true`, если значения `i` и `j` являются взаимно простыми (не имеют общих множителей), или `false` в противном случае. Используйте свое решение к упражнению 1.4.6 для вывода массива. Подсказка: используйте решето Эратосфена.

1.4.17. Измените фрагмент кода электронной таблицы в тексте, чтобы вычислить *средневзвешенные* значения по строкам; веса оценок задаются одномерным массивом `weights[]`. Например, чтобы последний из трех экзаменов в нашем примере имел вдвое больший вес, чем первые два, используется массив

```
double[] weights = { 0.25, 0.25, 0.50 };
```

Обратите внимание: сумма весов должна быть равна 1.

1.4.18. Напишите фрагмент кода для умножения двух *прямоугольных* матриц, которые не обязаны быть квадратными. *Примечание*: чтобы скалярное произведение вычислялось корректно, количество столбцов в первой матрице должно совпадать с количеством строк во второй матрице. Если размеры матриц не удовлетворяют этому условию, выведите сообщение об ошибке.

1.4.19. Напишите программу для «логического умножения» двух квадратных матриц с элементами `boolean`. Используйте операции логического OR вместо + и логического AND вместо *.

1.4.20. Измените программу `SelfAvoidingWalk` (листинг 1.4.4), чтобы она вычисляла и выводила среднюю длину путей и вероятность попадания в тупик. Выводите отдельно средние длины путей с выходом и тупиковых путей.

1.4.21. Измените программу `SelfAvoidingWalk`, чтобы она вычисляла и выводила среднюю площадь наименьшего прямоугольника, ориентированного по координатным осям, вмещающего тупиковые пути.

Упражнения повышенной сложности

1.4.22. *Моделирование бросков кубиков.* Следующий код вычисляет плотность распределения вероятности для суммы двух кубиков:

```
int[] frequencies = new int[13];
for (int i = 1; i <= 6; i++)
   for (int j = 1; j <= 6; j++)
      frequencies[i+j]++;

double[] probabilities = new double[13];
for (int k = 1; k <= 12; k++)
   probabilities[k] = frequencies[k] / 36.0;
```

Значение `probabilities[k]` определяет вероятность того, что сумма очков на кубиках равна `k`. Проведите эксперименты, проверяющие эти вычисления, моделируя броски `n` кубиков и подсчитывая частоты каждого значения при вычислении суммы случайных целых чисел, равномерно распределенных в диапазоне от 1 до 6. Насколько большим должно быть значение `n`, чтобы ваши эмпирические результаты совпали с теоретичскими до трех знаков?

1.4.23. *Самая длинная возвышенность.* Для заданного массива целых чисел определите длину и местонахождение самой длинной непрерывной серии одинаковых значений, до и после которой располагаются элементы с меньшим значением.

1.4.24. *Эмпирическая проверка случайной перестановки.* Проведите вычислительные эксперименты, доказывающие, что код случайной перестановки («тасования» элементов) работает как задумано. Напишите программу `ShuffleTest`, которая получает два целых числа `m` и `n` в аргументах командной строки, выполняет `n` случайных перестановок в массиве длины `m`, инициализируемом значениями `a[i] = i` перед каждой перестановкой, и выводит такую таблицу `m` × `m`, что строка `i` содержит количество попаданий `i` в позицию `j` для всех `j`. Все значения в полученном массиве должны быть близки к `n / m`.

1.4.25. *Неправильная случайная перестановка.* Предположим, в коде случайной перестановки случайное число выбирается в диапазоне от 0 до `n-1` (вместо диапазона

от `i` до `n-1`). Покажите, что полученный порядок не является равновероятно одним из `n!` возможных. Проведите проверку из предыдущего упражнения в этой версии.

1.4.26. *Случайная перестановка музыки.* Вы переводите плеер в режим случайного воспроизведения. Он воспроизводит каждую из `n` песен, прежде чем повторяться. Напишите программу для оценки вероятности того, что две песни в соседних позициях списка не будут воспроизводиться подряд (то есть песня 3 не будет следовать за песней 2, песня 10 — за песней 9 и т. д.).

1.4.27. *Минимумы в перестановках.* Напишите программу, которая получает целое число `n` как аргумент командной строки, генерирует случайную перестановку, выводит перестановку и количество лево-правых минимумов в перестановке (количество раз, когда очередной элемент является наименьшим из встречавшихся до настоящего момента). Затем напишите программу, которая получает два целых числа `m` и `n` как аргументы командной строки, генерирует `m` случайных перестановок длины `n` и выводит среднее количество лево-правых минимумов в сгенерированных перестановках. Дополнительное задание: сформулируйте гипотезу относительно количества лево-правых минимумов в перестановке длины `n` как функции `n`.

1.4.28. *Обратная перестановка.* Напишите программу, которая получает перестановку целых чисел от 0 до `n-1` как аргументы командной строки и выводит обратную перестановку. (Если исходная перестановка хранится в массиве `a[]`, то обратной перестановкой называется массив `b[]`, для которого `a[b[i]] = b[a[i]] = i`.) Обязательно проверьте, что входные данные действительно образуют перестановку.

1.4.29. *Матрица Адамара.* Матрица Адамара $n \times n$ $H(n)$ представляет собой матрицу с элементами `boolean`, обладающую особым свойством: в любых ее двух строках различаются ровно $n/2$ значений. (Благодаря этому свойству она особенно полезна для проектирования кодов исправления ошибок.) $H(1)$ — матрица 1×1 с единственным элементом, равным `true`, а для любых `n > 1` матрица $H(2n)$ вычисляется размещением четырех копий $H(n)$ в квадрате удвоенного размера с последующим инвертированием всех значений в правой нижней копии $n \times n$, как видно из следующих примеров (как обычно, `T` обозначает `true`, а `F` — `false`):

H(1)	*H(2)*	*H(4)*
T	T T	T T T T
	T F	T F T F
		T T F F
		T F F T

Напишите программу, которая получает целое число `n` как аргумент командной строки и выводит $H(n)$. Предполагается, что `n` является степенью 2.

1.4.30. *Слухи.* Алиса устраивает вечеринку с n другими гостями, включая Боба. Боб рассказывает об этом одному из гостей. Человек, услышавший этот слух впервые, немедленно рассказывает о нем одному другому человеку, случайно равномерно выбирая его из числа гостей, за исключением Алисы и человека, от которого он

эту информацию услышал. Если человек (включая Боба) получает информацию во второй раз, он не передает ее дальше. Напишите программу для оценки вероятности того, что слух о вечеринке достигнет всех участников (не считая Алисы). Также оцените количество участников, до которых доберется слух о вечеринке.

1.4.31. *Подсчет простых чисел.* Сравните `PrimeSieve` с методом, который использовался для демонстрации команды `break` в конце раздела 1.3. Это классический пример компромисса между затратами памяти и времени: программа `PrimeSieve` работает быстро, но для работы ей необходим массив `boolean` длины `n`; другое решение использует всего две целочисленные переменные, но работает намного медленнее. Оцените величину этих различий: найдите значение `n`, для которого второе решение проводит вычисление примерно за то же время, что и запуск программы `PrimeSieve` с аргументом `1000000`.

1.4.32. *Сапер.* Напишите программу, которая получает в аргументах командной строки три числа `m`, `n` и `p` и строит массив `m` × `n` с элементами `boolean`, в котором каждый элемент занят с вероятностью `p`. В игре «Сапер» занятые ячейки изображают мины, а пустые считаются безопасными. Выведите массив, обозначая мины звездочками, а пустые ячейки — точками. Затем создайте целочисленный двумерный массив с количеством мин, расположенным по соседству (сверху, снизу, слева, справа и по диагонали).

```
* * . . .     * * 1 0 0
. . . . .     3 3 2 0 0
. * . . .     1 * 1 0 0
```

Напишите код так, чтобы в нем было как можно меньше потенциальных особых случаев, используя массив `(m+2)` × `(n+2)` с элементами `boolean`.

1.4.33. *Поиск дубликата.* Для заданного массива длины `n`, элементы которого лежат в диапазоне от 1 до `n`, напишите фрагмент кода для поиска дубликатов. Дополнительный массив использовать нельзя (но и сохранять исходное содержимое массива не нужно).

1.4.34. *Длина случайных блужданий без самопересечений.* Предположим, размер сетки не ограничен. Проведите эксперименты для оценки средней длины пути.

1.4.35. *Трехмерные блуждания без самопересечений.* Экспериментальным путем докажите, что для трехмерных блужданий без самопересечений вероятность попадания в тупик равна 0. Вычислите среднюю длину пути для разных значений `n`.

1.4.36. *Случайные обходчики.* Предположим, n случайных обходчиков начинают движение в центре сетки $n \times n$ и на каждом шаге перемещаются на одну ячейку, всякий раз выбирая направление (налево, направо, вверх, вниз) с равной вероятностью. Напишите программу, которая поможет сформулировать и протестировать гипотезу относительно количества шагов перед тем, как будут посещены все ячейки.

1.4.37. *Бридж.* В партии в бридж четырем игрокам сдаются по 13 карт. Один из важных статистических показателей раздачи — распределение мастей, то есть

количество карт каждой масти на руке. Какой расклад более вероятен: 5–3–3–2, 4–4–3–2 или 4–3–3–3?

1.4.38. *Дни рождения.* Предположим, люди заходят в пустую комнату до тех пор, пока среди них не найдется пара людей с одинаковым днем рождения. Сколько людей в среднем должно зайти в комнату, чтобы нашлось совпадение? Проведите эксперименты для оценки значения. Предполагается, что дни рождения определяются равномерно распределенными целыми числами от 0 до 364.

1.4.39. *Собиратель купонов.* Запустите эксперименты для проверки классического теоретического результата, согласно которому ожидаемое количество купонов, необходимое для сбора всех возможных n значений, равно приблизительно $n H_n$, где H_n — n-е гармоническое число. Например, если вы внимательно понаблюдаете за карточным столом (а карты сдаются из достаточно большого количества колод, перемешанных равномерно), вам придется в среднем дожидаться сдачи приблизительно 235 карт, прежде чем вы получите карты всех номиналов.

1.4.40. *Тасование методом чередования.* Напишите программу для изменения порядка колоды из n карт с использованием модели Гилберта — Шеннона — Ридса — тасования методом чередования. Сначала сгенерируйте случайное целое число r в биномиальном распределении: бросьте монетку n раз; пусть r — количество выпавших «орлов». Теперь разделите колоду на две стопки: первые r карт и остальные $n-r$ карт. Чтобы завершить тасование, возьмите верхние карты с двух стопок и поместите их под низ новой стопки. Если в первой стопке осталось n_1 карт, а во второй стопке — n_2 карт, переложите следующую карту из первой стопки с вероятностью $n_1/(n_1 + n_2)$ или из второй стопки с вероятностью $n_2/(n_1 + n_2)$. Выясните, сколько таких операций тасования необходимо применить к колоде из 52 карт, чтобы получить (почти) равномерно распределенную колоду.

1.4.41. *Биномиальные коэффициенты.* Напишите программу, которая получает целое число `n` в аргументе командной строки и создает двумерный ступенчатый массив `a[][]`, в котором элемент `a[n][k]` содержит вероятность выпадения ровно `k` «орлов» при `n` бросках монетки (при условии, что выпадение «орла» или «решки» равновероятно). Эти числа образуют так называемое биномиальное распределение: если умножить каждый элемент в строке `i` на 2^n, вы получите *биномиальные коэффициенты* (коэффициенты x^k в $(x+1)^n$), образующие *треугольник Паскаля.* Чтобы вычислить их, начните с `a[n][0] = 0.0` для всех `n` и `a[1][1] = 1.0`, затем вычислите значения последовательных строк слева направо по формуле `a[n][k] = (a[n-1][k] + + a[n-1][k-1]) / 2.0`.

Треугольник Паскаля	*Биномиальное распределение*
`1`	`1`
`1 1`	`1/2 1/2`
`1 2 3`	`1/4 1/2 1/4`
`1 3 3 1`	`1/8 3/8 3/8 1/8`
`1 4 6 4 1`	`1/16 1/4 3/8 1/4 1/16`

1.5. Ввод и вывод

В этом разделе мы расширим набор простых абстракций (аргументы командной строки и стандартный вывод), которые использовались до сих пор для взаимодействия программм Java с внешним миром, и дополним его *стандартным вводом*, *стандартной графикой* и *стандартным звуком*. Стандартный ввод упрощает написание программ, способных обрабатывать любые объемы входных данных, и взаимодействие с программами; *стандартная графика* позволяет работать с графическими представлениями компьютерных изображений, освобождая разработчика от необходимости кодировать все данные в текстовом виде; и *стандартный звук* позволяет работать со звуком в программах. Эти расширения очень практичны; вы увидите, что они открывают перед вами совершенно новый мир программирования.

Под объединяющим термином «ввод/вывод» понимаются механизмы взаимодействия программ с внешним миром. Операционная система вашего компьютера управляет физическими устройствами, подключенными к компьютеру. Для реализации стандартных абстракций ввода/вывода используются библиотеки методов, которые, в свою очередь, обращаются к операционной системе.

Вы уже умеете передавать аргументы в командной строке и выводить строки в терминальном окне; в этом разделе мы постараемся предоставить куда более широкий набор инструментов для обработки и представления данных. Как и методы `System.out.print()` и `System.out.println()`, которые вы уже использовали в программах, эти методы не реализуют функции в чисто математическом смысле — они призваны создавать некий побочный эффект на устройстве ввода или устройстве вывода. Сейчас нас будет интересовать использование таких устройств для передачи информации в программы и из них.

Важнейшая особенность стандартных механизмов ввода/вывода — отсутствие ограничений на объем ввода/вывода (с точки зрения программы). Ваши программы могут потреблять входные данные и генерировать выходные данные бесконечно.

Стандартные механизмы ввода/вывода часто применяются для работы с файлами на внешних запоминающих устройствах вашего компьютера. Стандартный ввод, стандартный вывод, стандартная графика и стандартный звук легко связываются с файлами. При наличии таких связей ваши программы Java могут легко сохранять свои результаты в файлах для долгосрочного хранения или для последующего использования в других программах либо для иных целей.

Краткий обзор

Традиционная модель, которую мы использовали при написании программ Java до настоящего момента, хорошо служила нам с раздела 1.1. Чтобы сформировать контекст, мы начнем с краткого обзора модели.

Программа Java получает входные данные из командной строки и выводит результаты в виде текста. По умолчанию и аргументы командной строки, и стандартный вывод ассоциируются с приложением, получающим команды (то, в котором вы вводите команды `java` и `javac`). Мы будем обозначать такое приложение общим термином «терминальное окно». Как показало время, эта модель предоставляет удобный и понятный механизм непосредственного взаимодействия наших программ с данными.

Аргументы командной строки

Этот механизм, который мы использовали для получения входных данных для наших программ, является стандартной частью программирования на языке Java. Во всех классах присутствует метод `main()`, который в своем аргументе получает массив `args[]` типа `String`. Этот массив содержит набор аргументов командной строки, введенный с клавиатуры и переданный в Java операционной системой. По установленным правилам как Java, так и операционная система обрабатывают аргументы в виде строк, и при передаче числового аргумента следует использовать такие методы, как `Integer.parseInt()` или `Double.parseDouble()`, для преобразования `String` к соответствующему типу.

Стандартный вывод

Для вывода выходных значений в предыдущих примерах мы использовали системные методы `System.out.println()` и `System.out.print()`. Java помещает результаты выполнения этих вызовов в абстрактный поток символов, называемый *стандартным выводом*. По умолчанию операционная система связывает стандартный вывод с терминальным окном. Соответственно весь вывод в наших программах до сих пор осуществлялся в терминальном окне.

В качестве отправной точки (и для напоминания) программа `RandomSeq` (листинг 1.5.1) использует эту модель. Она получает аргумент командной строки `n` и выводит последовательность из `n` случайных чисел от 0 до 1.

К аргументам командной строки и стандартному выводу будут добавлены три новых механизма, которые устраняют их ограничения и предоставляют гораздо более удобную модель программирования. С этими механизмами можно будет рассматривать программу Java на абстрактном уровне как преобразование стандартного входного потока и аргументов командной строки в потоки стандартного вывода, стандартной графики и стандартного звука.

Стандартный ввод

Наш класс `StdIn` представляет собой библиотеку, реализующую абстракцию стандартного ввода в дополнение к абстракции стандартного вывода. Подобно тому как вы можете в любой момент вывести значение в стандартный вывод (выходной поток, поток вывода) во время выполнения программы, вы также можете в любой момент прочитать значение из стандартного входного потока (стандартного ввода).

Листинг 1.5.1. Генерирование случайной последовательности

```
public class RandomSeq
{
   public static void main(String[] args)
   {  // Вывод случайной серии вещественных значений в диапазоне [0, 1)
      int n = Integer.parseInt(args[0]);
      for (int i = 0; i < n; i++)
         System.out.println(Math.random());
   }
}
```

Программа демонстрирует традиционную модель, которую мы использовали ранее в программировании на Java. Она получает аргумент командной строки n и выводит n случайных чисел от 0.0 до 1.0. С точки зрения программы длина входных данных неограниченна.

```
% java RandomSeq 1000000
0.2498362534343327
0.5578468691774513
0.5702167639727175
0.32191774192688727
0.6865902823177537
...
```

Стандартная графика

Наш класс `StdDraw` позволяет создавать графические изображения в программах. Его простая графическая модель в основном ориентирована на построение рисунков из точек и линий в окне на компьютере. `StdDraw` также включает средства для работы с текстом, цветом и анимацией.

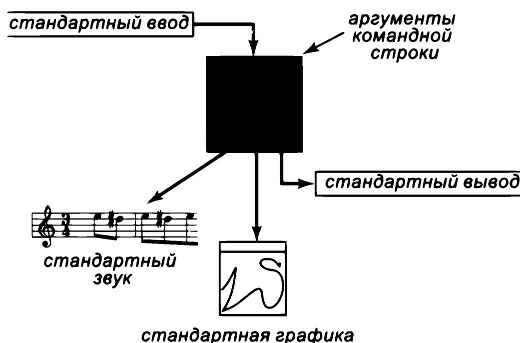
Стандартный звук

Наш класс `StdAudio` предназначен для работы со звуком в программах. Он использует стандартный формат для преобразования массива чисел в звук.

В Java существуют встроенные средства, поддерживающие такие абстракции, как стандартный ввод, стандартная графика и стандартный звук, но они достаточно сложны в использовании, поэтому мы разработали для них более простой и удобный интерфейс в наших библиотеках `StdIn`, `StdDraw` и `StdAudio`. Чтобы модель программирования была логически завершенной, мы также добавили библиотеку `StdOut`. Чтобы использовать эти библиотеки, следует открыть доступ к файлам StdIn.java, StdOut.java, StdDraw.java и StdAudio.java в Java (см. раздел «Вопросы и ответы» в конце этого раздела).

Абстракции стандартного ввода и стандартного вывода берут начало в разработке операционной системы Unix в 1970-х годах. В той или иной форме они присутствуют во всех современных операционных системах. И хотя эти абстракции прими-

тивны по сравнению с различными механизмами, разработанными с того времени, современные программисты продолжают использовать их как надежные средства передачи данных программам. Мы разработали библиотеки стандартной графики и стандартного звука по аналогии с этими абстракциями, чтобы упростить работу с графикой и звуком в ваших программах.



Обобщенная структура программы Java

Стандартный вывод

Методы Java `System.out.print()` и `System.out.println()` реализуют базовые абстракции ввода/вывода, необходимые для наших программ. Тем не менее для того чтобы операции со стандартным вводом и стандартным выводом выполнялись по единой схеме (и для внесения ряда технических усовершенствований), начиная с этого раздела и в оставшейся части книги мы будем использовать аналогичные методы, определенные в библиотеке `StdOut`. Методы `StdOut.print()` и `StdOut.println()` почти не отличаются от методов Java, которые мы использовали ранее (различия описаны на сайте книги — впрочем, сейчас они для нас неважны). А сейчас вас должен больше интересовать метод `StdOut.printf()`, рассматриваемый в этом разделе, потому что он предоставляет больше возможностей для управления внешним видом выводимых данных. Эта функциональность поддерживалась языком C с начала 1970-х годов и сохранилась во многих современных языках, потому что она исключительно полезна.

`public class StdOut`

`void print(String s)`	выводит s в стандартный вывод
`void println(String s)`	выводит s и символ новой строки
`void println()`	выводит символ новой строки
`void printf(String format, ... )`	выводит аргументы в стандартный вывод в виде, описанном форматной строкой format

API нашей библиотеки статических методов стандартного вывода

С самого первого раза, когда мы выводили значения `double` в программах, нам приходилось отвлекаться на лишнюю точность в выводимых значениях. Например, при выводе числа π методом `System.out.print(Math.PI)` выводится строка `3.141592653589793`, хотя, возможно, вы бы предпочли получить результат `3.14` или `3.14159`. Методы `print()` и `println()` выводят значения с плавающей точкой всегда с 15 десятичными знаками, даже если мы были бы рады более короткой записи. Метод `print()` обладает большей гибкостью. Например, он позволяет задать количество знаков в дробной части при преобразовании чисел с плавающей точкой в строки для вывода. Так, вызов `StdOut.printf("%7.5f", Math.PI)` выведет `3.14159`, а `System.out.print(t)` в программе `Newton` (листинг 1.3.6) можно заменить вызовом

```
StdOut.printf("The square root of %.1f is %.6f", c, t);
```

для получения следующего вывода:

```
The square root of 2.0 is 1.414214
```

Далее мы разберем смысл этой строки и принцип действия этого метода, а также варианты для работы с другими встроенными типами данных.

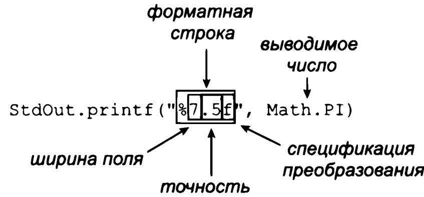
Основы форматирования при выводе

В простейшем случае `printf()` получает два аргумента. Первый аргумент называется *форматной строкой*. Он содержит *спецификацию преобразования* — описание того, как второй аргумент должен быть преобразован в строку для вывода. Спецификация преобразования записывается в форме `%w.pc`, где `w` и `p` — целые числа, а `c` — символ, и интерпретируется следующим образом.

- `w` — *ширина поля*, то есть нужное количество выводимых символов. Если фактическое количество выводимых символов превышает ширину поля (или равно ей), то ширина поля игнорируется; в противном случае вывод дополняется пробелами слева. Отрицательная ширина поля означает, что вывод должен дополняться пробелами справа.
- `.p` — *точность*. Для вещественных чисел точность определяет количество цифр в дробной части, для строк — количество выводимых символов (не обязательно вся ширина поля). С целыми числами точность не используется.
- `c` — *код преобразования*. Самые распространенные коды преобразования: `d` (целочисленные типы Java), `f` (значения с плавающей точкой), `e` (значения с плавающей точкой в научной записи), `s` (строковые значения) и `b` (значения `boolean`).

Ширину поля и точность можно опустить, но код преобразования должен присутствовать в каждой спецификации.

Самое важное, что следует запомнить о `printf()`, — то, что *код преобразования должен соответствовать типу соответствующего аргумента*. Иначе говоря, компи-



Строение команды вывода с форматированием

лятор Java должен знать, как привести тип аргумента к типу, определяемому кодом преобразования. Любой тип данных может быть преобразован в `String`, но если вы используете команду `StdOut.printf("%12d", Math.PI)` или `StdOut.printf("%4.2f", 512)`, произойдет ошибка времени выполнения `IllegalFormatConversionException`[1].

Форматная строка

Форматная строка может содержать символы, не входящие в спецификацию преобразования. Спецификация преобразования заменяется значением аргумента (преобразованным в строку в соответствии с описанием), а все остальные символы выводятся в неизменном виде. Например, команда

```
StdOut.printf("PI is approximately %.2f.\n", Math.PI);
```

выводит строку

```
PI is approximately 3.14.
```

Обратите внимание на необходимость явного включения символа новой строки `\n` в форматную строку для вывода новой строки вызовом `printf()`.

Множественные аргументы

Метод `printf()` может получать более двух аргументов. В этом случае для каждого дополнительного аргумента в форматную строку включается дополнительная спецификация преобразования — возможно, отделенная от предыдущей другими символами, которые в таком случае выводятся без преобразования. Например, при выплате кредита может использоваться код, внутренний цикл которого содержит команды

```
String formats = "%3s  $%6.2f   $%7.2f   $%5.2f\n";
StdOut.printf(formats, month[i], pay, balance, interest);
```

[1] Автоматическое приведение типа не может справиться с такой ситуацией, так как для него необходимо, чтобы компилятор «знал» требуемый тип значения, но здесь форматная строка для компилятора – это всего лишь строка символов. Ее интерпретация происходит только внутри метода, во время его выполнения, когда все передаваемые значения уже сформированы. — *Примеч. науч. ред.*

для вывода второй и последующих строк в таблице в следующем виде (см. упражнение 1.5.13):

```
     payment    balance  interest
Jan  $299.00   $9742.67   $41.67
Feb  $299.00   $9484.26   $40.59
Mar  $299.00   $9224.78   $39.52
...
```

Печать с форматированием удобна, потому что такой код намного компактнее кода с конкатенацией, используемого для создания выходных строк.

Мы описали только базовые возможности; за дополнительной информацией обращайтесь на сайт книги.

тип	*код*	*типичный литерал*	*пример форматной строки*	*преобразованные строковые значения при выводе*
`int`	`d`	`512`	`"%14d"`	`"           512"`
			`"%-14d"`	`"512           "`
`double`	`f`	`1595.1680010754388`	`"%14.2f"`	`"       1595.17"`
	`e`		`"%.7f"`	`"1595.1680011"`
			`"%14.4e"`	`"    1.5952e+03"`
`String`	`s`	`"Hello, World"`	`"%14s"`	`"  Hello, World"`
			`"%-14s"`	`"Hello, World  "`
			`"%-14.5s"`	`"Hello         "`
`boolean`	`b`	`true`	`"%b"`	`"true"`

Правила преобразования формата для printf()
(многие другие возможности описаны на сайте книги)

Стандартный ввод

Наша библиотека `StdIn` получает данные из стандартного входного потока, который может быть пустым или содержащим последовательность значений, разделенных пропусками (пробелами, табуляциями, символами новой строки и т. д.). Каждое значение представляет собой строку или значение одного из примитивных типов Java. Одна из ключевых особенностей стандартного входного потока заключается в том, что чтение из него является *разрушающим* — программа *поглощает* прочитанные значения. После того как программа прочитает значение, она не сможет вернуться и прочитать его заново. Это допущение создает некоторые неудобства, но оно отражает физические характеристики некоторых устройств ввода. Описание API `StdIn` приведено на следующей странице. Методы делятся на четыре категории.

- Методы для чтения отдельных значений (по одному).
- Методы чтения строк (по одной).
- Методы чтения символов (по одному).
- Методы чтения серии однотипных значений.

Обычно лучше не смешивать методы из разных категорий в одной программе. Эти методы, как правило, не требуют пояснений (имя метода описывает выполняемое им действие), но некоторые особенности работы заслуживают более внимательного рассмотрения, поэтому мы подробно разберем несколько примеров.

`public class StdIn`

	Методы для чтения отдельных лексем из стандартного ввода
`boolean isEmpty()`	стандартный ввод пуст (или содержит только пробельные символы)?
`int readInt()`	читает слово, преобразует его к типу int и возвращает результат
`double readDouble()`	читает слово, преобразует его к типу double и возвращает результат
`boolean readBoolean()`	читает слово, преобразует его к типу boolean и возвращает результат
`String readString()`	читает слово и возвращает его в виде значения String
	Методы для чтения символов из стандартного ввода
`boolean hasNextChar()`	в стандартном вводе еще остались символы?
`char readChar()`	читает символ из стандартного ввода и возвращает его
	Методы для чтения строк из стандартного ввода
`boolean hasNextLine()`	в стандартном вводе еще осталась текстовая строка?
`String readLine()`	читает оставшуюся часть текстовой строки и возвращает ее в виде String
	Методы для чтения оставшихся символов из стандартного ввода
`int[] readAllInts()`	читает все оставшиеся в потоке слова и возвращает их в виде массива int
`double[] readAllDoubles()`	читает все оставшиеся в потоке слова и возвращает их в виде массива double
`boolean[] readAllBooleans()`	читает все оставшиеся в потоке слова и возвращает их в виде массива Boolean
`String[] readAllStrings()`	читает все оставшиеся слова и возвращает их в виде массива String
`String[] readAllLines()`	читает все оставшиеся текстовые строки и возвращает их в виде массива String
`String readAll()`	читает весь оставшийся ввод и возвращает его в виде String

Примечание 1: «словом» (token) здесь называется максимальная последовательность символов, не являющихся пробельными.

Примечание 2: перед чтением слова все начальные пробелы отбрасываются.

Примечание 3: существуют аналогичные методы для чтения значений типа byte, short, long и float.

Примечание 4: все методы чтения инициируют исключение, если им не удается прочитать следующее значение (либо нет больше входных данных, либо они не соответствуют ожидаемому типу).

API нашей библиотеки статических методов стандартного ввода

Ввод данных

Когда вы вводите команду `java`, чтобы запустить программу на языке Java из командной строки, на самом деле вы: 1) вводите команду для запуска программы; 2) передаете аргументы командной строки и 3) начинаете наполнять стандартный поток ввода. Строка символов, вводимая в окне терминала после командной строки, образует стандартный поток ввода. Вводя символы, вы взаимодействуете с программой. Программа ожидает, когда вы начнете вводить символы в терминальном окне.

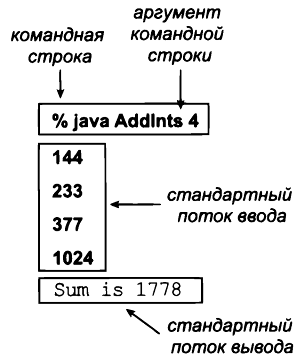
Рассмотрим пример: программа `AddInts` получает аргумент командной строки `n`, читает `n` чисел из стандартного ввода, суммирует их и выводит результат в стандартный вывод. Если запустить программу командой `java AddInts 4`, после получения аргумента командной строки программа вызывает метод `StdIn.readInt()` и ждет, пока вы введете с клавиатуры целое число. Допустим, первым значением должно быть число 144. Вы вводите `1`, потом `4` и снова `4` — пока не происходит ничего, потому что `StdIn` не знает, что ввод числа строки, содержащей число, завершен. Но когда вы нажимаете клавишу Return, чтобы сообщить о завершении ввода, `StdIn.readInt()` немедленно возвращает значение `144`. Ваша программа прибавляет его к `sum` и вызывает `StdIn.readInt()` еще раз. И снова ничего не происходит, пока не будет введено второе значение: если ввести `2`, потом `3`, снова `3` и нажать Return, чтобы сообщить о завершении ввода, `StdIn.readInt()` вернет значение `233`, которое также прибавляется к `sum`. Когда так будут введены все четыре числа, `AddInts` не будет ожидать поступления новых данных и выведет значение `sum`, как и предполагалось.

```
public class AddInts
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      int sum = 0;
      for (int i = 0; i < n; i++)
      {
         int value = StdIn.readInt();
         sum += value;
      }
      StdOut.println("Sum is " + sum);
   }
}
```

разбор аргумента командной строки

чтение из стандартного потока ввода

запись в стандартный поток вывода



Строение команды

Формат ввода

Если вы введете `abc`, `12.2` или `true`, когда `StdIn.readInt()` ожидает `int`, метод отреагирует исключением `InputMismatchException`. Формат для каждого типа

почти не отличается от формата, используемого для определения литералов в программах Java. Для удобства `StdIn` рассматривает цепочки смежных пробельных символов как один пробел и позволяет разделять строки такими цепочками. Неважно, сколько пробелов разделяет числа, вводите ли вы числа в одной строке, разделяете их табуляциями или разносите на несколько строк (кроме того, что ваше терминальное приложение обрабатывает стандартный ввод по одной физической строке, поэтому оно будет ждать нажатия клавиши Return, прежде чем направить все числа в этой строке в стандартный ввод). Во входном потоке могут одновременно находиться значения разных типов, но когда программа ожидает получить значение конкретного типа, во входном потоке должно содержаться значение именно этого типа.

Интерактивный ввод

Программа `TwentyQuestions` (листинг 1.5.2) представляет простой пример взаимодействия с пользователем. Программа генерирует случайное число и предлагает пользователю угадать его. (Попутно отметим, что бинарный поиск позволяет всегда получить ответ не более чем за 20 вопросов, — см. раздел 4.2.) Принципиальное отличие этой программы от всех остальных, написанных до настоящего момента, заключается в том, что пользователь может управлять программной логикой во время выполнения программы. Эта возможность была чрезвычайно важна на ранней стадии вычислительных технологий, но сейчас такие программы пишутся редко, потому что современные приложения обычно получают входные данные через графический интерфейс, — см. главу 3. Даже такая простая программа, как `TwentyQuestions`, наглядно показывает, что обеспечение взаимодействия с пользователем порой бывает очень непростым делом, потому что вам приходится заранее учитывать все возможные варианты ввода.

Обработка входного потока произвольного размера

Как правило, потоки ввода содержат конечный объем данных: ваша программа проходит по входному потоку, поглощая значения до тех пор, пока поток не опустеет. Однако ограничений на размер входного потока не существует, и некоторые программы просто обрабатывают весь полученный ввод. Программа `Average` (листинг 1.5.3) читает серию чисел с плавающей точкой из стандартного ввода и выводит их среднее арифметическое. Она демонстрирует ключевую особенность потоков ввода: длина потока неизвестна программе. Пользователь вводит все имеющиеся числа, после чего программа вычисляет их среднее значение. Прежде чем читать очередное число, программа использует метод `StdIn.isEmpty()` для проверки наличия чисел в потоке ввода. Как сообщить программе о том, что данных больше не осталось? Для этого пользователь вводит специальную комбинацию символов, известную как *конец файла*. К сожалению, терминальные приложения, которые обычно встречаются в современных операционных системах, используют

Листинг 1.5.2. Интерактивный ввод

secret	загаданное число
guess	догадка пользователя

```
public class TwentyQuestions
{
   public static void main(String[] args)
   {  // Программа генерирует число и реагирует на догадки,
      // пока пользователь пытается угадать число.
      int secret = 1 + (int) (Math.random() * 1000000);
      StdOut.print("I'm thinking of a number ");
      StdOut.println("between 1 and 1,000,000");
      int guess = 0;
      while (guess != secret)
      {  // Программа запрашивает догадку и выводит ответ.
         StdOut.print("What's your guess? ");
         guess = StdIn.readInt();
         if (guess == secret) StdOut.println("You win!");
         if (guess < secret)  StdOut.println("Too low ");
         if (guess > secret)  StdOut.println("Too high");
      }
   }
}
```

Программа реализует простую игру по угадыванию числа. Пользователь вводит числа, каждое из которых может рассматриваться как неявный вопрос («Загадано это число?»), а программа сообщает, оказалась ли догадка пользователя больше или меньше загаданного числа. Пользователь всегда может добраться до ответа («You win!») менее чем за 20 вопросов. Для использования этой программы необходимо открыть доступ к классам StdIn и StdOut в коде Java (см. первый вопрос в разделе «Вопросы и ответы» в конце раздела).

```
% java TwentyQuestions
I'm thinking of a number between 1 and 1,000,000
What's your guess? 500000
Too high
What's your guess? 250000
Too low
What's your guess? 375000
Too high
What's your guess? 312500
Too high
What's your guess? 300500
Too low
...
```

разные символы для этой критически важной комбинации. В этой книге мы будем использовать комбинацию Ctrl+D (во многих системах комбинация Ctrl+D должна располагаться в отдельной строке); также конец файла часто обозначается комбинацией Ctrl+Z в отдельной строке. Программа `Average` проста, но она предоставляет принципиально новую возможность в программировании: со стандартным вводом вы можете писать программы, обрабатывающие неограниченные объемы данных. Как вы увидите, написание таких программ позволяет эффективно организовать обработку данных.

Листинг 1.5.3. Вычисление среднего арифметического для числового потока

```
public class Average
{
   public static void main(String[] args)
   {  // Вычисление среднего для чисел из стандартного ввода.
      double sum = 0.0;
      int n = 0;
      while (!StdIn.isEmpty())
      {  // Чтение числа из стандартного ввода и прибавление его к sum.
         double value = StdIn.readDouble();
         sum += value;
         n++;
      }
      double average = sum / n;
      StdOut.println("Average is " + average);
   }
}
```

n	количество прочитанных чисел
sum	накопленная сумма

Программа читает из стандартного ввода последовательность чисел с плавающей точкой, вычисляет их среднее арифметическое и выводит его в стандартный вывод (при условии, что сумма не вызывает переполнения). С точки зрения программы размер потока ввода неограничен. Справа внизу приведены команды, которые используют механизм перенаправления и каналы (см. следующий подраздел) для передачи программе около 100 000 чисел.

```
% java Average
10.0 5.0 6.0
3.0
7.0 32.0
<Ctrl-D>
Average is 10.5
```

```
% java RandomSeq 100000 > data.txt
% java Average < data.txt
Average is 0.5010473676174824
% java RandomSeq 100000 | java Average
Average is 0.5000499417963857
```

Стандартный ввод стал существенным шагом вперед от модели аргументов командной строки, которую мы использовали ранее. Есть два основных усовершенствования, которые демонстрируют программы TwentyQuestions и Average. Во-первых, стандартный ввод позволяет работать с программой в интерактивном режиме — с аргументами командной строки можно предоставить данные программе только *перед* началом выполнения. Во-вторых, он позволяет читать большие объемы данных — с аргументами командной строки можно вводить только значения, помещающиеся в командной строке[1]. В самом деле, как демонстрирует программа Average, объем данных может быть практически неограниченным, и это предположение упрощает многие программы. Третий довод в пользу стандартного ввода заключается в том, что операционная система позволяет сменить источник стандартного ввода, чтобы вам не приходилось вводить все данные вручную. Сейчас мы рассмотрим механизмы, которые обеспечивают эту возможность.

[1] Размер буфера, предоставляемого для командной строки, достаточно велик и его хватает для большинства применений. Кроме того, в Unix существуют программы (команды) для «расширения» командной строки, например xargs. — *Примеч. науч. ред.*

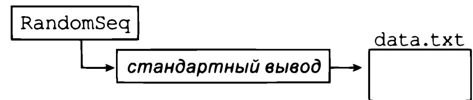
Перенаправление и каналы

Во многих приложениях ввод входных данных в стандартном потоке ввода в терминальном окне неприемлем, потому что в этом случае вычислительные возможности наших программ ограничиваются объемом вводимых данных (и скоростью набора). Кроме того, информацию, введенную в стандартном потоке ввода, часто требуется сохранить для последующего использования. Чтобы обойти эти ограничения, следует вспомнить, что стандартный ввод является *абстракцией* — программа ожидает, что она будет читать данные из входного потока, но при этом не зависит от *источника* этого потока. Стандартный вывод также является аналогичной абстракцией. Сила этих абстракций напрямую обусловлена нашей возможностью (средствами операционной системы) выбирать другие источники стандартного ввода и стандартного вывода: файлы, сетевые ресурсы или другие программы. Эти механизмы реализованы во всех современных операционных системах.

Перенаправление стандартного вывода в файл

Добавляя простую директиву в команду запуска программы, можно *перенаправить* стандартный поток вывода в файл — для долгосрочного хранения или для передачи в качестве ввода другой программе. Например, команда

```
% java RandomSeq 1000 > data.txt
```



Перенаправление стандартного вывода в файл

сообщает, что стандартный поток вывода не должен выводиться в терминальном окне; вместо этого он записывается в текстовый файл с именем data.txt. Каждый вызов `System.out.print()` или `System.out.println()` будет добавлять текст в конец файла. В данном примере это приводит к созданию файла с тысячей случайных значений. В терминальном окне при этом ничего не отображается; все данные направляются прямо в файл, указанный после знака >. Так можно сохранить информацию для последующего чтения. Обратите внимание: для работы этого механизма нам не потребовалось никаких изменений в программе `RandomSeq` (листинг 1.5.1) — программа использует абстракцию стандартного вывода, и переход на другую реализацию этой абстракции на ней никак не отражается. Перенаправление может использоваться для сохранения вывода любой написанной вами программы. После того как вы потратите значительные усилия для получения результата, этот результат часто бывает нужно сохранить для использования в будущем. В современных системах для сохранения информации можно воспользоваться механизмом копирования/вставки (или другим аналогичным механизмом, предоставляемым операционной системой)[1], но для больших объемов данных копировать вручную неудобно. В то же время механизм перенаправления специально создавался для упрощения работы с большими объемами данных.

[1] Здесь подразумевается графическая среда и оконный интерфейс. — *Примеч. науч. ред.*

Листинг 1.5.4. Простой фильтр

```
public class RangeFilter
{
   public static void main(String[] args)
   {  // Исключение чисел, не входящих в диапазон от lo до hi.
      int lo = Integer.parseInt(args[0]);
      int hi = Integer.parseInt(args[1]);
      while (!StdIn.isEmpty())
      {  // Обработка одного числа.
         int value = StdIn.readInt();
         if (value >= lo && value <= hi)
            StdOut.print(value + " ");
      }
      StdOut.println();
   }
}
```

lo	нижняя граница диапазона
hi	верхняя граница диапазона
value	текущее число

Фильтр копирует в выходной поток только те числа из входного потока, которые входят в диапазон, заданный аргументами командной строки. Длина потоков не ограничивается.

```
% java RangeFilter 100 400
358 1330 55 165 689 1014 3066 387 575 843 203 48 292 877 65 998
358 165 387 203 292
<Ctrl-D>
```

Перенаправление данных из файла в стандартный ввод

Аналогичным образом можно перенаправить стандартный поток ввода, чтобы класс `StdIn` читал данные из файла, а не из терминального окна:

```
% java Average < data.txt
```

```
% java Average < data.txt
```

Перенаправление данных из файла в стандартный ввод

Эта команда читает серию чисел из файла data.txt и вычисляет их среднее арифметическое. Знак < — директива, которая приказывает операционной системе реализовать стандартный поток, прочитав содержимое текстового файла data.txt, — вместо того чтобы ожидать, пока пользователь введет данные в терминальном окне. Когда программа вызывает `StdIn.readDouble()`, операционная система читает значение из файла. При этом файл data.txt может быть создан в любом приложении, не обязательно в программе Java, — на вашем компьютере найдется множество приложений, которые могут создавать текстовые файлы. Возможность перенаправления данных из файла в стандартный ввод позволяет создавать *код, управляемый данными*: вы можете изменить данные, обрабатываемые программой, без внесения каких-либо изменений в программу. Вместо этого данные хранятся в файлах, а вы пишете программы, которые читают данные из стандартного потока ввода.

Взаимодействие двух программ

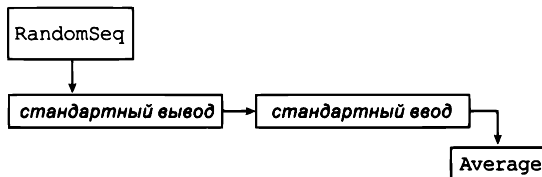
Самый гибкий и универсальный способ наполнения входного и выходного потоков — поручить это вашей же программе! В этом механизме используются *каналы* (pipes). Например, команда

```
% java RandomSeq 1000 | java Average
```

сообщает, что стандартный поток вывода программы RandomSeq и стандартный поток ввода для программы Average — это *один и тот же* поток. По сути, все выглядит так, словно программа RandomSeq набирает числа, генерируемые ею, в терминальном окне во время выполнения Average. Этот пример также эквивалентен выполнению следующей серии команд:

```
% java RandomSeq 1000 > data.txt
% java Average < data.txt
```

Только в предыдущем примере файл data.txt не создается. Это различие очень важно, потому что оно снимает еще одно ограничение на размер потоков ввода и вывода, которые могут обрабатываться приложением. Например, 1000 в примере можно заменить на 1000000000, даже если у вас на компьютере не хватит места для хранения миллиарда чисел (хотя время для их обработки все равно потребуется). Когда в программе RandomSeq вызывается метод System.out.println(), в конец потока добавляется строка; когда Average вызывает StdIn.readInt(), строка считывается из начала потока (и после этого удаляется из него). Точная последовательность происходящих событий зависит от операционной системы: она может выполнять RandomSeq, пока программа не выдаст некоторый результат, после чего запустить Average для потребления этого результата, или же запустить программу Average, пока той не потребуются данные, и тогда запустить RandomSeq для получения необходимого ввода. Конечный результат остается одним и тем же, но вашим программам не нужно отвлекаться на эти подробности — они работают исключительно с абстракциями стандартного ввода и вывода.



Передача вывода одной программы на ввод другой программы

Фильтры

Каналы, одна из базовых особенностей исходной системы Unix в начале 1970-х годов, до сих пор сохраняются в современных системах, потому что они предостав-

ляют простую абстракцию для передачи данных между различными программами. О силе этой абстракции свидетельствует то, что многие программы Unix продолжают использоваться сегодня для обработки файлов, в тысячи и миллионы раз бóльших, чем представляли себе авторы этих программ. С другими программами Java можно обмениваться данными посредством вызова специальных методов, но стандартный ввод и стандартный вывод позволяют взаимодействовать с программами, написанными в другое время, а возможно, и на другом языке. Используя стандартный ввод и стандартный вывод, вы фактически соглашаетесь на реализацию простого интерфейса с окружающим миром.

Во многих типичных задачах удобно рассматривать программу как *фильтр*, который каким-то образом преобразует стандартный поток ввода в стандартный поток вывода в расчете на применение каналов для взаимодействия между программами. Например, программа `RangeFilter` (листинг 1.5.4) получает два аргумента командной строки и передает в стандартный вывод только те числа из стандартного ввода, которые лежат в заданном диапазоне. Представьте, например, что стандартный ввод содержит данные измерений от некоторого прибора, а фильтр используется для отсечения данных, выходящих за пределы интересующего вас диапазона.

Некоторые стандартные фильтры, написанные для Unix, продолжают существовать (иногда под другими именами) в виде команд современных операционных систем. Например, фильтр `sort` сортирует строки из стандартного потока ввода в заданном порядке:

```
% java RandomSeq 6 | sort
0.035813305516568916
0.14306638757584322
0.348292877655532103
0.5761644592016527
0.7234592733392126
0.9795908813988247
```

Сортировка рассматривается в разделе 4.2. Другой полезный фильтр, `grep`, выводит из стандартного ввода строки, соответствующие заданному шаблону. Например, если ввести команду

```
% grep lo < RangeFilter.java
```

будет получен следующий результат

```
// Исключение чисел, не входящих в диапазон от lo до hi.
int lo = Integer.parseInt(args[0]);
    if (value >= lo && value <= hi)
```

Программисты часто используют инструменты, подобные `grep`, для того чтобы вспомнить имена переменных или отдельные аспекты использования языка. Третий полезный фильтр, `more`, читает данные из стандартного потока ввода и выводит их в терминальном окне по страницам (экранам). Например, если ввести команду

```
% java RandomSeq 1000 | more
```

вы увидите столько чисел, сколько помещается в окне; программа `more` ожидает нажатия клавиши «пробел» перед выводом следующей порции. Возможно, термин «фильтр» выбран не совсем удачно — изначально он подразумевал такие программы, как `RangeFilter`, которые записывают некоторое подмножество данных из стандартного ввода в стандартный вывод. Тем не менее сейчас он часто используется для описания любых программ, которые читают данные из стандартного ввода и записывают их в стандартный вывод.

Множественные потоки

Для многих характерных задач пишутся программы, которые получают данные из нескольких источников и/или выдают результаты для нескольких приемников. В разделе 3.1 мы рассмотрим библиотеки `Out` и `In`, которые обобщают `StdOut` и `StdIn` для поддержки множественных потоков ввода и вывода. Эти библиотеки включают поддержку перенаправления потоков не только к файлам, но и к веб-страницам.

Обработка больших объемов информации играет важную роль во многих вычислительных приложениях. Представьте ученого, которому нужно проанализировать данные, собранные в ходе серии экспериментов; биржевого брокера, анализирующего информацию о недавних финансовых операциях; или студента, собирающего подборку фильмов и музыки. В этих (и множестве других) ситуаций программы, управляемые данными, стали нормой. Стандартный вывод, стандартный ввод, перенаправление и каналы предоставляют средства для решения таких задач в программах Java. Вы можете собирать на своем компьютере файлы, полученные из Интернета или из стандартных устройств, и использовать перенаправление и каналы для подключения данных к программам. Многие (если не большинство) примеры в этой книге поддерживают такую возможность.

Стандартная графика

До настоящего момента все абстракции ввода/вывода были ориентированы исключительно на текстовые строки. Пора ввести абстракцию для вывода графических изображений. Библиотека проста в использовании, а визуальные средства позволяют работать с существенно бо́льшим объемом информации, чем обычный текст. Как и в случае с `StdIn` и `StdOut`, наша стандартная абстракция графики реализуется библиотекой `StdDraw`, доступ к которой необходимо предоставить Java (см. первый вопрос в разделе «Вопросы и ответы» в конце раздела). Стандартная графика работает очень просто: представьте абстрактное устройство графического вывода, которое позволяет отображать линии и точки на двумерном «холсте»[1].

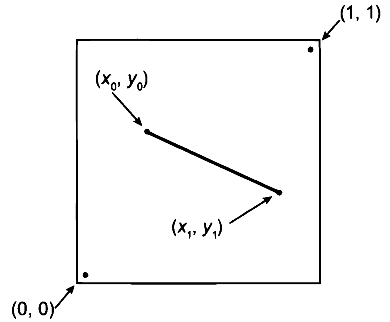
---

[1] Буквально «canvas», и эта абстракция часто используется в программировании. Аналогичная абстракция в Windows GDI носит название «контекст устройства» (device context). — *Примеч. науч. ред.*

Устройство реагирует на команды, которые выдаются нашей программой в виде вызовов методов StdDraw следующего вида:

```
public class StdDraw (основные команды графического вывода)
    void line(double x0, double y0, double x1, double y1)
    void point(double x, double y)
```

Как и методы стандартного ввода и вывода, эти методы почти не нуждаются в документировании: `StdDraw.line()` чертит отрезок прямой, соединяющий точку (x_0, y_0) с точкой (x_1, y_1); координаты точек передаются как аргументы вызова. Метод `StdDraw.point()` изображает небольшой кружок с центром в точке (x, y), координаты которой передаются как аргументы вызова. По умолчанию рисунок строится на единичном квадрате (все координаты x и y лежат в диапазоне от 0 до 1). StdDraw выводит содержимое холста в окне на экране вашего компьютера; линии и точки отображаются черным цветом на белом фоне. Окно включает команду меню для сохранения графики в файле в формате, подходящем для печати на бумаге или для размещения на веб-страницах.

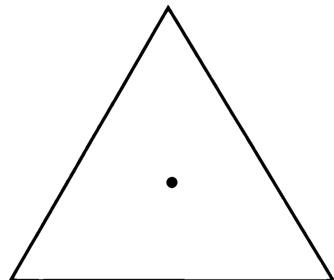


`StdDraw.line(x0, y0, x1, y1);`

Ваш первый рисунок

Эквивалент приложения HelloWorld для вывода графики с использованием StdDraw рисует равносторонний треугольник с точкой внутри. Треугольник строится из трех отрезков: из точки (0, 0) в левом нижнем углу в точку (1, 0), из этой точки в третью точку $(1/2, \sqrt{3}/2)$ и из этой точки обратно в (0, 0). После этого остается сделать последний штрих: нарисовать точку в середине треугольника. После того как вы успешно откомпилируете и запустите Triangle, вы сможете писать собственные программы, которые строят изображения из отрезков и точек. Средства графического вывода поднимают результаты, выводимые вашими программами, на совершенно новый уровень.

```
public class Triangle
{
   public static void main(String[] args)
   {
      double t = Math.sqrt(3.0)/2.0;
      StdDraw.line(0.0, 0.0, 1.0, 0.0);
      StdDraw.line(1.0, 0.0, 0.5, t);
      StdDraw.line(0.5,   t, 0.0, 0.0);
      StdDraw.point(0.5, t/3.0);
   }
}
```



Первый рисунок

Начав строить на компьютере графические изображения, вы сразу получаете отдачу в виде качественного улучшения ваших программ. В компьютерной программе

можно строить изображения, которые совершенно нереально построить вручную. В частности, вместо просмотра данных в числовом виде можно использовать диаграммы, куда более выразительные. Вскоре мы приведем другие примеры работы с графикой, но сначала необходимо рассмотреть еще несколько команд.

Управляющие команды

Размер холста по умолчанию равен 512 × 512 пикселов; если вы захотите изменить его, вызовите `setCanvasSize()` перед любыми другими методами вывода. Система координат для стандартной графики по умолчанию представляет собой единичный квадрат, однако диаграммы часто бывает удобнее строить в другом масштабе.

Например, в типичной ситуации используются координаты из некоторого диапазона по оси x и/или оси y. Также часто требуется рисовать отрезки разной толщины и точки нестандартного размера. Для таких случаев в `StdDraw` существуют следующие методы:

`public class StdDraw` *(основные управляющие методы)*

`void setCanvasSize(int w, int h)`	создает в окне холст с шириной w и высотой h (в пикселах)
`void setXscale(double x0, double x1)`	устанавливает для оси x диапазон координат (x0, x1)
`void setYscale(double y0, double y1)`	устанавливает для оси y диапазон координат (y0, y1)
`void setPenRadius(double radius)`	устанавливает радиус пера radius

Примечание: одноименные методы без аргументов возвращают значения по умолчанию, то есть диапазон координат единичного квадрата по осям x и y и радиус пера 0.002.

Например, последовательность вызовов

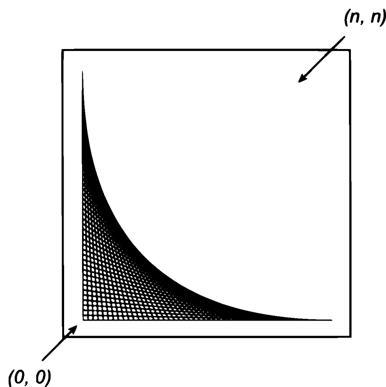
```
StdDraw.setXscale(x0, x1);
StdDraw.setYscale(y0, y1);
```

назначает *ограничивающий прямоугольник* изображения — систему координат, в которой левый нижний угол имеет координаты (x_0, y_0), а правый верхний угол — (x_1, y_1). Масштабирование — простейшее из преобразований, часто встречающихся в компьютерной графике. В примерах этой главы оно используется тривиально для приведения изображения в соответствие с данными.

Перо имеет круглую форму, чтобы концы линий закруглялись; когда вы назначаете радиус пера `r` и рисуете точку, на экране рисуется круг с радиусом `r`. Радиус пера по умолчанию равен 0,002 независимо от масштабирования координат. Это значение соответствует приблизительно 1/500 ширины окна по умолчанию; таким

образом, если нарисовать 200 точек, равномерно распределенных вдоль горизонтальной или вертикальной линии, отдельные круги будут видны, но при рисовании 250 точек результ будет выглядеть как сплошная линия. Вызывая метод `StdDraw.setPenRadius(0.01)`, вы фактически говорите, что толщина отрезков и размер точек должны быть в 5 раз больше стандартного значения 0,002.

```
int n = 50;
StdDraw.setXscale(0, n);
StdDraw.setYscale(0, n);
for (int i = 0; i <= n; i++)
   StdDraw.line(0, n-i, i, 0);
```



Масштабирование для перевода в целочисленные координаты

Фильтрация данных

Одно из простейших применений стандартной графики — построение графика с преобразованием («фильтрацией») стандартного ввода в стандартную графику. Программа `PlotFilter` (листинг 1.5.5) представляет собой как раз такой фильтр: она читает из стандартного ввода серию точек, определяемых координатами (x, y), и выводит эти точки в окне. Программа предполагает, что первые четыре числа в стандартном вводе задают ограничивающий прямоугольник, чтобы диаграмму можно было масштабировать без лишнего перебора точек для определения масштаба. Графическое представление точек получается гораздо более выразительным (и более компактным), чем сами числа. Изображение, построенное программой из листинга 1.5.5, существенно упрощает визуальную оценку свойств точек по сравнению со списком координат (например, карта с обозначенными на ней городами дает наглядное представление о плотности населения). В различных ситуациях, связанных с обработкой данных, касающихся объектов реального мира, графическое изображение почти всегда оказывается самым содержательным способом представления данных. Программа `PlotFilter` показывает, как легко строятся такие изображения.

Листинг 1.5.5. Преобразование стандартного ввода в графику

```
public class PlotFilter
{
   public static void main(String[] args)
   {
      // Масштабирование по первым четырем значениям.
      double x0 = StdIn.readDouble();
      double y0 = StdIn.readDouble();
      double x1 = StdIn.readDouble();
      double y1 = StdIn.readDouble();
      StdDraw.setXscale(x0, x1);
      StdDraw.setYscale(y0, y1);
     // Чтение координат и вывод точек средствами стандартной графики.
      while (!StdIn.isEmpty())
      {
         double x = StdIn.readDouble();
         double y = StdIn.readDouble();
         StdDraw.point(x, y);
      }
   }
}
```

x0	левая граница
y0	нижняя граница
x1	правая граница
y1	верхняя граница
x, y	текущая точка

Программа читает серию точек из стандартного ввода и выводит их средствами стандартной графики. (Предполагается, что первые четыре числа содержат минимальные и максимальные координаты x и y.) Файл USA.txt содержит координаты 13 509 городов в Соединенных Штатах.

```
% java PlotFilter < USA.txt
```

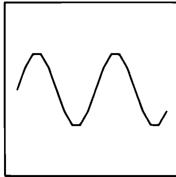
Построение графика функции

Другое типичное применение стандартной графики — отображение экспериментальных данных или значений математической функции. Например, представьте, что вы хотите построить график функции $y = \sin(4x) + \sin(20x)$ в интервале $[0, \pi]$. Эта задача является типичным примером *дискретизации*: интервал содержит беско-

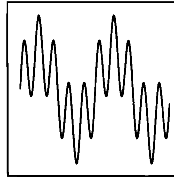
нечное количество точек, но мы ограничиваемся вычислением функции в конечном их множестве. Сначала мы выбираем множество значений x, после чего вычисляем значения y, получая значение функции в каждой из точек x. Затем программа строит график функции, соединяя соседние точки отрезками; результат, построенный этим способом, называется *кусочно-линейной аппроксимацией*.

```
double[] x = new double[n+1];
double[] y = new double[n+1];
for (int i = 0; i <= n; i++)
   x[i] = Math.PI * i / n;
for (int i = 0; i <= n; i++)
   y[i] = Math.sin(4*x[i]) + Math.sin(20*x[i]);
StdDraw.setXscale(0, Math.PI);
StdDraw.setYscale(-2.0, 2.0);
for (int i = 1; i <= n; i++)
   StdDraw.line(x[i-1], y[i-1], x[i], y[i]);
```

n = 20



n = 200



Построение графика функции

Проще всего использовать равномерное распределение значений x. Сначала программа заранее выбирает размер выборки, после чего вычисляет значения x делением интервала на размер выборки. Чтобы все значения гарантированно помещались в видимой части холста, программа выбирает масштаб оси x в соответствии с диапазоном значений x, а масштб оси y — в соответствии с максимальным и минимальным значением функции на этом интервале. Гладкость кривой зависит от свойств функции и размера выборки. Если размер выборки будет слишком малым, построенный график может быть неточным (недостаточно гладким, на нем могут быть пропущены резкие отклонения и т. д.); при слишком большом размере выборки построение графика может занять слишком много времени, так как вычисление некоторых функций достаточно трудоемко. (В разделе 2.4 будет рассмотрен метод построения гладкой кривой без чрезмерного количества точек.) Этот метод позволяет построить график любой функции на ваш выбор: выберите интервал x, на котором будет строиться график, вычислите значения функции в точках, равномерно распределенных по интервалу, определите и вычислите масштаб по оси y, прорисуйте отрезки, образующие график.

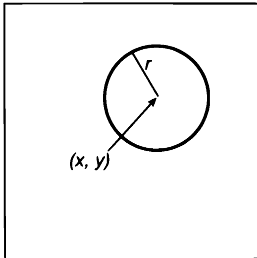
Контуры и закрашенные фигуры

`StdDraw` также включает методы для вычерчивания окружностей, квадратов, прямоугольников и произвольных многоугольников. Если имя метода совпадает

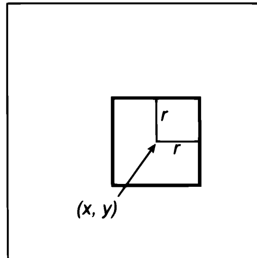
с названием фигуры, метод ограничивается прорисовкой контура. Если же имя начинается с `filled`, то метод выводит фигуру закрашенной. Как обычно, мы приводим таблицу со сводкой доступных методов:

public class StdDraw (*фигуры*)
void circle(double x, double y, double radius)
void filledCircle(double x, double y, double radius)
void square(double x, double y, double r)
void filledSquare(double x, double y, double r)
void rectangle(double x, double y, double r1, double r2)
void filledRectangle(double x, double y, double r1, double r2)
void polygon(double[] x, double[] y)
void filledPolygon(double[] x, double[] y)

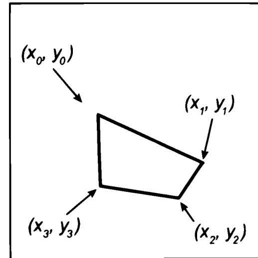
Аргументы методов `circle()` и `filledCircle()` задают круг с радиусом `r` и центром в точке `(x, y)`; аргументы методов `square()` и `filledSquare()` задают квадрат со стороной `2r` и центром в точке `(x, y)`; аргументы `rectangle()` и `filledRectangle()` определяют прямоугольник с шириной `2r`, высотой `2s` и центром `(x, y)`; наконец, аргументы методов `polygon()` и `filledPolygon()` задают последовательность точек, соединяемых отрезками прямой (с добавлением отрезка от последней точки к первой).




```
StdDraw.circle(x, y, r);
```




```
StdDraw.square(x, y, r);
```




```
double[] x = {x0, x1, x2, x3};
double[] y = {y0, y1, y2, y3};
StdDraw.polygon(x, y);
```

Текст и цвет

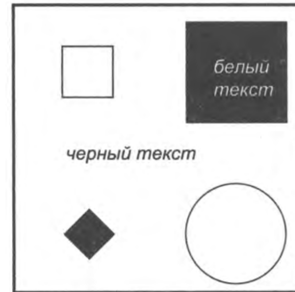
Время от времени появляется необходимость в пометке или цветовом выделении различных элементов изображений. `StdDraw` содержит метод для вывода текста, метод для назначения параметров, связанных с текстом, и еще один метод для изменения цвета пера. В книге эти возможности используются довольно редко,

но они могут быть очень полезными, особенно для вывода на экране монитора. Многочисленные примеры использования этих методов представлены на сайте книги.

```
public class StdDraw (команды вывода текста и управления цветом)
    void text(double x, double y, String s)
    void setFont(Font font)
    void setPenColor(Color color)
```

В этом коде `Font` и `Color` — непримитивные типы, о которых вы узнаете в разделе 3.1 (до того времени оставим подробности `StdDraw`). Поддерживаются следующие цвета пера: **BLACK**, **BLUE**, **CYAN**, **DARK_GRAY**, **GRAY**, **GREEN**, **LIGHT_GRAY**, **MAGENTA**, **ORANGE**, **PINK**, **RED**, **WHITE**, **YELLOW** и **BOOK_BLUE**; все они определены в виде констант `StdDraw`. Например, вызов `StdDraw.setPenColor(StdDraw.GRAY)` переключается на серый цвет пера. По умолчанию используется черное перо (`BLACK`). Шрифта по умолчанию в `StdDraw` хватит для большинства создаваемых вами изображений (информация о работе с другими шрифтами размещена на сайте книги). Например, с помощью этих методов можно выделить на графиках функций важные значения; возможно, вы захотите разработать аналогичные методы для пометки других частей изображений.

```
StdDraw.square(.2, .8, .1);
StdDraw.filledSquare(.8, .8, .2);
StdDraw.circle(.8, .2, .2);
double[] xd = { .1, .2, .3, .2 };
double[] yd = { .2, .3, .2, .1 };
StdDraw.filledPolygon(xd, yd);
StdDraw.text(.2, .5, "black text");
StdDraw.setPenColor(StdDraw.WHITE);
StdDraw.text(.8, .8, "white text");
```



Примеры работы с фигурами и текстом

С помощью средств работы с фигурами, цветом и текстом можно создать самые разнообразные изображения — но не увлекайтесь. По стандартам современных графических библиотек наши команды `StdDraw` довольно примитивны, и, скорее всего, для построения особенно эффектного изображения потребуется множество таких вызовов. С другой стороны, выделение цветом важной информации на иллюстрациях часто бывает оправданным, равно как и использование цвета для представления значений данных.

Двойная буферизация

В `StdDraw` реализована поддержка эффективного приема компьютерной графики, который называется *двойной буферизацией*. Если двойная буферизация включена

вызовом enableDoubleBuffering(), весь вывод осуществляется на так называемом *теневом* (*offscreen*) *холсте*, или холсте (контексте) *в памяти*. «Теневой» холст не отображается на экране; он существует только в памяти компьютера. Только при вызове show() изображение переносится с «теневого» холста на экранный холст, где он появляется в окне стандартной графики. По сути, при использовании двойной буферизации все линии, точки, фигуры и текст, которые вы рисуете, собираются в памяти, а потом прорисовываются все сразу по запросу. Двойная буферизация позволяет вам точно управлять тем, *когда* происходит прорисовка.

Одна из причин использовать двойную буферизацию — эффективность при выполнении большого числа операций вывода. Пошаговое отображение сложного рисунка в процессе его построения может оказаться запредельно медленным на многих компьютерах. Например, можно радикально ускорить программу 1.5.5, добавив вызов enableDoubleBuffering() перед циклом while и вызов show() после цикла while. После этого точки появляются на экране все сразу (а не по одной).

Самая важная роль отводится двойной буферизации при создании компьютерной анимации, когда быстрое переключение статических изображений создает иллюзию движения. Такие эффекты позволяют создавать привлекательные, динамичные визуализации научных явлений. Анимация создается повторением следующих четырех шагов:

- очистка «теневого» холста;
- вывод объектов на «теневом» холсте;
- копирование содержимого «теневого» холста на экранный холст;
- небольшая пауза.

Для поддержки первого и последнего шага StdDraw предоставляет три дополнительных метода. Методы clear() стирают содержимое холста, заполняя его белым или другим заданным цветом. Чтобы вы могли управлять видимой скоростью анимации, метод pause() получает аргумент dt и приостанавливает StdDraw на dt миллисекунд, прежде чем продолжать выполнение следующих вызовов.

public class StdDraw (*расширенные управляющие команды*)	
void enableDoubleBuffering()	разрешает двойную буферизацию
void disableDoubleBuffering()	запрещает двойную буферизацию
void show()	копирует содержимое «теневого» холста на экранный холст
void clear()	заполняет холст белым цветом (по умолчанию)
void clear(Color color)	заполняет холст цветом color
void pause(double dt)	ожидает dt миллисекунд

Столкновения шара

Аналог программы «Hello, World» для анимации отображает черный шар, который движется по экрану и отражается от стенок по законам неупругих соударений. Допустим, шар находится в позиции (r_x, r_y) и вы хотите создать иллюзию его перемещения в близлежащую позицию, допустим $(r_x + 0{,}01, r_y + 0{,}02)$. Для этого следует выполнить четыре действия.

- Заполнить «теневой» холст белым цветом.
- Нарисовать черный шар в новой позиции на «теневом» холсте.
- Скопировать содержимое «теневого» холста на экранный холст.
- Сделать небольшую паузу.

Для создания иллюзии движения эти действия повторяются для всей серии положений шага (образующих прямую линию в данном случае). Без двойной буферизации вместо плавной анимации появляется «моргание», которое сопровождает перерисовку изображения шара между черным и белым цветом.

Программа `BouncingBall` (листинг 1.5.6) реализует эту процедуру для создания иллюзии перемещения шара в квадрате 2 × 2, центр которого расположен в начале координат. Текущая позиция шара находится в точке (r_x, r_y), а обновленная позиция на каждом шаге вычисляется прибавлением v_x к r_x и v_y к r_y. Так как пара (v_x, v_y) определяет фиксированное расстояние, на которое шар перемещается за единицу времени, она представляет *скорость*. Чтобы шар оставался в границах окна стандартной графики, эффект столкновения шара со стенами моделируется по законам неупругих столкновений. Этот эффект реализуется легко: когда шар ударяется о вертикальную стену, скорость в направлении x меняется с v_x на $-v_x$, а при ударе о горизонтальную стену скорость в направлении y меняется с v_y на $-v_y$. Конечно, чтобы понаблюдать за движением шара, вам придется загрузить код с сайта книги и запустить его на компьютере. Чтобы изображение лучше смотрелось на печатной странице, мы изменили программу `BouncingBall` и выбрали серый цвет фона, чтобы был лучше виден след от шара в процессе перемещения (см. упражнение 1.5.34).

Стандартная графика в нашей модели программирования — это то, о чем можно сказать «одна картинка стоит тысячи слов». Она представляет естественную абстракцию, которую вы используете для общения ваших программ с внешним миром. С ней вы можете легко строить графики функций и визуальные представления данных, которые часто применяются в науке и технике. Примеры такого рода часто будут встречаться в книге. Не жалейте времени на изучение примеров программ, приведенных на нескольких последних страницах. Также вы найдете много полезных примеров такого рода на сайте книги и в упражнениях; безусловно, вы сможете дать выход своей творческой энергии, применяя `StdDraw` для решения разных задач. Сможете ли вы нарисовать n-конечную звезду? Смоделировать влияние гравитации на перемещения шара? Вы не поверите, насколько просто решаются эти и другие подобные задачи.

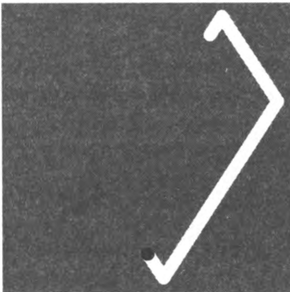
Листинг 1.5.6. Столкновения шара

```
public class BouncingBall
{
   public static void main(String[] args)
   {  // Моделирование движения шара.
      StdDraw.setXscale(-1.0, 1.0);
      StdDraw.setYscale(-1.0, 1.0);
      double rx = 0.480, ry = 0.860;
      double vx = 0.015, vy = 0.023;
      double radius = 0.05;
      StdDraw.enableDoubleBuffering();
      while(true)
      {  // Обновление позиции шара и его перерисовка.
         if (Math.abs(rx + vx) + radius > 1.0) vx = -vx;
         if (Math.abs(ry + vy) + radius > 1.0) vy = -vy;
         rx += vx;
         ry += vy;
         StdDraw.clear();
         StdDraw.filledCircle(rx, ry, radius);
         StdDraw.show();
         StdDraw.pause(20);
      }
   }
}
```

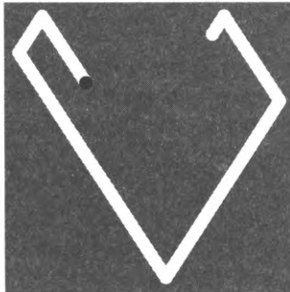
`rx, ry`	позиция
`vx, vy`	скорость
`dt`	время ожидания
`radius`	радиус шара

Программа моделирует движение шара в поле с координатами от −1 до +1. Шар отталкивается от границ по законам неупругих соударений. Двадцатимиллисекундная задержка в StdDraw.pause() сохраняет на экране изображение черного шара, хотя большинство его пикселов меняет цвет с черного на белый и наоборот. Нижние иллюстрации, на которых показана траектория шара, получены в результате выполнения измененной версии этого кода (см. упражнение 1.5.34).

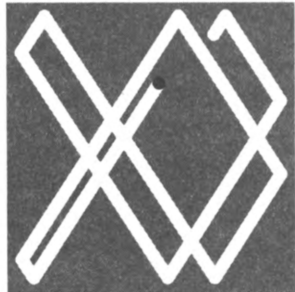
100 шагов



200 шагов



500 шагов



В следующей таблице приведена сводка методов `StdDraw`, рассмотренных в этом разделе:

```
public class StdDraw
```

методы графического вывода

```
void line(double x0, double y0, double x1, double y1)
void point(double x, double y)
void circle(double x, double y, double radius)
void filledCircle(double x, double y, double radius)
void square(double x, double y, double radius)
void filledSquare(double x, double y, double radius)
void rectangle(double x, double y, double r1, double r2)
void filledRectangle(double x, double y, double r1, double r2)
void polygon(double[] x, double[] y)
void filledPolygon(double[] x, double[] y)
void text(double x, double y, String s)
```

управляющие методы

`void setXscale(double x0, double x1)`	устанавливает для оси x диапазон координат (x0, x1)
`void setYscale(double y0, double y1)`	устанавливает для оси y диапазон координат (y0, y1)
`void setPenRadius(double radius)`	устанавливает радиус пера radius
`void setPenColor(Color color)`	назначает цвет пера color
`void setFont(Font font)`	выбирает шрифт font
`void setCanvasSize(int w, int h)`	создает в окне холст w на h
`void enableDoubleBuffering()`	разрешает двойную буферизацию
`void disableDoubleBuffering()`	запрещает двойную буферизацию
`void show()`	копирует содержимое резервного холста на экранный холст
`void clear(Color color)`	заполняет холст цветом color
`void pause(int dt)`	ожидает dt миллисекунд
`void save(String filename)`	сохраняет изображение в файле .jpg или .png

Примечание: одноименные методы без аргументов возвращают значения по умолчанию.

API нашей библиотеки статических методов стандартной графики

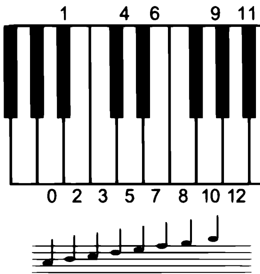
Стандартный звук

Наконец, как последнюю абстракцию для ввода-вывода мы рассмотрим `StdAudio` — библиотеку для воспроизведения, преобразования и синтезирования звука. Скорее всего, вы уже использовали свой компьютер для обработки музыки, а теперь вы научитесь писать собственные программы для работы со звуком. Одновременно вы познакомитесь с некоторыми концепциями, лежащими в основе уже традиционной и очень важной области компьютерных технологий и научных вычислений:

обработки цифровых сигналов. Сейчас мы затронем лишь малую часть этой увлекательной темы, но вас может удивить простота базовых концепций.

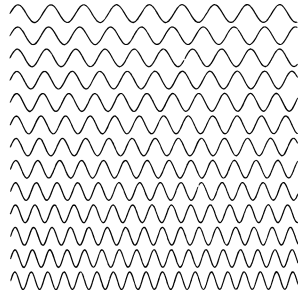
Основные понятия

Звук представляет собой восприятие вибрации молекул, в том числе и наших барабанных перепонок. Следовательно, колебательные процессы играют ключевую роль для понимания звука. Начнем с рассмотрения музыкальной ноты «ля» первой октавы, известной как «*концертное ля*». Эта нота представляет собой не что иное, как синусоидальную волну с частотой колебаний 440 раз в секунду. Функция $sin(t)$ повторяется через каждые 2π единиц, поэтому если построить график функции $\sin(2\pi t \times 440)$, вы получите кривую, которая колеблется 440 раз в секунду. Когда вы играете ноту «ля», дергая гитарную струну, прогоняя воздух через трубу или вызывая вибрацию небольшого конуса в динамике, эта синусоидальная волна является основной частотой звука, который вы слышите и распознаете как «концертное ля». Частота измеряется в герцах (количестве колебаний в секунду). Увеличение или уменьшение частоты вдвое соответствует смещению на *октаву* музыкального звукоряда вверх или вниз. Например, нота с частотой 880 Гц расположена октавой выше «концертного ля», а нота с частотй 110 Гц — двумя октавами ниже. Человеческий слух способен воспринимать частоты в диапазоне от 20 до 20 000 Гц. Амплитуда (значение y) соответствует громкости звука. Мы строим наши графики в интервале от -1 до $+1$ и предполагаем, что на любых устройствах записи и воспроизведения звука будет выполнено необходимое масштабирование, а дальнейшее управление масштабированием осуществляется поворотом регулятора громкости.



нота	i	частота
A	0	440.00
A♯ or B♭	1	466.16
B	2	493.88
C	3	523.25
C♯ or D♭	4	554.37
D	5	587.33
D♯ or E♭	6	622.25
E	7	659.26
F	8	698.46
F♯ or G♭	9	739.99
G	10	783.99
G♯ or A♭	11	830.61
A	12	880.00

$440 \times 2^{i/12}$



Ноты, частоты и волны

Другие ноты

Другие ноты хроматического звукоряда определяются простой математической формулой. Хроматический звукоряд состоит из 12 нот, равномерно распределенных на логарифмической шкале (основание 2). Чтобы узнать частоту i-й ноты над заданной нотой, следует умножить частоту последней на 2 в $(i/12)$-й степени. Другими

словами, частота каждой ноты хроматического звукоряда в точности равна частоте предыдущей ноты звукоряда, умноженной на 2 в степени 1/12 (около 1,06). И этой информации достаточно для создания музыки! Например, чтобы воспроизвести мелодию «Братец Яков», следует воспроизвести каждую из нот «ля», «си», «до-диез» и «ля», проигрывая синусоидальную волну подходящей частоты каждой ноты в течение примерно половины секунды, после чего повторить. Именно это и делает основной метод библиотеки `StdAudio`, `StdAudio.play()`.

Дискретизация

Для представления кривой в области цифрового звука используется дискретизация с постоянными интервалами — в точности так же, как при построении графиков функций. Частота дискретизации должна быть достаточно высокой для точного представления кривой[1] — в области цифрового звука часто применяется частота 44 100 Гц. Для ноты «концертное ля» эта частота соответствует представлению каждого периода синусоидальной волны примерно 100 точками. Так как дискретизация выполняется с постоянными интервалами, достаточно вычислять только координаты y точек дискретизации. Все очень просто: звук представляется массивом вещественных чисел (из диапазона от -1 до $+1$). Метод `StdAudio.play()` получает массив в аргументе и воспроизводит звук, представленный этим массивом, на вашем компьютере. Предположим, вы хотите воспроизвести «концертное ля» в течение 10 секунд. На частоте 44100 Гц для этого потребуется массив типа `double` с длиной 441 001. Для заполнения массива используйте цикл `for`, который вычисляет значение функции $\sin(2\pi t \times 440)$ для $t = 0/44100, 1/44100, 2/44100, 3/44100, \ldots, 441000/44100$. После того как массив будет заполнен этими значениями, можно вызывать функцию `StdAudio.play()` как в следующем фрагменте:

```
int SAMPLING_RATE = 44100;        // Частота дискретизации
int hz = 440;                     // Концертное "ля"
double duration = 10.0;           // 10 секунд
int n = (int) (SAMPLING_RATE * duration);
double[] a = new double[n+1];
for (int i = 0; i <= n; i++)
   a[i] = Math.sin(2*Math.PI * i * hz / SAMPLING_RATE);
StdAudio.play(a);
```

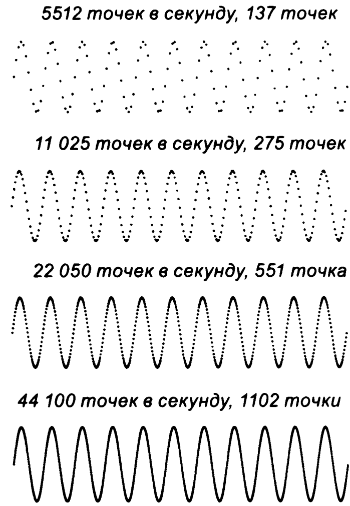
Перед вами своего рода программа «Hello, World» в области цифрового звука. Когда вы добьетесь того, что ваш компьютер воспроизведет эту ноту, вы сможете написать код для воспроизведения других нот и музыки! Создание звука и построение графика синусоиды различаются только устройством вывода. В самом деле, будет полезно и поучительно передать одни и те же числа библиотекам стандартной графики и стандартного звука (см. упражнение 1.5.27).

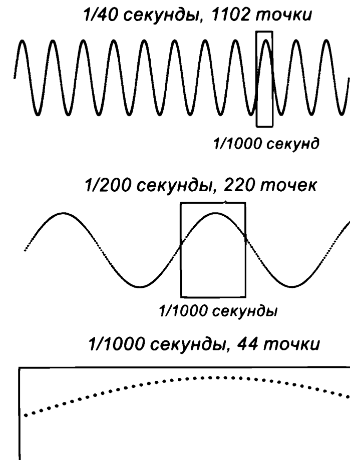
---

[1] Минимально необходимой считается частота дискретизации, вдвое превышающая наибольшую частоту оцифровываемого сигнала, которую нужно передать. Это утверждение носит название *теоремы отсчетов*, или теоремы Котельникова—Найквиста—Уиттакера—Шеннона (см. Вопросы и ответы в конце раздела). — *Примеч. науч. ред.*

Сохранение в файле

Музыка может занимать много места на вашем компьютере. На частоте 44100 Гц четырехминутная песня соответствует 4 × 60 × × 44100 = 10 584 000 чисел. На практике для представления чисел, образующих песню, обычно используется двоичный формат — более компактный по сравнению с последовательностями цифр, используемыми для стандартного ввода и вывода. За прошедшие годы было разработано много таких форматов; `StdAudio` использует формат .wav. Информацию о формате .wav можно найти на сайте книги, но знать все подробности вам пока не обязательно, потому что `StdAudio` берет на себя все преобразования. Наша стандартная библиотека для работы со звуком позволяет читать файлы .wav, записывать файлы .wav и преобразовывать файлы .wav в массивы значений `double` для обработки.





Дискретизация синусоидальной волны

Программа `PlayThatTune` (листинг 1.5.7) наглядно демонстрирует, что библиотека `StdAudio` способна превратить ваш компьютер в музыкальный инструмент. Она получает из стандартного ввода ноты, индексированные по хроматическому звукоряду от «концертного ля», и воспроизводит их посредством стандартного аудиоустройства в системе. Нетрудно представить себе множество расширений этой базовой схемы, часть из которых будет представлена в упражнениях.

Мы включили стандартный звук в базовый арсенал программных инструментов, потому что обработка звука относится к числу важнейших применений вычислительных технологий, которые вам, безусловно, известны. Цифровая обработка сигналов не только приобрела впечатляющее практическое и коммерческое значение, но и оказала огромное влияние на науку и технику — в ней тесно переплетены физика и компьютерные технологии. Другие составляющие цифровой обработки сигналов будут рассмотрены в книге позднее. (Например, в разделе 2.1 рассказано о том, как создавать звуки более музыкальные, чем «чистые» звуки, сгенерированные программой `PlayThatTune`.)

Листинг 1.5.7. Цифровая обработка сигналов

```
public class PlayThatTune
{
   public static void main(String[] args)
   {  // Чтение мелодии из StdIn и ее воспроизведение.
      int SAMPLING_RATE = 44100;
      while (!StdIn.isEmpty())
      {  // Чтение и воспроизведение одной ноты.
         int pitch = StdIn.readInt();
         double duration = StdIn.readDouble();
         double hz = 440 * Math.pow(2, pitch / 12.0);
         int n = (int) (SAMPLING_RATE * duration);
         double[] a = new double[n+1];
         for (int i = 0; i <= n; i++)
            a[i] = Math.sin(2*Math.PI * i * hz / SAMPLING_RATE);
         StdAudio.play(a);
      }
   }
}
```

`pitch`	расстояние от ноты «ля»
`duration`	время воспроизведения ноты
`hz`	частота
`n`	количество точек
`a[]`	данные дискретизации синусоиды

Эта программа превращает ваш компьютер в музыкальный инструмент. Она читает ноты и временные интервалы из стандартного ввода и воспроизводит чистый тон, соответствующий каждой ноте для заданной продолжительности, с помощью стандартного аудиоустройства. Каждая нота задается высотой звука (расстоянием от «концертного ля»). После чтения каждой ноты и продолжительности программа создает массив, выполняя дискретизацию синусоидальной волны и продолжительности на частоте 44 100 точек в секунду, и воспроизводит данные, вызывая метод `StdAudio.play()`.

```
% more elise.txt
7 0.25
6 0.25
7 0.25
6 0.25
7 0.25
2 0.25
5 0.25
3 0.25
0 0.50
```

```
% java PlayThatTune < elise.txt
```

В следующей таблице приведена сводка методов `StdAudio`:

`public class StdAudio`	
`void play(String filename)`	воспроизводит заданный файл .wav
`void play(double[] a)`	воспроизводит заданный набор звуковых данных
`void play(double x)`	воспроизводит данные на 1/44100 секунды
`void save(String filename, double[] a)`	сохраняет данные в файле .wav
`double[] read(String filename)`	читает данные из файла .wav

API нашей библиотеки статических методов стандартного звука

Выводы

Ввод/вывод убедительно демонстрирует силу абстракции, потому что стандартный ввод, стандартный вывод, стандартная графика и стандартный звук могут связываться с разными физическими устройствами в разное время без внесения каких-либо изменений в программы. И даже несмотря на значительные различия между устройствами, мы можем писать программы, способные выполнять операции ввода/вывода без привязки к свойствам конкретных устройств. С этого момента методы `StdOut`, `StdIn`, `StdDraw` и/или `StdAudio` будут использоваться почти во всех программах этой книги. Для экономии места мы будем обозначать эти библиотеки сокращением `Std*`. Одно из важных преимуществ этих библиотек заключается в том, что использование этих библиотек позволяет переключаться на новые устройства — более быстрые, дешевые или вместительные — без каких-либо изменений в программе. В такой ситуации технические подробности соединения обслуживаются операционной системой и реализациями `Std*`. В современных системах в комплект поставки новых устройств обычно включается программное обеспечение, которое решает все проблемы автоматически — как для операционной системы, так и для Java.

Вопросы и ответы

В. Как предоставить доступ к `StdIn`, `StdOut`, `StdDraw` и `StdAudio` программам на языке Java?

О. Если вы выполнили подробные инструкции по установке Java на сайте книги, эти библиотеки должны быть уже доступны Java. Также возможен другой вариант — скопировать файлы StdIn.java, StdOut.java, StdDraw.java и StdAudio.java с сайта книги и поместить их в один каталог с той программой, в которой они используются.

В. Что означает сообщение об ошибке *«Exception in thread "main" java.lang. NoClassDefFoundError: StdIn?»*

О. Библиотека `StdIn` недоступна для Java.

В. Почему вы не используете стандартные библиотеки Java для ввода, графики и звука?

О. Мы их используем, но предпочитаем работать с более простыми абстрактными моделями. Библиотеки Java, на базе которых построены `StdIn`, `StdDraw` и `StdAudio`, создавались для разработки программ, рассчитанных на реальную эксплуатацию, а библиотеки и их API достаточно громоздки. Если вас интересует, как они выглядят, обратитесь к коду в файлах StdIn.java, StdDraw.java и StdAudio.java.

В. Давайте конкретно: если я использую формат `%2.4f` для значения типа `double`, то я получаю две цифры в целой части и четыре цифры в дробной части, верно?

О. Нет, этот формат означает только 4 цифры в дробной части. Первое значение определяет ширину всего поля. Например, формат `%7.2f` определяет формат из 7 символов: 4 цифры в целой части, точка и 2 цифры в дробной части[1].

В. Какие еще существуют коды преобразования для `printf()`?

О. Для целочисленных значений — `o` для восьмеричных значений и `x` для шестнадцатеричных. Также существует множество форматов для даты и времени. За дополнительной информацией обращайтесь на сайт книги.

В. Может ли моя программа заново прочитать данные из стандартного ввода?

О. Нет. Данные можно прочитать только один раз — точно так же, как вы не сможете отменить команду `println()`.

В. Что произойдет, если моя программа попытается прочитать данные из стандартного ввода, когда в нем не останется данных?

О. Произойдет ошибка. Метод `StdIn.isEmpty()`, проверяющий наличие данных, поможет избежать таких ошибок.

В. Почему метод `StdDraw.square(x, y, r)` рисует квадрат с шириной `2*r` вместо `r`?

О. Ради единства с функцией `StdDraw.circle(x, y, r)`, у которой третий аргумент определяет радиус окружности, а не диаметр. В этом контексте `r` обозначает радиус вписанной в квадрат окружности.

В. Мое терминальное окно «зависает» в конце работы программы, использующей `StdAudio`. Что сделать, чтобы программа выходила в режим командной строки без нажатия Ctrl+C?

О. Включите вызов `System.exit(0)` в последнюю строку `main()`. Не спрашивайте зачем[2].

В. Могу ли я использовать отрицательные целые числа для определения нот ниже «концертного ля» при подготовке входных данных для `PlayThatTune`?

О. Да. Вообще говоря, наш выбор совмещения 0 с «концертным ля» был произвольным. В популярном стандарте *MIDI Tuning Standard* нумерация начинается с ноты «до» пятью октавами ниже «концертного ля». По этой схеме «концертному ля» присваивается номер 69, а в использовании отрицательных чисел нет необходимости.

В. Почему я получаю странные результаты при попытке озвучить синусоидальную волну с частотой 30 000 Гц (и более)?

О. *Частота Найквиста*, равная половине частоты дискретизации, определяет наивысшую частоту, которая позволяет воспроизвести сигнал без искажений. Для стандартного звука частота дискретизации равна 44 100 Гц, а частота Найквиста соответственно составляет 22 050 Гц.

---

[1] Если выводимое значение оказалось более длинным, то, как правило, общая ширина поля автоматически увеличивается. — *Примеч. науч. ред.*

[2] Этот метод явно инициирует завершение текущей программы. — *Примеч. науч. ред.*

Упражнения

1.5.1. Напишите программу, которая читает целые числа (столько, сколько введет пользователь) из стандартного ввода и выводит максимальное и минимальное значение.

1.5.2. Измените программу из предыдущего упражнения, чтобы обрабатывались только положительные целые числа (если это требование не выполнено, программа должна напомнить пользователю о том, что числа должны быть положительными).

1.5.3. Напишите программу, которая получает целое число `n` как аргумент командной строки, читает `n` чисел с плавающей точкой из стандартного ввода и выводит их *математическое ожидание* (среднее арифметическое) и *среднеквадратичное отклонение* (квадратный корень из суммы квадратов разностей с математическим ожиданием, деленной на `n-1`).

1.5.4. Доработайте программу из предыдущего управления и создайте фильтр, который читает `n` чисел с плавающей точкой из стандартного ввода и выводит те числа, которые удалены от математического ожидания более чем на 1,5 среднеквадратичного отклонения.

1.5.5. Напишите программу, которая читает из стандартного ввода последовательность целых чисел и выводит число, повторы которого образуют самую длинную непрерывную подпоследовательность, а также длину этой подпоследовательности. Например, для входной последовательности 1 2 2 1 5 1 1 7 7 7 7 1 1 программа должна выдать: «Longest run: 4 consecutive 7s».

1.5.6. Напишите фильтр, который читает последовательность целых чисел и выводит их, исключая смежные повторяющиеся элементы. Например, если на вход поступила последовательность 1 2 2 1 5 1 1 7 7 7 7 1 1 1 1 1 1 1 1, ваша программа должна вывести 1 2 1 5 1 7 1.

1.5.7. Напишите программу, которая получает целое число `n` как аргумент командной строки, читает `n-1` разных целых чисел от 1 до `n` и определяет отсутствующее среди них значение.

1.5.8. Напишите программу, которая читает положительные числа с плавающей точкой из стандартного ввода и выводит их средние геометрические и гармонические значения. *Среднее геометрическое* `n` положительных чисел $x_1, x_2, ..., x_n$ вычисляется по формуле $(x_1 \times x_2 \times ... \times x_n)^{1/n}$. Среднее гармоническое вычисляется по формуле $n/(1/x_1 + 1/x_2 + ... + 1/x_n)$. *Подсказка*: при вычислении среднего геометрического используйте логарифм, чтобы избежать переполнения.

1.5.9. Допустим, файл input.txt содержит две строки — `F` и `F`. Что сделает следующая команда (см. упражнение 1.2.35)?

```
% java Dragon < input.txt | java Dragon | java Dragon
```

```
public class Dragon
{
```

```
    public static void main(String[] args)
    {
        String dragon = StdIn.readString();
        String nogard = StdIn.readString();
        StdOut.print(dragon + "L" + nogard);
        StdOut.print(" ");
        StdOut.print(dragon + "R" + nogard);
        StdOut.println();
    }
}
```

1.5.10. Напишите фильтр `TenPerLine`, который читает из стандартного ввода последовательность целых чисел из диапазона от 0 до 99 и выводит их по 10 чисел в строке с выравниванием столбцов. Затем напишите программу `RandomIntSeq`, которая получает два целочисленных аргумента командной строки `m` и `n` и выводит `n` случайных целых чисел от 0 до `m-1`. Протестируйте свои программы командой `java RandomIntSeq 200 100 | java TenPerLine`.

1.5.11. Напишите программу, которая читает текст из стандартного ввода и выводит количество слов в тексте. В контексте данного упражнения «словом» считается последовательность символов, не являющихся пробельными, заключенная между пробельными символами.

1.5.12. Напишите программу для чтения строк из стандартного ввода. Каждая строка содержит имя и два целых числа. Затем используйте `printf()` для вывода таблицы со столбцами, содержащими имена, целые числа и результат деления первого числа на второе с точностью до трех знаков в дробной части. Например, такая программа может использоваться для вычисления средних оценок студентов.

1.5.13. Напишите программу, которая выводит таблицу ежемесячных платежей, оставшейся суммы и уплаченных процентов по кредиту. В аргументах командной строки должны передаваться три числа: количество лет, сумма кредита и процентная ставка (см. упражнение 1.2.24).

1.5.14. Какая из следующих задач *требует* сохранения всех значений из стандартного ввода (допустим, в массиве), а какая может быть реализована как фильтр с использованием фиксированного количества переменных? В каждом случае входные данные поступают из стандартного ввода и состоят из `n` вещественных чисел в диапазоне от 0 до 1.

- Вывод максимального и минимального числа.
- Вывод суммы квадратов `n` чисел.
- Вывод среднего арифметического `n` чисел.
- Вывод медианы `n` чисел.
- Вывод процента чисел, больших среднего арифметического.
- Вывод `n` чисел в порядке возрастания.
- Вывод `n` чисел в случайном порядке.

1.5.15. Напишите программу, которая получает как аргументы командной строки три числа x, y и z типа `double`, читает из стандартного ввода последовательность координат точек (x_i, y_i, z_i) и выводит координаты точки, ближайшей к (x, y, z). Напомним, что квадрат растояния между точками (x, y, z) и (xi, yi, zi) равен $(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$. По соображениям эффективности не используйте метод `Math.sqrt()`.

1.5.16. Для заданных позиций и масс последовательности объектов напишите программу для вычисления их центра масс — усредненной позиции n объектов, взвешенных пропорционально массам. Если позиции и массы заданы тройками (x_i, y_i, m_i), то центр масс (x, y, m) определяется следующими формулами:

$$m = m_1 + m_2 + \ldots + m_n$$

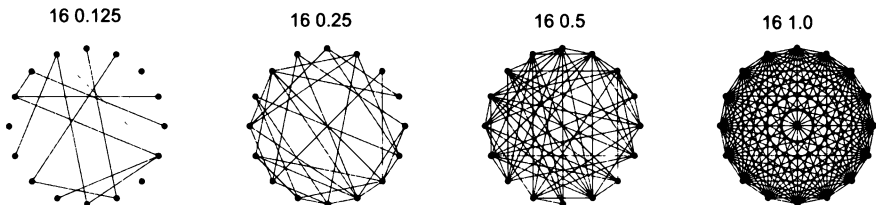
$$x = (m_1 x_1 + \ldots + m_n x_n) / m$$

$$y = (m_1 y_1 + \ldots + m_n y_n) / m$$

1.5.17. Напишите программу, которая читает последовательность вещественных чисел от -1 до $+1$ и выводит их средний модуль, средний квадрат и количество переходов через нуль. *Средний модуль* вычисляется как среднее арифметическое абсолютных значений. Средний квадрат равен среднему арифметическому квадратов значений данных. *Количество переходов через нуль* определяется количеством переходов значений данных из области строго отрицательных значений в область строго положительных значений и наоборот. Эти три статистических показателя широко применяются при анализе цифровых сигналов.

1.5.18. Напишите программу, которая получает целое число `n` в аргументе командной строки и рисует «шахматную доску» из красных и черных квадратов. Левый нижний квадрат окрашен в красный цвет.

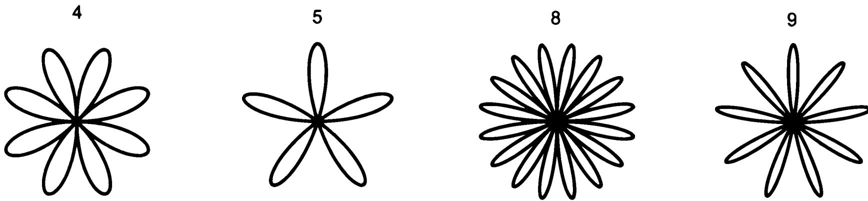
1.5.19. Напишите программу, которая получает в аргументах командной строки целое число `n` и число с плавающей точкой `p` (от 0 до 1), наносит на окружность `n` равномерно распределенных точек, а затем с вероятностью `p` для каждой пары точек рисует серую линию, соединяющую эти точки.



1.5.20. Напишите код для прорисовки символов карточных мастей. Чтобы изобразить символ масти «червы», выведете закрашенный ромб, а затем присоедините к нему два закрашенных полукруга в левой верхней и верхней правой части.

1.5.21. Напишите программу, которая получает целое число `n` как аргумент командной строки и рисует цветок с `n` лепестками (при нечетном `n`) или `2n` лепестками

(при четном n), используя полярные координаты (r, θ) функции $r = \sin(n\ \theta)$ для θ в диапазоне от 0 до 2π радиан.

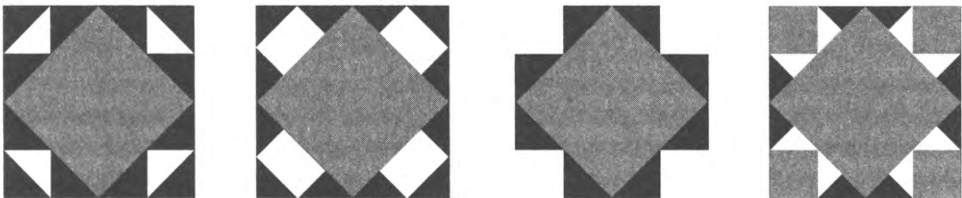


1.5.22. Напишите программу, которая получает строку s как аргумент командной строки и выводит ее на экране в режиме «бегущей строки»: текст двигается слева направо, выходит за правый край экрана и возвращается с левого края. Добавьте второй аргумент командной строки для управления скоростью.

1.5.23. Измените программу `PlayThatTune`, чтобы она получала дополнительные аргументы командной строки для управления громкостью (умножайте значение каждого отсчета на громкость) и темп (умножайте продолжительность каждой ноты на темп).

1.5.24. Напишите программу, которая получает имя файла .wav и скорость воспроизведения r как аргументы командной строки и воспроизводит файл с заданной скоростью. Сначала используйте вызов `StdAudio.read()` для чтения файла в массив `a[]`. Если `r = 1`, воспроизведите `a[]`; в противном случае создайте новый массив `b[]` с размером, равным произведению r на длину `a[]`. Если r < 1, заполните `b[]` непосредственно отсчетами оригинала; если r > 1, заполните `b[]` данными, полученными интерполяцией оригинала. Затем воспроизведите `b[]`.

1.5.25. Напишите программы, использующие `StdDraw` для создания каждого из следующих рисунков.



1.5.26. Напишите программу `Circles`, которая выводит закрашенные круги со случайными радиусами, расположенные в случайных позициях единичного квадрата; результат должен выглядеть примерно так, как показано далее. Ваша программа должна получать четыре аргумента командной строки: количество кругов, вероятность того, что каждый круг окажется черным, минимальный радиус и максимальный радиус.

Упражнения повышенной сложности

1.5.27. *Визуализация аудио.* Измените программу `PlayThatTune`, чтобы воспроизводимые значения передавались библиотеке стандартной графики и вы могли понаблюдать за звуковыми волнами во время воспроизведения. Вам придется поэкспериментировать с отображением нескольких кривых на холсте для синхронизации звука и изображения.

1.5.28. *Статистический опрос.* Во время сбора статистических данных для политических опросов очень важно получить несмещенную выборку зарегистрированных избирателей. Допустим, имеется файл с данными `n` зарегистрированных избирателей, по одному на строку. Напишите фильтр для вывода равномерно распределенной случайной выборки размера `m` (см. листинг 1.4.1).

1.5.29. *Анализ рельефа.* Допустим, рельеф местности представлен двумерной сеткой высот (в метрах). *Пик* представляет собой точку сетки, для которой все четыре соседние ячейки (сверху, снизу, справа и слева) имеют строго меньшие высоты. Напишите программу `Peaks`, которая читает описание рельефа из стандартного ввода, после чего вычисляет и выводит количество пиков в этом рельефе.

1.5.30. *Гистограмма.* Предположим, стандартный поток ввода представляет собой последовательность значений `double`. Напишите программу, которая получает целое число `n` и два вещественных числа `lo` и `hi` как аргументы командной строки и использует `StdDraw` для представления гистограммой количества чисел из стандартного потока ввода, которые попадают в каждый из `n` одинаковых интервалов, полученных делением `(lo, hi)` на `n` интервалов равного размера.

1.5.31. *Спирограф.* Напишите программу, которая получает три аргумента `R`, `r` и `a` типа `double` как аргументы командной строки и рисует *спирограф* (или более точно с технической точки зрения — *эпициклоиду*) — кривую, полученную качением круга с радиусом `r` внутри большего закрепленного круга с радиусом `R`. Если смещение пера от центра катящегося круга равно `(r+a)`, то уравнение получаемой кривой в момент времени `t` определяется формулами

$$x(t) = (R + r)\cos(t) - (r + a)\cos((R + r)t/r)$$

$$y(t) = (R + r)\sin(t) - (r + a)\sin((R + r)t/r)$$

Эти кривые получили известность благодаря популярной игрушке с зубчатыми кольцами разного размера. На кольцах имеются маленькие отверстия; пользователь вставляет в отверстие «перо» (карандаш или ручку) и прокатывает диск по внутренней поверхности большего кольца, чтобы нарисовать кривую.

1.5.32. *Часы.* Напишите программу для анимации секундной, минутной и часовой стрелки обычных стрелочных часов. Используйте вызов `StdDraw.pause(1000)`, чтобы обновлять изображение приблизительно один раз в секунду.

1.5.33. *Осциллограф.* Напишите программу, которая моделирует вывод осциллографа и рисует *фигуры Лиссажу*. Эти фигуры названы по имени французского физика Жюля Антуана Лиссажу, изучавшего траектории точки, совершающей одновременно два гармонических колебания в двух взаимно перпендикулярных направлениях. Предполагается, что колебания являются синусоидальными, так что кривая описывается следующими параметрическими уравнениями:

$$x(t) = A_x \sin (w_x t + \theta_x)$$

$$y(t) = A_y \sin (w_y t + \theta_y)$$

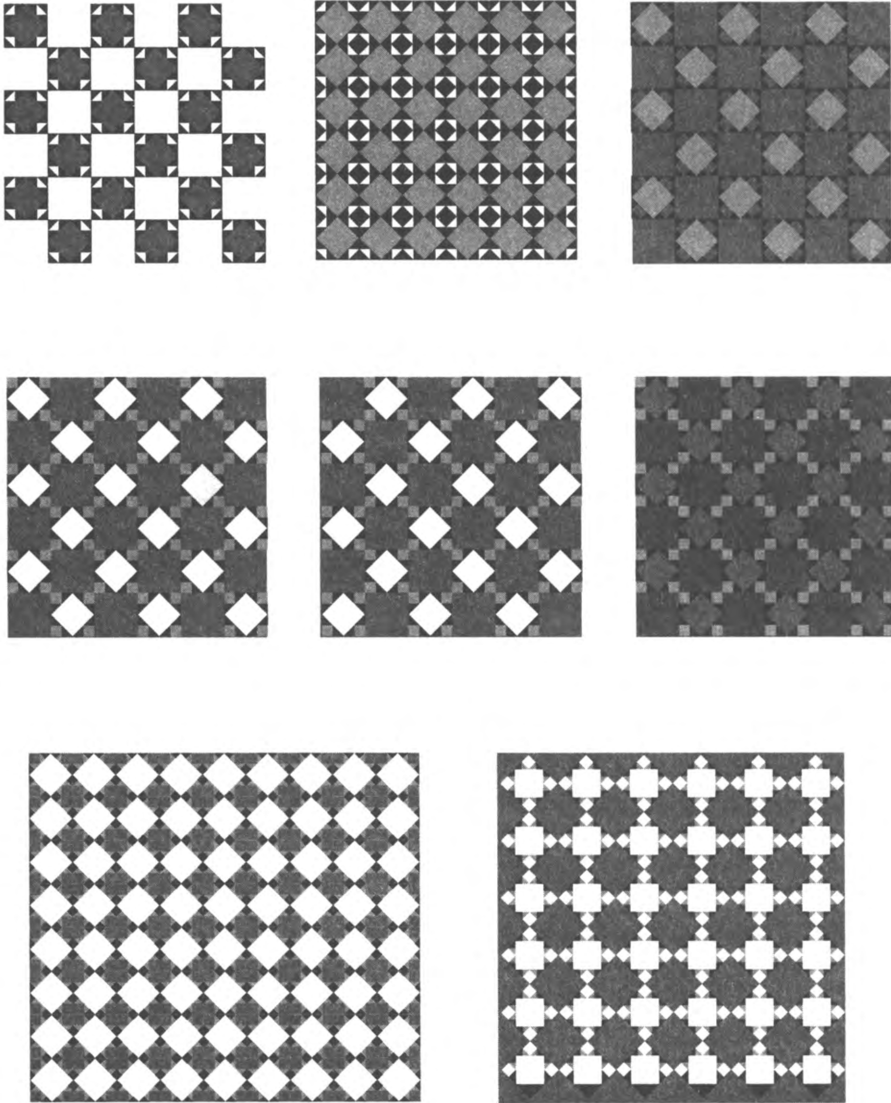
Шесть аргументов A_x, w_x, θ_x, A_y, w_y и θ_y должны передаваться из командной строки.

1.5.34. *Шар с траекторией.* Измените программу `BouncingBall` так, чтобы она создавала изображения наподобие приведенных в тексте, с выводом траектории шара на сером фоне.

1.5.35. *Шар с гравитацией.* Измените программу `BouncingBall` так, чтобы в ней действовала сила тяготения в вертикальном направлении. Добавьте вызовы `StdAudio.play()` для создания звукового эффекта при столкновении шара со «стенками» и другой звуковой эффект для столкновения с «полом».

1.5.36. *Случайные мелодии.* Напишите программу, использующую `StdAudio` для воспроизведения случайных мелодий. Поэкспериментируйте с сохранением тональности, назначением высокой вероятности шагам на целый тон, повторениями и другими правилами для создания более правдоподобных мелодий.

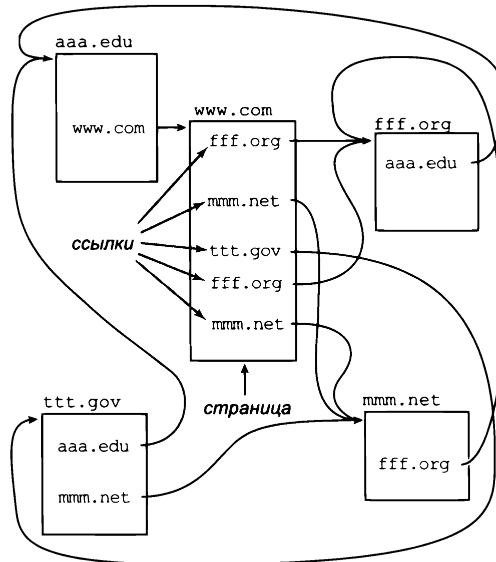
1.5.37. *Мозаика.* Используя ваше решение упражнения 1.5.25, напишите программу `TilePattern`, которая получает целое число `n` как аргумент командной строки и строит узор `n` × `n` с использованием выбранной вами мозаичной плитки. Добавьте второй аргумент командной строки для построения узора «шахматной доски». Добавьте третий аргумент командной строки для выбора цвета. Используйте узоры на следующей странице для построения узора мозаичного пола. Дайте волю творческому воображению! (*Примечание*: такие узоры можно найти во многих античных — и современных — зданиях.)



1.6. Пример: случайный серфинг

Работа в Интернете стала неотъемлемой частью нашей повседневной жизни. Это стало возможным в том числе и благодаря научным исследованиям в области стуктуры Всемирной паутины, которые ведутся с момента ее возникновения. В этом разделе рассматривается простая модель Сети, которая оказалась особенно подходящей для демонстрации некоторых из ее свойств. Различные модификации этой

модели широко встречаются на практике; они стали ключевым фактором бурного роста поисковых приложений в Интернете. Эта модель, известная как *модель случайного серфинга* (или *случайного блуждания на графе*), имеет простое описание. Сеть рассматривается как фиксированное множество веб-страниц; каждая страница содержит фиксированный набор *гиперссылок*, и каждая ссылка ведет на другую страницу. (Для краткости мы будем использовать термины «*страницы*» и «*ссылки*».) Нас интересует, что происходит с веб-пользователем, который случайным образом переходит от страницы к странице — либо вводя адрес в адресной строке браузера, либо щелкая по ссылке на текущей странице. В основе веб-структуры лежит математическая модель, называемая *графом*; она подробно рассматривается в конце книги (см. раздел 4.5).



Страницы и ссылки

Обсуждение работы с графом пока откладывается. Вместо этого мы сосредоточимся на вычислениях, связанных с естественной и хорошо изученной вероятностной моделью, точно описывающей поведение случайного серфинга.

Первым шагом нашего изучения модели случайного серфинга станет ее более точная формулировка. Прежде всего нужно понять, что же означает случайное перемещение от страницы к странице. Оба метода перехода к новой странице отражены в следующем интуитивном правиле 90/10: *предполагается, что в 90% случаев в ходе случайного серфинга пользователь щелкает по случайной ссылке на текущей странице (все ссылки выбираются с равной вероятностью), а в 10% случаев пользователь переходит напрямую к случайной странице (все веб-страницы выбираются с равной вероятностью).*

Сразу видно, что у этой модели есть недостатки, потому что вы по собственному опыту знаете, что поведение реального пользователя намного сложнее.

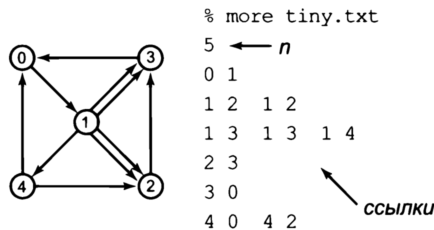
- Ссылки или страницы никогда не выбираются с равной вероятностью.
- Реальной возможности напрямую перейти к любой странице Всемирной паутины не существует.
- Распределение 90/10 (или любое другое фиксированное распределение) — всего лишь предположение.
- В модели не учитывается кнопка Назад или закладки.

Несмотря на все недостатки, модель достаточно богата возможностями, чтобы теоретики могли многое узнать о свойствах Паутины при ее изучении. Чтобы оценить модель, рассмотрите небольшой пример на предыдущей странице. Как вы думаете, на какую страницу пользователь перейдет с наибольшей вероятностью?

Наше поведение в Интернете отчасти напоминает случайный серфинг, поэтому понимание процесса случайного серфинга представляет огромный интерес для людей, строящих веб-инфраструктуру и веб-приложения. Эта модель помогает понять поведение каждого из миллиардов веб-пользователей. В этом разделе мы будем использовать основные средства программирования, описанные в этой главе, для изучения модели и ее следствий.

Формат входных данных

Наше изучение поведения случайного серфинга должно опираться на разные графы, а не на один конкретный пример. А это означает, что наш код должен управляться данными: сами данные хранятся в файлах, а написанная вами программа читает данные из стандартного ввода. На первом шаге этого решения определяется формат ввода, который может использоваться для структурирования информации во входных файлах. Вы можете определить тот формат входных данных, который вам удобен.



Формат входных данных для задачи случайного серфинга

Позднее вы узнаете, как читать веб-страницы в программах Java (раздел 3.1) и как преобразовать имена в числа (раздел 4.4), а также освоите другие приемы эффективной работы с графами. А пока будем считать, что есть n веб-страниц, пронумеро-

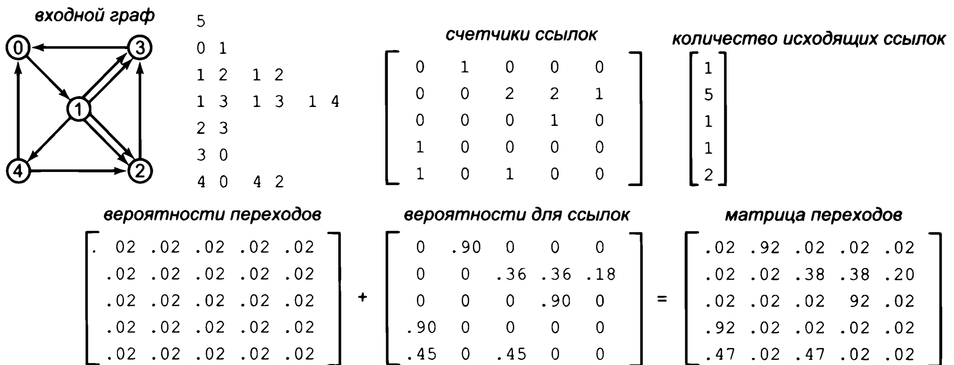
ванных от 0 до `n-1`, а ссылки представляются упорядоченными парами таких чисел: первое число определяет страницу, содержащую ссылку, а второе — страницу, на которую ведет ссылка. С этими соглашениями простейшим форматом ввода для задачи случайного серфинга будет поток с целым числом (значение `n`), за которым следует серия пар целых чисел (представления всех ссылок). `StdIn` интерпретирует последовательность смежных пробельных символов как один разделитель, поэтому вы можете как разместить данные одной ссылки в отдельной строке ввода, так и объединить данные нескольких ссылок в одну строку.

Матрица переходов

Поведение случайного серфинга будет полностью определяться двумерной матрицей, которую мы будем называть *матрицей переходов*. Для `n` веб-страниц создается матрица `n` × `n`, в которой значение в строке `i` и столбце `j` определяет вероятность перехода со страницы `i` на страницу `j` в процессе случайного серфинга. Первая задача — написать код, который создает такую матрицу для любого заданного ввода. По правилу 90/10 вычисления не особо сложны. Задача решается в три шага.

- Прочитать `n` и создать массивы `counts[][]` и `outDegrees[]`.
- Прочитать ссылки и подсчитать ссылки, чтобы элемент `counts[i][j]` содержал количество ссылок с `i` на `j`, а элемент `outDegrees[i]` — количество ссылок с `i` на любые страницы.
- Использовать правило 90/10 для подсчета вероятностей.

Первые два шага выполняются элементарно, третий не намного сложнее: нужно умножить `counts[i][j]` на `0.90/outDegree[i]`, если существует ссылка с `i` на `j` (случайная ссылка выбирается с вероятностью 0,9), а затем прибавить `0.10/n` к каждому элементу (переход к случайной странице с вероятностью 0,1). Программа `Transition` (листинг 1.6.1) выполняет эти вычисления: этот фильтр читает описание графа из стандартного ввода и выводит соответствующую матрицу переходов в стандартный вывод.



Матрица переходов computation

Листинг 1.6.1. Пример: случайный серфинг

```
public class Transition
{
   public static void main(String[] args)
   {
      int n = StdIn.readInt();
      int[][] counts = new int[n][n];
      int[] outDegrees = new int[n];
      while (!StdIn.isEmpty())
      {  // Накопление счетчиков ссылок.
         int i = StdIn.readInt();
         int j = StdIn.readInt();
         outDegrees[i]++;
         counts[i][j]++;
      }

      StdOut.println(n + " " + n);
      for (int i = 0; i < n; i++)
      {  // Вывод распределения вероятностей для строки i.
         for (int j = 0; j < n; j++)
         {  // Вывод вероятности для строки i и столбца j.
            double p = 0.9*counts[i][j]/outDegrees[i] + 0.1/n;
            StdOut.printf("%8.5f", p);
         }
         StdOut.println();
      }
   }
}
```

`n`	количество страниц
`counts[i][j]`	количество ссылок со страницы i на страницу j
`outDegress[i]`	количество ссылок со страницы i куда угодно
`p`	вероятность перехода

Программа представляет собой фильтр, который читает ссылки из стандартного ввода и выдает соответствующую матрицу переходов в стандартный вывод. Сначала программа обрабатывает входные данные и подсчитывает исходящие ссылки для каждой страницы, а затем применяет правило 90/10 для построения матрицы переходов (см. текст). Предполагается, что в матрице на выходе нет ни одной страницы, не имеющей исходящих ссылок (см. упражнение 1.6.3).

```
% more tinyG.txt
5
0 1
1 2  1 2
1 3  1 3  1 4
2 3
3 0
4 0  4 2
```

```
% java Transition < tinyG.txt
5 5
 0.02000 0.92000 0.02000 0.02000 0.02000
 0.02000 0.02000 0.38000 0.38000 0.20000
 0.02000 0.02000 0.02000 0.92000 0.02000
 0.92000 0.02000 0.02000 0.02000 0.02000
 0.47000 0.02000 0.47000 0.02000 0.02000
```

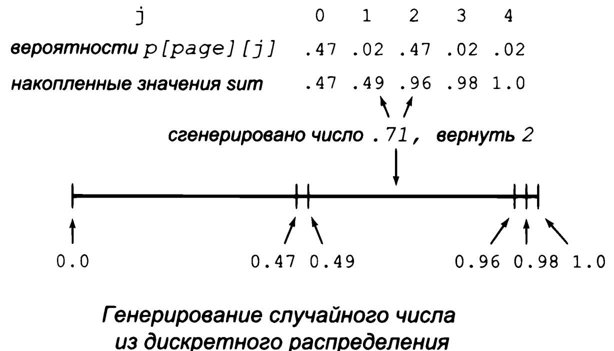
Матрица переходов играет важную роль, потому что каждая строка представляет распределение вероятностей — ее элементы полностью определяют следующее перемещение при случайном серфинге, предоставляя вероятности перехода к каждой из страниц. Обратите внимание: сумма всех элементов равна 1 (то есть пользователь всегда куда-то переходит).

Вывод `Transition` определяет другой формат файлов для матриц: за количеством строк и столбцов следуют значения элементов матрицы, перечисляемые по строкам. Теперь вы можете писать программы, которые читают и обрабатывают матрицы переходов.

Моделирование

Для заданной матрицы переходов моделирование перехода при случайном серфинге требует на удивление небольшого объема кода, как видно из программы `RandomSurfer` (листинг 1.6.2). Программа читает матрицу переходов из стандартного ввода и действует по установленным правилам, начиная со страницы 0 и получая количество перемещений в аргументе командной строки. Программа подсчитывает количество посещений каждой страницы. Деление счетчика на количество перемещений дает оценку вероятности посещения страницы. Эта вероятность известна под названием *ранга страницы*. Иначе говоря, `RandomSurfer` вычисляет оценку рангов всех страниц.

Одно случайное перемещение



Генерирование случайного числа из дискретного распределения

Ключевая роль в вычислениях отводится случайному перемещению, которое задается матрицей переходов. Программа содержит переменную `page`, значение которой определяет текущее местонахождение пользователя (номер текущей страницы). Строка с номером `page` в матрице дает для всех `j` вероятность того, что пользователь далее перейдет к строке с номером `j`. Иначе говоря, когда пользователь находится на странице `page`, нужно сгенерировать случайное число в диапазоне от 0 до `n-1` в соответствии с распределением, заданным строкой `page` в матрице переходов. Как это сделать? Мы воспользуемся приемом *рулеточного выбора*. Метод `Math.random()` может сгенерировать случайное число `r` от 0 до 1, но как это поможет получить случайную страницу? Попробуйте представить вероятности в строке `page` как множество из `n` интервалов в `(0, 1)`, в котором каждая вероятность соответствует длине интервала. Случайная переменная `r` попадает в один из интервалов с вероятностью, которая точно определяется длиной интервала. Эти рассуждения ведут к следующему коду:

```
double sum = 0.0;
for (int j = 0; j < n; j++)
{  // Поиск интервала, содержащего r.
   sum += p[page][j];
   if (r < sum) { page = j; break; }
}
```

Листинг 1.6.2. Моделирование случайного серфинга

```
public class RandomSurfer
{
   public static void main(String[] args)
   {  // Моделирование случайного серфинга.
      int trials = Integer.parseInt(args[0]);
      int n = StdIn.readInt();
      StdIn.readInt();

      // Чтение матрицы переходов.
      double[][] p = new double[n][n];
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            p[i][j] = StdIn.readDouble();

      int page = 0;
      int[] freq = new int[n];
      for (int t = 0; t < trials; t++)
      {  // Одно случайное перемещение к следующей странице.
         double r = Math.random();
         double sum = 0.0;
         for (int j = 0; j < n; j++)
         {  // Поиск интервала, содержащего r.
            sum += p[page][j];
            if (r < sum) { page = j; break; }
         }
         freq[page]++;
      }

      for (int i = 0; i < n; i++)      // Вывод рангов страниц.
         StdOut.printf("%8.5f", (double) freq[i] / trials);
      StdOut.println();
   }
}
```

`trials`	количество перемещений
`n`	количество страниц
`page`	текущая страница
`p[i][j]`	вероятность перехода со страницы i на страницу j
`freq[i]`	количество посещений страницы i

Программа использует матрицу переходов для моделирования поведения случайного серфинга. Она получает количество перемещений в аргументе командной строки, читает матрицу переходов, выполняет заданное количество перемещений и выводит относительную частоту посещений каждой страницы. Ключевая роль в вычислениях принадлежит моделированию случайного перехода к следующей странице.

```
% java Transition < tinyG.txt | java RandomSurfer 100
 0.24000 0.23000 0.16000 0.25000 0.12000
% java Transition < tinyG.txt | java RandomSurfer 1000000
0.27324 0.26568 0.14581 0.24737 0.06790
```

Переменная `sum` отслеживает границы интервалов, определяемых в строке `page`, а цикл `for` находит интервал, содержащий случайное значение `r`. Представьте, что в нашем примере пользователь находится на странице 4. Вероятности переходов равны 0,47, 0,02, 0,47, 0,02 и 0,02, поэтому `sum` принимает значения `0.0`, `0.47`, `0.49`, `0.96`, `0.98` и `1.0`. Эти значения указывают, что вероятности определяют пять ин-

тервалов `(0, 0.47)`, `(0.47, 0.49)`, `(0.49, 0.96)`, `(0.96, 0.98)` и `(0.98, 1)`, по одному для каждой страницы.

Теперь предположим, что вызов `Math.random()` вернул значение `0.71`. Переменная `j` увеличивается с 0 до 1 и 2 и на этом останавливается, поскольку значение `0.71` находится в интервале `(0.49, 0.96)`; следовательно, пользователь направляется на страницу 2. Затем те же вычисления проводятсядля страницы 2 и т. д. Для больших `n` вычисления можно существенно ускорить применением бинарного поиска (см. упражнение 4.2.38). Обычно в таких ситуациях ускорение весьма желательно, потому что нам, как вы вскоре увидите, потребуется очень большое количество случайных перемещений.

Марковские цепи

Случайный процесс, описывающий поведение случайного серфинга, известен под названием *марковской цепи* по имени русского математика Андрея Маркова, разработавшего концепцию в начале XX века. Марковские цепи хорошо изучены, находят широкое практическое применение и обладают рядом замечательных и полезных свойств. Например, вы можете спросить, почему `RandomSurfer` начинает случайный серфинг со страницы 0 — можно было бы ожидать случайного выбора начальной страницы. Предельная теорема для марковских цепей утверждает, что перемещение может начинаться откуда угодно, потому что вероятности того, что случайный серфинг достигнет конкретной страницы, остаются неизменными для всех стартовых страниц! Где бы ни начинался серфинг, процесс в конечном итоге стабилизируется до состояния, в котором дальнейшие перемещения новой информации уже не приносят. Это свойство модели называется *сходимостью* (*mixing*)[1]. На первый взгляд это кажется противоестественным, но именно это объясняет непротиворечивое поведение в ситуации, которая может показаться хаотичной. В текущем контексте это означает, что после достаточно долгого серфинга сеть выглядит более или менее одинаково для всех пользователей. Впрочем, не все марковские цепи обладают свойством сходимости. Например, если убрать из нашей модели случайный переход, некоторые конфигурации веб-страниц могут создать проблемы для пользователя. В самом деле, в Сети существуют группы страниц, известные как *«мышеловки»*, которые доступны для перехода по входным ссылкам, но не имеют выходных. Без случайного перехода пользователь мог бы застрять в «мышеловке» навсегда. Таким образом, главная цель правила 90/10 — обеспечение сходимости модели и исключение подобных аномалий (так называемая *живость* сети).

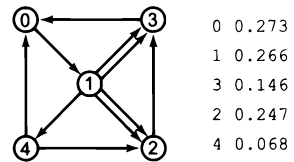
Ранжирование страниц

Модель `RandomSurfer` достаточно прозрачна: она выполняет в цикле заданное количество переходов со случайным перемещением по графу. В сходящейся мо-

[1] В данном случае термин *mixing* подразумевает «перемешивание» системы до однородного стационарного состояния, а *сходимость* — схождение (приведение) ее к этому состоянию, то есть оба термина описывают, по сути, одно и то же. — *Примеч. науч. ред.*

дели увеличение количества итераций повышает точность оценки вероятности попадания пользователя на каждую страницу (ранги страниц). Насколько хорошо эти результаты согласуются с вашей интуитивной догадкой в тот момент, когда вы впервые задумались над вопросом? Возможно, вы предположили, что страница 4 имеет наименьший ранг, но ожидали ли вы, что страницы 0 и 1 будут иметь более высокий ранг, чем страница 3? Если вы хотите знать, какая страница имеет наивысший ранг, понадобится большая точность. Программе `RandomSurfer` потребуется 10^n перемещений для получения ответов с точностью до `n` знаков и много больше перемещений для обеспечения стабильной точности результатов. В нашем примере для получения ответов с точностью до двух знаков после запятой потребуются десятки тысяч, а для трех знаков — миллионы итераций (см. упражнение 1.6.5). Итоговый результат показывает, что страница 0 обгоняет страницу 1: 27,3% против 26,6%. Удивительно, что такое малое различие проявляется в такой маломасштабной задаче: если вы предположили, что страница 0 будет посещаться пользователем с наибольшей вероятностью, вам повезло!

Точные оценки ранга страниц имеют практическую ценность во многих ситуациях. Во-первых, применение таких оценок для упорядочения результатов поиска намного лучше соответствует ожиданиям пользователей, чем предыдущие методы. Во-вторых, они могут служить своего рода мерой достоверности и надежности результатов, что сделало возможным вложение огромных сумм в интернет-рекламу на основании рангов. Даже в нашем крошечном примере ранги страниц могут убедить рекламодателя, что в рекламу на странице 0 стоит вкладывать до четырех раз больше средств, чем в рекламу на странице 4. Вычисление рангов страниц — интересная задача из области обработки данных, которая вдобавок имеет прочную математическую базу и представляет интерес для бизнеса.



Представление рангов страниц на гистограмме

Построение гистограммы

Библиотека `StdDraw` позволяет легко создать визуальное представление, которое дает представление о том, как относительные частоты посещения страниц в ходе случайного серфинга сходятся к рангам страниц. Включите двойную буферизацию, выполните соответствующее масштабирование координат `x` и `y` и добавьте следующий код в цикл случайного перемещения:

```
StdDraw.clear();
for (int i = 0; i < n; i++)
   StdDraw.filledRectangle(i, freq[i]/2.0, 0.25, freq[i]/2.0);
StdDraw.show();
StdDraw.pause(10);
```

Выполните `RandomSurfer` с большим числом опытов. Вы получите графическое представление гистограммы частот, которое в конечном итоге стабилизируется по рангам страниц. Стоит вам воспользоваться этим инструментом хотя бы один раз, и вы будете вспоминать о нем каждый раз, изучая новую модель (возможно, с небольшой корректировкой для более масштабных моделей).

Изучение других моделей

`RandomSurfer` и `Transition` — превосходные примеры программ, управляемых данными. Граф легко определяется файлом, таким как tiny.txt: файл начинается с целого числа `n`, за которым идут пары целых чисел от 0 до `n-1`, представляющие ссылки между страницами. Вы можете опробовать программу на различных моделях данных, как рекомендуется в упражнениях, или создать собственные графы для анализа. Если вас когда-либо интересовало, как работает ранжирование веб-страниц, эти вычисления помогут вам улучшить интуитивное представление о том, почему некоторые страницы ранжируются выше других. Какой странице будет назначен более высокий ранг? Той, которая содержит много ссылок на другие страницы, или же странице с небольшим количеством ссылок? Упражнения этого раздела предоставляют многочисленные возможности для изучения поведения случайного серфинга. Так как программа `RandomSurfer` использует стандартный ввод, вы также можете писать простые программы, генерирующие большие графики, передавать их результаты программам `Transition` и `RandomSurfer` и таким образом изучать поведение модели случайного серфинга на больших графах. Такая гибкость — одна из главных причин для использования стандартного ввода и стандартного вывода.

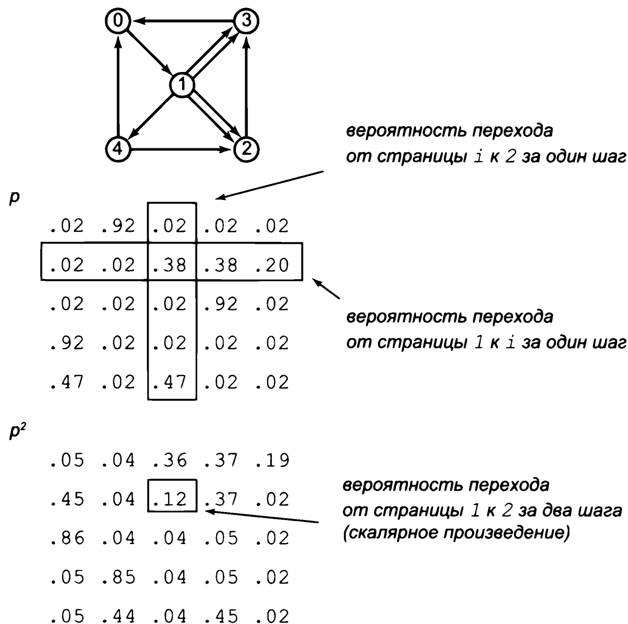
Прямое моделирование поведения случайного серфинга для понимания структуры сети выглядит привлекательно, но у этой модели есть свои недостатки. Задумайтесь над следующим вопросом: можно ли использовать эту модель для вычисления рангов страниц в веб-графе с миллионами (или миллиардами!) страниц и ссылок? Ответ вполне очевиден: нет, потому что для такого большого количества страниц вы даже не сможете сохранить матрицу переходов. Матрица для миллиардов страниц содержит *триллионы* элементов. На вашем компьютере найдется столько памяти? Но можно ли использовать `RandomSurfer` для нахождения рангов страниц для меньшего графа — допустим, из тысяч страниц? Чтобы ответить на этот вопрос, можно провести несколько сеансов моделирования, сохранить результаты для большого количества испытаний, а затем интерпретировать эти экспериментальные данные. Мы используем этот подход во многих научных задачах (один из примеров — задача о разорении игрока; раздел 2.4 посвящен другой такой задаче), но вычисления могут занять очень много времени, так как получение нужной точности потребует огромного количества опытов. Даже в нашем крошечном примере вы видели, что для вычисления рангов с точностью до трех или четырех знаков после запятой необходимы миллионы итераций. Для больших графов необходимое количество итераций для получения точных оценок становится совершенно невообразимым.

Сходимость аналитической модели на основе марковской цепи

Важно помнить, что ранги страниц являются свойством матрицы переходов, не подразумевающим конкретного метода их вычисления. Таким образом, программа RandomSurfer — всего лишь *один из* способов вычисления рангов. К счастью, простая вычислительная модель, основанная на хорошо изученной области математики, предоставляет намного более эффективный метод вычисления рангов страниц. Эта модель использует базовые арифметические операции с двумерными матрицами, которые рассматривались в разделе 1.4.

Метод перемножения

Какова вероятность того, что случайный серфинг приведет со страницы i на страницу j за два перехода? Первый переход идет на промежуточную страницу k, поэтому мы вычисляем вероятность перехода от i на k, а затем от k на j для всех возможных k, после чего суммируем результаты. В нашем примере вероятность перехода от 1 к 2 за два шага равна вероятности перехода 1–0–2 ((0,02 × 0,02) плюс вероятность перехода 1–1–2 (0,02 × 0,38) плюс вероятность перехода 1–2–2 (0,38 × 0,02) плюс вероятность перехода 1–3–2 (0,38 × 0,02) плюс вероятность перехода 1–4–2 (0,20 × 0,47), что дает в сумме 0,1172. Аналогичный процесс повторяется для каждой пары страниц. Но эти вычисления уже встречались вам ранее,



Метод возведения в степень

в определении умножения матриц: элемент на пересечении строки `i` и столбца `j` является результатом скалярного произведения строки `i` и столбца `j` оригинала. Другими словами, результатом умножения `p[][]` на себя будет матрица, в которой элемент на пересечении строки `i` и столбца `j` содержит вероятность перехода от страницы `i` к странице `j` при случайном серфинге. Не жалейте времени на изучение элементов матрицы двухшаговых переходов; это поможет вам лучше понять перемещения при случайном серфинге. Например, самое большое значение в квадрате находится на пересечении строки 2 и столбца 0; оно отражает тот факт, что на странице 2 есть всего одна ссылка, ведущая на страницу 3, на которой также находится только одна выходная ссылка на страницу 0. А следовательно, наиболее вероятным результатом серфинга, начинающегося на странице 2, будет переход к странице 0 за два перехода. Все остальные двухходовые маршруты содержат большее количество вариантов и обладают меньшей вероятностью. Важно заметить, что результат определяется точными вычислениями (до ограничений, накладываемых точностью операций с плавающей точкой в Java), в то время как `RandomSurfer` выдает приблизительный результат, а для повышения точности оценки потребуется больше итераций[1].

Метод возведения в степень

Вероятности для трехшаговых переходов можно вычислить повторным умножением `p[][]`, для четырехшаговых — еще одним и т. д. Однако операции умножения матрицы на матрицу обходятся дорого, а нас в действительности интересует умножение вектора на матрицу. В нашем примере умножение начинается с вектора

```
[1.0 0.0 0.0 0.0 0.0 ]
```

Этот вектор означает, что случайный серфинг начинается со страницы 0. Умножение вектора на матрицу переходов дает вектор

```
[.02 .92 .02 .02 .02 ]
```

с вероятностями того, что пользователь окажется на каждой из страниц после одного шага. Умножение *этого* вектора на матрицу переходов дает вектор

```
[.05 .04 .36 .37 .19 ]
```

который содержит вероятности того, что пользователь окажется на каждой из страниц после двух шагов. Например, вероятность перехода от 0 к 2 за два шага равна вероятности перехода 0–0–2 (0,02 × 0,02) плюс вероятность перехода 0–1–2 (0,92 × 0,38) плюс вероятность перехода 0–2–2 (0,02 × 0,02) плюс вероятность перехода 0–3–2 (0,02 × 0,02) плюс вероятность перехода 0–4–2 (0,02 × 0,47), что дает в сумме 0,36. В этих вычислениях проявляется закономерность: вектор с вероятностями того, что случайный серфинг приведет к каждой странице после

[1] По сути, это два принципиально разных вида моделирования: *аналитическое* и *имитационное*. — *Примеч. науч. ред.*

```
                 ranks[]                          p[][]                    newRanks[]
первый переход
                                          .02 .92 .02 .02 .02
                                          .02 .02 .38 .38 .20
 [ 1.0 0.0 0.0 0.0 0.0 ] *                .02 .02 .02 .92 .02  = [ .02 .92 .02 .02 .02 ]
                                          .92 .02 .02 .02 .02
                                          .47 .02 .47 .02 .02     вероятности перехода
                                                                  от страницы 0 к i за один шаг

второй переход                            вероятности перехода
                                          от страницы i к 2 за один шаг
   вероятности перехода                                            вероятность перехода
   от страницы 0 к i за один шаг          .02 .92 .02 .02 .02     от страницы 0 к 2 за два шага
                                          .02 .02 .38 .38 .20     (скалярное произведение)
 [ .02 .92 .02 .02 .02 ] *                .02 .02 .02 .92 .02  = [ .05 .04 .36 .37 .19 ]
                                          .92 .02 .02 .02 .02
                                          .47 .02 .47 .02 .02     вероятности перехода
                                                                  от страницы 0 к i за два шага
третий переход

   вероятности перехода                   .02 .92 .02 .02 .02
   от страницы 0 к i за два шага          .02 .02 .38 .38 .20
 [ .05 .04 .36 .37 .19 ] *                .02 .02 .02 .92 .02  = [ .44 .06 .12 .36 .03 ]
                                          .92 .02 .02 .02 .02
                                          .47 .02 .47 .02 .02     вероятности перехода
                                                                  от страницы 0 к i за три шага
                                                  .
                                                  .
                                                  .
20-й переход

   вероятности перехода                   .02 .92 .02 .02 .02
   от страницы 0 к i за 19 шагов          .02 .02 .38 .38 .20
 [ .27 .26 .15 .25 .07 ] *                .02 .02 .02 .92 .02  = [ .27 .26 .15 .25 .07 ]
                                          .92 .02 .02 .02 .02
                                          .47 .02 .47 .02 .02     вероятности перехода
                                                                  от страницы 0 к i за 20 шагов
                                                                  (стабильное состояние)
```

Метод возведения в степень для вычисления рангов страниц (предельные значения вероятностей перехода)

Листинг 1.6.3. Аналитическая модель на основе марковской цепи

```
public class Markov
{  // Вычисление рангов страниц после trials перемещений.
   public static void main(String[] args)
   {
      int trials = Integer.parseInt(args[0]);
      int n = StdIn.readInt();
      StdIn.readInt();
      // Чтение матрицы переходов.
      double[][] p = new double[n][n];
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            p[i][j] = StdIn.readDouble();

      // Применение степенного метода для вычисления рангов страниц.
      double[] ranks = new double[n];
      ranks[0] = 1.0;
      for (int t = 0; t < trials; t++)
      {  // Вычисление изменения рангов страниц от следующего шага.
         double[] newRanks = new double[n];
         for (int j = 0; j < n; j++)
         {  //  Новый ранг страницы j вычисляется как скалярное
            // произведение старых рангов и столбца j матрицы p[][].
            for (int k = 0; k < n; k++)
               newRanks[j] += ranks[k]*p[k][j];
         }
         for (int j = 0; j < n; j++)  // Обновление ranks[].
            ranks[j] = newRanks[j];
      }
      for (int i = 0; i < n; i++)     // Вывод рангов страниц.
         StdOut.printf("%8.5f", ranks[i]);
      StdOut.println();
   }
}
```

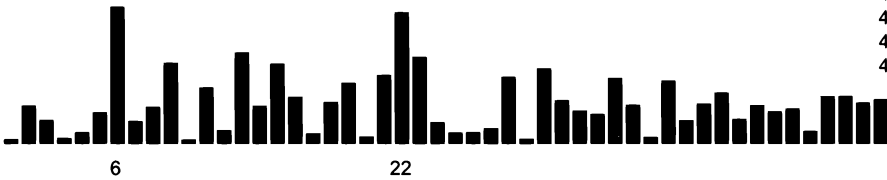
trials	количество шагов
n	количество страниц
p[][]	матрица переходов
ranks[]	ранги страниц
newRanks[]	новые ранги страниц

Программа читает матрицу переходов из стандартного ввода и вычисляет вероятности того, что случайный серфинг приведет к каждой странице (ранги страниц) после количества шагов, заданного в аргументе командной строки.

```
% java Transition < tinyG.txt | java Markov 20
 0.27245 0.26515 0.14669 0.24764 0.06806
% java Transition < tinyG.txt | java Markov 40
 0.27303 0.26573 0.14618 0.24723 0.06783
```

t шагов, в точности равен произведению соответствующего вектора для t-1 шагов и матрицы переходов. Согласно базовой предельной теореме для марковских цепей, процесс сходится к одному вектору независимо от начального состояния; другими словами, после достаточного количества переходов вероятность того, что пользователь окажется на каждой конкретной странице, не зависит от начальной точки. Программа Markov (листинг 1.6.3) содержит реализацию, которая может использоваться для проверки схождения в нашем примере. Например, она получает

0 0.00226
1 0.01681
2 0.00909
3 0.00279
4 0.00572
5 0.01586
6 0.06644
7 0.02092
8 0.01718
9 0.03978
10 0.00200
11 0.02770
12 0.00638
13 0.04452
14 0.01793
15 0.02582
16 0.02309
17 0.00508
18 0.02308
19 0.02562
20 0.00352
21 0.03357
22 0.06288
23 0.04268
24 0.01072
25 0.00473
26 0.00559
27 0.00774
28 0.03738
29 0.00251
30 0.03705
31 0.02340
32 0.01772
33 0.01349
34 0.02363
35 0.01934
36 0.00330
37 0.03144
38 0.01162
39 0.02343
40 0.01677
41 0.02108
42 0.02120
43 0.01627
44 0.02270
45 0.00578
46 0.02343
47 0.02368
48 0.01948
49 0.01579



Ранги страниц с гистограммой для примера большей размерности

те же результаты (ранги с точностью до двух цифр в дробной части), что и программа `RandomSurfer`, но всего с двумя матрично-векторными умножениями — вместо десятков тысяч итераций, необходимых для `RandomSurfer`. Еще 20 умножений обеспечивают результаты с точностью до трех цифр по сравнению с миллионами итераций `RandomSurfer`, и еще несколько итераций дают результаты с максимальной точностью (см. упражнение 1.6.6).

Марковские цепи хорошо изучены, но их влияние на Интернет не было оценено в полной мере вплоть до 1998 года, когда два аспиранта — Сергей Брин и Лоуренс Пейдж — дерзнули построить марковскую цепь для всей сети и вычислить вероятности того, что случайный серфинг посетит каждую ее страницу. Их работа произвела революцию в области веб-поиска и стала основой для метода ранжирования страниц, используемого Google — основанной ими чрезвычайно успешной компании, работающей в первую очередь в области веб-поиска. А если говорить более конкретно, они решили выдавать пользователю список веб-страниц, связанных с их поисковым запросом, по убыванию ранга. Ранги страниц (и связанные с ними методы) в наши дни занимают доминирующее положение, потому что они предоставляют пользователям более релевантные веб-страницы по сравнению с методами, применявшимися ранее (например, упорядочением страниц по количеству входящих ссылок). Вычисление рангов страниц требует огромных затрат времени из-за огромного количества существующих страниц, однако результат оказался чрезвычайно прибыльным, и затраты окупились.

Выводы

Полное понимание модели случайного серфинга выходит за рамки этой книги. Вместо этого мы постарались показать приложение, для которого нужно написать больше кода, чем для коротких программ, которые использовались для представления конкретных концепций. Какие конкретные уроки можно вынести из рассмотренного примера?

У нас имеется полноценная вычислительная модель. Примитивные типы данных и строки, условные и циклические конструкции, массивы, библиотеки стандартного ввода/вывода/графики/звука позволяют решать интересные и очень разнообразные задачи. В самом деле, теория вычислительной техники утверждает, что этой модели достаточно для реализации любых вычислений, которые могут быть выполнены на любом нормальном вычислительном устройстве. В следующих двух главах будут описаны два принципиальных расширения этой модели, направленные на кардинальное сокращение затрат времени и сил при разработке больших и сложных программ.

Код, управляемый данными, доминирует. Концепция использования стандартных потоков ввода и вывода и сохранения данных в файлах открывает множество возможностей. Мы пишем фильтры для преобразования одного вида входных данных к другому типу; генераторы, создающие огромные входные файлы для обработки; программы, способные работать с разными моделями. Мы можем сохранять дан-

ные для архивации или использования в будущем. Мы можем обработать данные, полученные из другого источника (будь то научный прибор или сетевой сайт), и сохранить их в файле. Концепция кода, управляемого данными, предоставляет простые и гибкие средства для поддержки таких операций.

Точность вычислений может быть иллюзорной. Было бы неправильно полагать, что программа выдает абсолютно точные ответы, лишь потому, что она выводит числа с множеством знаков в дробной части. Часто получение точных ответов становится самой сложной проблемой, которую приходится решать в ходе программирования.

Случайные числа с равномерным распределением — всего лишь первый шаг. Говоря о случайном поведении, часто мы подразумеваем нечто более сложное, чем модель «все значения равновероятны», которую нам предоставляет `Math.random()`. Во многих задачах (в том числе и в `RandomSurfer`) задействованы случайные числа с другими законами распределения.

Эффективность важна. Также не следует полагать, что ваш компьютер работает настолько быстро, что он справится с любыми вычислениями. Некоторые задачи требуют существенно больших затрат вычислительных ресурсов по сравнению с другими. Например, метод, использованный в программе `Markov`, намного эффективнее прямого имитационного моделирования поведения случайного серфинга, но и он оказывается слишком медленным для вычисления рангов в огромных веб-графах, встречающихся в реальности. В главе 4 подробно обсуждается тема быстродействия ваших программ, а до этого момента подробное рассмотрение этих вопросов откладывается. Пока запомните, что вам всегда следует хотя бы примерно представлять требования к быстродействию ваших программ.

Возможно, самый важный урок, который следует вынести из написания программ для сложных задач (таких, как пример из этого раздела): *отладка — непростое дело.* Готовые программы, приведенные в книге, скрывают это, но будьте уверены: каждая из них появилась в результате долгого тестирования, исправления ошибок и запуска программ на разных наборах входных данных. Обычно в тексте книги мы стараемся не описывать ошибки и процесс их исправления, потому что в итоге получается скучный отчет, который уделяет излишнее внимание плохому коду, однако вы сможете найти некоторые примеры и описания в упражнениях и на сайте книги.

Упражнения

1.6.1. Измените программу `Transition` так, чтобы вероятность перехода передавалась в аргументе командной строки. Используйте модифицированную версию программы для анализа изменения рангов страниц при переходе на правило 80/20 или 95/5.

1.6.2. Измените программу `Transition` так, чтобы подавить эффект множественных ссылок. Другими словами, если с одной страницы ведут сразу несколько

ссылок на другую страницу, рассматривайте их как одну ссылку. Создайте небольшой пример, показывающий, как это изменение может привести к изменению порядка рангов.

1.6.3. Измените программу `Transition` так, чтобы для страниц без исходящих ссылок строки заполнялись значением `1/n`, где `n` — количество столбцов.

1.6.4. Фрагмент кода `RandomSurfer`, генерирующий случайное перемещение, некорректно работает в том случае, если сумма вероятностей строки `p[page]` отлична от 1. Объясните, что происходит в этом случае, и предложите возможное решение проблемы.

1.6.5. Определите (с точностью до порядка) количество итераций, необходимых `RandomSurfer` для вычисления рангов страниц с точностью до 4 и 5 знаков в дробной части для tiny.txt.

1.6.6. Определите количество итераций, необходимых программе `Markov` для вычисления рангов страниц с точностью до 3, 4 и 10 знаков в дробной части для tiny.txt.

1.6.7. Загрузите с сайта книги файл medium.txt (с данными примера с 50 страницами, представленного в этом разделе) и добавьте в него ссылки *со страницы 23* на каждую другую страницу. Посмотрите, как это повлияет на ранги страниц, и объясните результат.

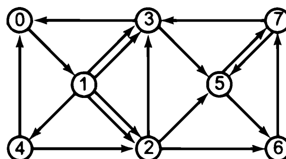
1.6.8. Добавьте в файл medium.txt (см. предыдущее упражнение) ссылки *на страницу 23* с каждой другой страницы. Посмотрите, как это повлияет на ранги страниц, и объясните результат.

1.6.9. Допустим, вы находитесь на странице 23 графа из файла medium.txt. Существует ли ссылка, которую вы можете добавить со своей страницы на другую страницу, чтобы это привело к *повышению* ранга *вашей* страницы?

1.6.10. Предположим, вы находитесь на странице 23 графа из файла medium.txt. Существует ли ссылка, которую вы можете добавить со своей страницы на другую страницу, чтобы это привело к *понижению* ранга *этой* страницы?

1.6.11. Используйте программы `Transition` и `RandomSurfer` для определения рангов страниц восьмистраничного графа, изображенного ниже.

1.6.12. Используйте программы `Transition` и `Markov` для определения рангов страниц восьмистраничного графа, изображенного ниже.



Пример с восемью страницами

Упражнения повышенной сложности

1.6.13. *Возведение матрицы в квадрат.* Напишите программу, похожую на программу `Markov`. Ваша программа должна вычислять ранги страниц многократным возведением матрицы в квадрат, то есть вычислением последовательности p, p^2, p^4, p^8, p^{16} и т. д. Убедитесь в том, что все строки матрицы сходятся к одним и тем же значениям.

1.6.14. *Случайная сеть.* Напишите генератор для `Transition`, который получает как аргументы командной строки количество страниц `n` и количество ссылок `m` и выводит в стандартный поток вывода значение `n`, за которым следуют `m` случайных пар целых чисел от 0 до `n-1`. (Более реалистичные веб-модели рассматриваются в разделе 4.5.)

1.6.15. *Приемники и распределители.* Добавьте в свой генератор из предыдущего упражнения фиксированное количество *приемников*, на которые ведут ссылки с 10% страниц (случайно выбранных), и *распределителей*, с которых ведут ссылки на 10% страниц. Вычислите ранги страниц. У кого ранги выше — у приемников или распределителей?

1.6.16. *Ранги страниц.* Спроектируйте граф, в котором на страницу с наивысшим рангом ведет меньше ссылок, чем на какую-либо другую страницу.

1.6.17. *Интервал посещения. Интервалом посещения* для страницы называется ожидаемое количество переходов между посещениями этой страницы в ходе случайного серфинга. Проведите эксперименты по оценке интервалов посещения для графа tiny.txt, сравните интервалы посещения с рангами страниц, сформулируйте гипотезу относительно их связи и проверьте свою гипотезу на файле medium.txt.

1.6.18. *Время обхода.* Напишите программу, которая оценивает время, необходимое для того, чтобы в ходе случайного серфинга (начинающегося со случайной страницы) каждая страница была посещена хотя бы один раз.

1.6.19. *Графическое моделирование.* Создайте графическую модель, в которой размер точки, представляющей каждую страницу, пропорционален рангу этой страницы. Чтобы программа управлялась данными, разработайте формат файла с включением координат, указывающих, где должна отображаться каждая страница. Протестируйте программу на файле medium.txt.

Глава 2
Функции и модули

Эта глава посвящена конструкции, которая имеет для программной логики не меньшее значение, чем условные переходы и циклы: *функциям*, которые позволяют передавать управление различным фрагментам кода *и возвращаться обратно*. Функции (которые в Java называются *статическими методами*)[1] играют важную роль, потому что они позволяют четко разделить разные задачи в программе и предоставляют обобщенный механизм повторного использования кода.

Функции могут объединяться в независимо компилируемые *модули*. Модули используются для разбиения вычислительных задач на небольшие подзадачи. Позднее в этой главе вы научитесь строить собственные модули и использовать их в стиле программирования, называемом *модульным программированием*.

Некоторые модули разрабатываются именно для того, чтобы их код можно было затем использовать во многих других программах. Такие модули называются *библиотеками*. В частности, в этой главе будут рассмотрены библиотеки для генерирования случайных чисел, анализа данных и ввода/вывода массивов. Библиотеки заметно расширяют набор операций, которые мы используем в своих программах.

Особое внимание уделяется функциям, которые передают управление сами себе (этот процесс называется *рекурсией*). На первый взгляд рекурсия выглядит немного странно, но она позволяет разрабатывать простые программы для решения сложных задач, которые было бы намного труднее реализовать иным способом.

Всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это. Эта мантра неоднократно повторяется в этой главе, а в конце главы будет рассмотрен пример решения сложной задачи за счет разбиения ее на меньшие подзадачи и независимой разработки модулей, взаимодействующих друг с другом для решения подзадач[2].

---

[1] В контексте объектно-ориентированного программирования (и, соответственно, в языке Java) *статическими* принято называть те методы, которые не зависят от конкретного экземпляра какого-либо класса. В понятие *статических данных* вкладывается иной смысл. — *Примеч. науч. ред.*

[2] Такой подход принято называть *декомпозицией* задачи. Он исключительно важен во многих областях техники, а для по-настоящему сложных и масштабных задач является единственным возможным. — *Примеч. науч. ред.*

2.1. Определение функций

В языке Java функции реализуются конструкцией, которая называется *статическим методом*. Модификатор `static` отличает такие методы от тех, что описаны в главе 3, — пока мы будем просто применять его, а суть различий опишем позднее. На самом деле, вы уже использовали статические методы с самого начала книги, от математических функций (таких, как `Math.abs()` и `Math.sqrt()`) до всех методов `StdIn`, `StdOut`, `StdDraw` и `StdAudio`. В каждой написанной вами программе Java присутствует статический метод с именем `main()`. А в этом разделе вы научитесь *определять* собственные статические методы.

В математике *функцией* называется отображение входного значения из некоторой области (*область определения*) на значение из другой области (*область значений*). Например, функция $f(x) = x^2$ отображает 2 на 4, 3 на 9, 4 на 16 и т. д. На первых порах мы работали со статическими методами, реализующими математические функции, потому что последние вам хорошо знакомы. Многие стандартные математические функции реализованы в библиотеке `Math` языка Java, но ученые и инженеры работают с множеством разнообразных математических функций, которые невозможно вместить в одну библиотеку. В начале этого раздела вы научитесь реализовывать такие функции самостоятельно.

Позднее вы узнаете, что возможности статических методов не ограничиваются реализацией математических функций: областью определения или значений статических функций могут быть строки и другие типы, вдобавок они могут генерировать побочные эффекты (например, выводить данные). В этом разделе также рассматривается использование статических методов для структурирования программного кода, а следовательно, упрощения сложных задач программирования.

Статические методы обеспечивают применение ключевой концепции, которая в корне изменит ваш подход к программированию: *всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это*. Мы будем постоянно подчеркивать этот факт в этом разделе и еще неоднократно вернемся к нему в книге. Когда вы пишете текст, вы разбиваете его на абзацы; когда вы пишете программу, разбивайте ее на методы. Разделение большой задачи на несколько меньших задач в программировании гораздо важнее, чем в литературе, потому что оно сильно упрощает *отладку*, *сопровождение* и *повторное использование* кода, а все эти аспекты чрезвычайно важны для разработки качественного программного обеспечения.

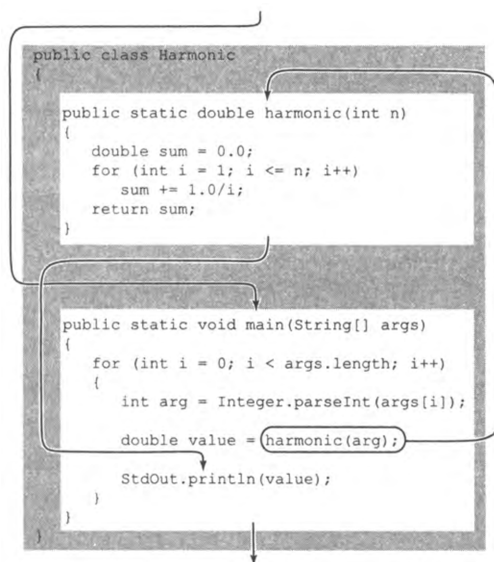
Статические методы

Как вы знаете из опыта использования библиотеки Java `Math`, работать со статическими методами несложно. Например, при включении в программу вызова `Math.`

`abs(a-b)` результат будет таким же, как если бы вы заменили вызов возвращаемым значением, полученным при вызове метода Java `Math.abs()` с передачей выражения `a-b` в аргументе. Такое использование настолько естественно, что вряд ли нуждается в дополнительных пояснениях. Если вас интересует, как система добивается этого, вы увидите, что происходит изменение последовательности выполнения кода программы. Факт изменения программной логики при вызове методов имеет настолько же фундаментальное значение, как и при выполнении условных переходов и циклов.

Чтобы *определить* в файле Java статический метод (кроме метода `main()`), следует указать сигнатуру метода, за которой следуют строки кода, образующие тело метода. Определение вскоре будет рассмотрено более подробно, а пока начнем с простого примера `Harmonic` (листинг 2.1.1) — программы, демонстрирующей роль методов в программной логике. В программу включен статический метод `harmonic()`, который получает целочисленный аргумент `n` и возвращает `n`-е гармоническое число (см. листинг 1.3.5).

Программа 2.1.1 превосходит нашу исходную реализацию вычисления гармонических чисел (листинг 1.3.5), потому что в ней четко разделены две основные операции, выполняемые программой: вычисление гармонических чисел и взаимодействие с пользователем. (Для демонстрационных целей программа 2.1.1 получает несколько аргументов командной строки вместо одного.) Напоминаем: ***всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это.***



Передача управления при вызове статического метода

Листинг 2.1.1. Гармонические числа (обновленная версия)

```
public class Harmonic
{
    public static double harmonic(int n)
    {
        double sum = 0.0;
        for (int i = 1; i <= n; i++)
            sum += 1.0/i;
        return sum;
    }

    public static void main(String[] args)
    {
        for (int i = 0; i < args.length; i++)
        {
            int arg = Integer.parseInt(args[i]);
            double value = harmonic(arg);
            StdOut.println(value);
        }
    }
}
```

sum	накапливаемая сумма
arg	аргумент
value	возвращаемое значение

Программа определяет два статических метода: метод harmonic() получает целочисленный аргумент n и вычисляет n-е гармонические числа (см. листинг 1.3.5), а метод main() тестирует harmonic() с целочисленными аргументами, заданными в командной строке.

```
% java Harmonic 1 2 4
1.0
1.5
2.083333333333333
```

```
% java Harmonic 10 100 1000 10000
2.9289682539682538
5.187377517639621
7.485470860550343
9.787606036044348
```

Передача управления

Хотя программа `Harmonic` внешне похожа на встречавшиеся ранее программы с использованием математических функций, мы подробно разберем ее, чтобы вы поняли, что собой представляет статический метод и как он работает. Программа `Harmonic` состоит из двух статических методов: `harmonic()` и `main()`. Хотя метод `Harmonic` находится в самом начале кода, первой выполняемой строкой в программе Java является первая строка метода `main()`. Следующие команды работают как обычно, если не считать того, что конструкция `harmonic(arg)` (*вызов* статического метода `harmonic()`) передает управление в первую строку кода `harmonic()` из любой точки, где эта конструкция встречается в программе. Кроме того, Java инициализирует *переменную-параметр* (далее просто *параметр*) `n` для `harmonic()` значением `arg` из `main()` на момент вызова. Затем Java выполняет код `harmonic()`, как обычно, пока не обнаружит команду `return`; эта команда возвращает управление в точку кода `main()`, содержащую вызов `harmonic()`. Кроме того, вызов метода `harmonic(arg)` порождает значение — это значение задается в строке `return`. В на-

шем примере это значение переменной `sum` из `harmonic()` на момент выполнения `return`. Далее Java присваивает это *возвращаемое значение* переменной `value`. Конечный результат точно соответствует нашим интуитивным ожиданиям: первое значение присваивается `value` и выводится в виде `1.0` — значение, вычисленное в коде `harmonic()`, когда параметр `n` инициализирован значением 1. Следующее значение присваивается `value` и выводится в виде `1.5` — значение, вычисленное `harmonic()`, когда `n` инициализирован значением 2. Процесс повторяется для каждого аргумента командной строки, а управление передается туда и обратно между `harmonic()` и `main()`.

Трассировка вызова функции

Чтобы лучше понять, как происходит передача управления между вызовами функции, представьте, что каждая функция выводит вначале свое имя и значение аргумента (или аргументов), а непосредственно перед возвратом — свое возвращаемое значение; величина отступа увеличивается при каждом вызове и уменьшается при каждом возврате. Это расширяет возможности трассировки программ по сравнению с простым выводом значений переменных, использовавшимся начиная с раздела 1.2. Отступы наглядно показывают, как происходит передача управления, и помогают убедиться в том, что каждая функция делает именно то, что вы ожидаете. Если вы хотите узнать, как работает программа, обычно вы добавляете в нее вызовы `StdOut.println()`. Если возвращаемые значения соответствуют ожидаемым, то подробная трассировка кода функции становится излишней, и вы избавляетесь от значительного объема работы.

```
i = 0
arg = 1
harmonic(1)
   sum = 0.0
   sum = 1.0
   return 1.0
value = 1.0
i = 2
arg = 2
harmonic(2)
   sum = 0.0
   sum = 1.0
   sum = 1.5
   return 1.5
value = 1.5
i = 3
arg = 4
harmonic(4)
   sum = 0.0
   sum = 1.0
   sum = 1.5
   sum = 1.8333333333333333
   sum = 2.083333333333333
   return 2.083333333333333
value = 2.083333333333333
```

Трассировка вызова функции для команды `java Harmonic 1 2 4`

В оставшейся части этой главы все программы будут связаны с созданием и использованием статических методов, поэтому вам стоит более подробно изучить их базовые свойства. После этого мы изучим несколько примеров реализаций функций и их практических применений.

Терминология

Стоит четко обозначить различия между абстрактными концепциями и механизмами Java для их реализации (конструкция Java `if` реализует ветвление, конструкция `while` реализует цикл и т. д.). В идее математической функции скрываются сразу несколько концепций, каждой из которых соответствует своя конструкция Java

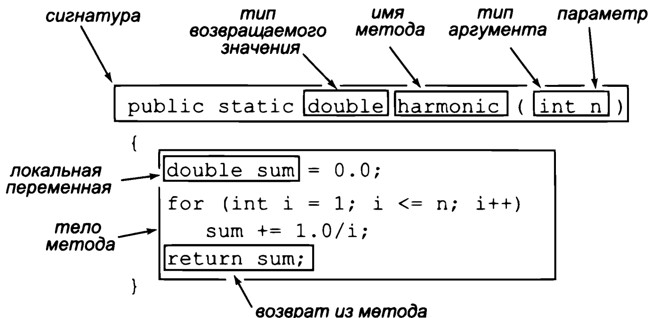
(сводка этих концепций представлена в таблице далее). Хотя эти формальные конструкции служили математикам веками (и программистам — десятилетиями), мы воздержимся от подробного описания следствий этих связей и ограничимся лишь теми аспектами, которые помогут вам в изучении программирования.

концепция	конструкция Java	описание
функция	статический метод	отображение
входное значение	аргумент	входные данные функции
выходное значение	возвращаемое значение	выходные данные функции
формула	тело метода	определение функции
независимая переменная	параметр	символическое имя, представляющее входное значение

Когда мы используем символическое имя в формуле, определяющей математическую функцию (например, $f(x) = 1 + x + x^2$), символическое имя x представляет некоторое входное значение, которое будет подставлено в формулу для определения выходного значения. В языке Java мы называем такое символическое имя *параметром*, а конкретное входное значение, для которого выполняется функция, — *аргументом*.

Определение статического метода

Первая строка определения статического метода — так называемая *сигнатура* — содержит имя метода и имена всех параметров. Также в ней указывается тип каждого параметра и тип возвращаемого значения метода. Сигнатура состоит из ключевого слова `public`; ключевого слова `static`; типа возвращаемого значения; имени метода; последовательности из одного или нескольких типов и имен параметров, разделенных запятыми и заключенных в круглые скобки. Смысл ключевого слова `public` будет рассмотрен в следующем разделе, а смысл ключевого слова `static` — в главе 3. (Формально сигнатура в языке Java включает только имя метода и типы параметров, но мы оставим этот нюанс для специалистов.) За сигнатурой следует



Строение статического метода

тело метода, заключенное в фигурные скобки. Тело состоит из любых конструкций, которые рассматривались в главе 1. Оно также может содержать операцию `return`, которая возвращает управление обратно в точку вызова метода, с передачей результата вычисления или *возвращаемого значения*. В теле функции могут объявляться *локальные переменные*, которые существуют только внутри метода, в котором они были объявлены.

Вызовы функций

Как вы уже видели, вызов статического метода в Java состоит из имени метода, за которым следуют аргументы, разделенные запятыми и заключенные в круглые скобки, — так, как это привычно для записи математических функций. Как упоминалось в разделе 1.2, вызов метода является выражением, поэтому вызовы могут использоваться внутри более сложных выражений. Каждый аргумент также является выражением — Java вычисляет его значение и передает полученный результат методу. Таким образом, вы можете использовать вызовы вида `Math.exp(-x*x/2) / Math.sqrt(2*Math.PI)`, и компилятор Java поймет, что вы имеете в виду[1].

```
for (int i = 0; i < args.length; i++)
{
   arg = Integer.parseInt(args[i]);
   double value = harmonic( arg );
   StdOut.prinln(value);
}
```

аргумент

вызов функции

Строение вызова функции

Множественные аргументы

Статические методы Java, как и математические функции, могут получать несколько аргументов и, следовательно, иметь несколько параметров. Например, следующий статический метод вычисляет длину гипотенузы прямоугольного треугольника с катетами `a` и `b`:

```
public static double hypotenuse(double a, double b)
{  return Math.sqrt(a*a + b*b);  }
```

И хотя параметры в данном случае относятся к одному типу `double`, в общем случае они могут иметь разные типы. Тип и имя каждого параметра объявляются в сигнатуре функции, их объявления разделяются запятыми.

---

[1] Тем не менее следует быть осторожным при таком объединении методов с побочными эффектами: метод будет вырабатывать этот эффект при каждом вызове, что не всегда соответствует желаемому алгоритму. — *Примеч. науч. ред.*

Множественные методы

В файле .java можно определить сколько угодно статических методов. У каждого метода есть тело, состоящее из последовательности строк кода, заключенной в фигурные скобки. Методы существуют независимо друг от друга и могут следовать в файле в произвольном порядке. Статический метод может вызывать любые другие статические методы из того же файла или из библиотеки Java — например, библиотеки `Math`, как показывает следующая пара методов:

```
public static double square(double a)
{  return a*a;  }

public static double hypotenuse(double a, double b)
{  return Math.sqrt(square(a) + square(b));  }
```

Кроме того, как видно из следующего раздела, статический метод также может вызывать статические методы из других файлов .java (при условии, что эти файлы доступны для текущей программы). А в разделе 2.3 рассматриваются вариации непривычной идеи о том, что статический метод может вызывать даже *сам себя*.

Перегрузка

Статические методы с разными сигнатурами являются разными статическими методами. Например, часто требуется определить одну операцию для значений разных числовых типов как в следующем наборе статических методов для вычисления модуля (абсолютного значения):

```
public static int abs(int x)
{
   if (x < 0) return -x;
   else       return  x;
}

public static double abs(double x)
{
   if (x < 0.0) return -x;
   else         return  x;
}
```

Перед вами два разных метода, но между ними достаточно сходства, чтобы им можно было присвоить одно имя (`abs`). Использование одного имени для двух статических методов с разными сигнатурами называется *перегрузкой*; этот прием часто встречается в программировании Java. Например, библиотека Java `Math` использует этот подход для определения реализаций `Math.abs()`, `Math.min()` и `Math.max()` для всех примитивных числовых типов. Также перегрузка часто используется для определения двух разных версий метода, одна из которых при вызове обязательно получает аргумент, а другая может использовать для этого аргумента значение по умолчанию.

Множественные операции возврата

Метод может содержать сколько угодно операций возврата `return`: управление возвращается в точку вызова сразу же при достижении любой из них. Приведенная

ниже функция проверки простых чисел естественным образом определяется с несколькими командами return:

```
public static boolean isPrime(int n)
{
   if (n < 2) return false;
   for (int i = 2; i <= n/i; i++)
      if (n % i == 0) return false;
   return true;
}
```

Вычисление модуля для значения типа int	`public static int abs(int x)`
	`{`
	`   if (x < 0) return -x;`
	`   else return x;`
	`}`

Вычисление модуля для значения типа double	`public static double abs(double x)`
	`{`
	`   if (x < 0.0) return -x;`
	`   else return x;`
	`}`

Проверка, является ли число простым	`public static boolean isPrime(int n)`
	`{`
	`   if (n < 2) return false;`
	`   for (int i = 2; i <= n/i; i++)`
	`      if (n % i == 0) return false;`
	`   return true;`
	`}`

Гипотенуза прямоугольного треугольника	`public static double hypotenuse(double a, double b)` `{ return Math.sqrt(a*a + b*b); }`

Вычисление гармонического числа	`public static double harmonic(int n)`
	`{`
	`   double sum = 0.0;`
	`   for (int i = 1; i <= n; i++)`
	`      sum += 1.0 / i;`
	`   return sum;`
	`}`

Генерирование целого случайного числа с равномерным распределением в диапазоне [0,n)	`public static int uniform(int n)` `{ return (int) (Math.random() * n); }`

Вычерчивание треугольника	`public static void drawTriangle(double x0, double y0,`
	`                                double x1, double y1,`
	`                                double x2, double y2 )`
	`{`
	`   StdDraw.line(x0, y0, x1, y1);`
	`   StdDraw.line(x1, y1, x2, y2);`
	`   StdDraw.line(x2, y2, x0, y0);`
	`}`

Типичный код реализации функций (статических методов)

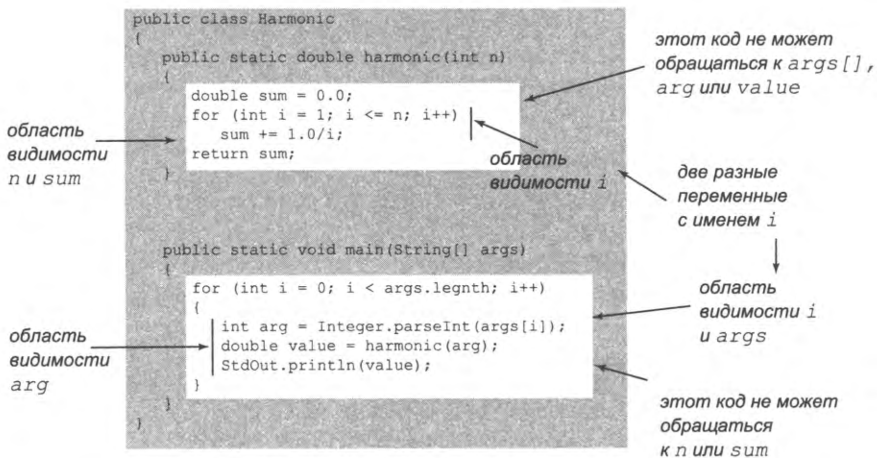
Даже несмотря на присутствие нескольких команд `return`, каждый статический метод возвращает при любом вызове только одно конкретное значение: то, которое указано в той строке `return`, которая будет выполнена первой. Некоторые программисты настаивают на том, что каждый метод должен содержать только одну строку `return`, но мы в этой книге не будем придерживаться столь жестких правил.

Единственное возвращаемое значение

Метод Java передает в вызвавший его код всего одно возвращаемое значение, тип которого объявлен в сигнатуре метода. Это ограничение создает меньше проблем, чем может показаться, потому что типы возвращаемого значения не ограничиваются только примитивными, поэтому методы могут возвращать более сложную и разнообразную информацию. Например, как будет показано позднее в этом разделе, массивы тоже могут использоваться в качестве возвращаемых значений.

Область видимости

Областью видимости переменной называется та часть программы, которая может обращаться к этой переменной по имени. В Java действует общее правило: область видимости переменных, объявленных в блоке, ограничивается строками этого блока[1]. В частности, область видимости переменной, объявленной в статическом методе, ограничивается телом метода. Это означает, что вы не сможете обращаться к переменной, объявленной в одном статическом методе, из другого статического



Область видимости локальных переменных и параметров

[1] Такие переменные с ограниченной областью видимости принято называть *локальными* (по отношению к соответствующему методу или иному блоку). Хороший стиль программирования предполагает использование преимущественно локальных переменных. — *Примеч. науч. ред.*

метода. Если метод содержит меньшие блоки, например тело конструкций `if` или `for`, то область видимости любой переменной, объявленной в одном из этих блоков, ограничивается только этим блоком. В программировании одни и те же имена переменных очень часто используются в различных блоках кода, никак не связанных друг с другом. При этом фактически объявляются разные, не зависящие друг от друга переменные. Например, мы следовали этому принципу при использовании индекса `i` в двух разных циклах `for` одной программы. Один из основополагающих принципов проектирования программных продуктов заключается в том, что каждая переменная должна объявляться в минимально возможной области видимости. В частности, мы используем статические методы еще и потому, что они упрощают отладку за счет ограничения области видимости переменной.

Побочные эффекты

В математике функция отображает одно или несколько входных значений на некоторое выходное значение. В программировании многие функции строятся по одной схеме: они получают один или несколько аргументов, а их единственная цель — вернуть некоторое значение. *Чистой функцией* называется функция, которая для одних и тех же аргументов всегда возвращает одно и то же значение без каких-либо явных побочных эффектов (поглощения вводимых данных, создания выводимых или иного изменения состояния системы). Функции `harmonic()`, `abs()`, `isPrime()` и `hypotenuse()` служат примерами чистых функций.

Однако в программировании также бывают необходимы функции, которые создают побочные эффекты. Собственно, мы часто используем функции, *единственным* смыслом которых является создание побочных эффектов. В языке Java статический метод может использовать ключевое слово `void` вместо возвращаемого типа; это означает, что метод не имеет возвращаемого значения. Явное присутствие `return` в статическом методе с типом `void` необязательно; после того как Java выполнит последнюю строку кода метода, управление возвращается в точку вызова.

Например, статический метод `StdOut.println()` создает полезный побочный эффект: он выводит заданный аргумент в стандартный поток вывода (и не имеет возвращаемого значения). Аналогичным образом следующий статический метод тоже создает побочный эффект — он вычерчивает треугольник средствами библиотеки стандартной графики (и не имеет возвращаемого значения):

```
public static void drawTriangle(double x0, double y0,
                                double x1, double y1,
                                double x2, double y2)
{
   StdDraw.line(x0, y0, x1, y1);
   StdDraw.line(x1, y1, x2, y2);
   StdDraw.line(x2, y2, x0, y0);
}
```

Обычно написание статического метода, который одновременно создает побочные эффекты и возвращает значение, считается признаком плохого стиля программи-

рования. Одно из существенных исключений встречается в функциях, читающих входные данные. Например, метод `StdIn.readInt()` одновременно возвращает прочитанное значение (целое число) и создает побочный эффект (поглощает одно целое число из стандартного потока ввода). В этой книге статические методы `void` используются для двух основных целей:

- для ввода/вывода с использованием библиотек `StdIn`, `StdOut`, `StdDraw` и `StdAudio`;
- для обработки содержимого массива.

Вы использовали статические методы `void` для вывода еще в методе `main()` в первой программе `HelloWorld`; их использование с массивами будет рассмотрено позднее в этом разделе. В Java можно писать методы, обладающие другими побочными эффектами, но мы постараемся не делать этого до главы 3, в которой для этого будет использоваться особый способ, поддерживаемый Java.

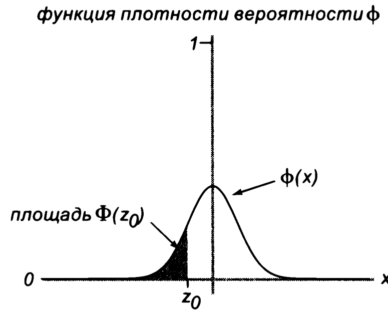
Реализация математических функций

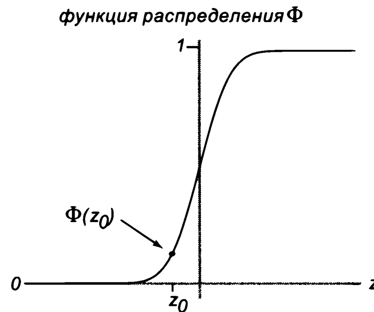
Почему бы просто не использовать методы, уже определенные в языке Java, например `Math.sqrt()`? Конечно, ничто не мешает вам использовать их, когда они существуют. К сожалению, количество разнообразных полезных математических функций огромно, а библиотека содержит лишь небольшой их набор. Когда вы сталкиваетесь с математической функцией, отсутствующей в библиотеке, вам приходится реализовывать соответствующий статический метод.

Для примера рассмотрим код, который мог бы пригодиться в типичной ситуации, актуальной для многих старшеклассников и студентов США. За последний год свыше 1 миллиона студентов сдавали экзамены при поступлении в колледж. Оценки на экзаменах с выбором ответов из нескольких вариантов лежат в диапазоне от 400 (низшая) до 1600 (высшая). Эти оценки используются при принятии важных решений: например, студенты-спортсмены должны набрать не менее 820 баллов, а минимальный порог отбора для некоторых академических стипендий составляет 1500 баллов. Какой процент сдающих экзамены не проходит порог для спортсменов? Какой процент заслуживает получения стипендии?

В области статистики существуют две функции, которые позволят нам получить точные ответы на эти вопросы. *Гауссово* (*нормальное*) распределение случайных величин характеризуется *функцией плотности вероятности* в форме колокола и определяется формулой $\phi(x) = e^{-x/2}/\sqrt{2\pi}$, а *функция распределения* $\Phi(z)$ для гауссовых случайных величин определяется как площадь под графиком $\phi(x)$ над осью x и слева от вертикальной линии $x = z$. Эти функции играют важную роль в науке, технике и финансах, потому что они часто встречаются в точных моделях из реального мира и при этом необходимы для понимания экспериментальной погрешности.

В частности, эти функции точно описывают распределение экзаменационных оценок в нашем примере как функции математического ожидания (среднее ариф-





Гауссово распределение случайных величин

метическое оценок) и среднеквадратичного отклонения (квадратный корень из среднего арифметического суммы квадратов разностей между каждой оценкой и математическим ожиданием). Для математического ожидания μ и среднеквадратичного отклонения σ результатов экзаменов процент студентов с оценками меньше заданного значения z достаточно точно аппроксимируется функцией $\Phi((z - \mu)/\sigma)$. Статические методы для вычисления ϕ и Φ отсутствуют в библиотеке Java `Math`, поэтому нам придется разработать собственные их реализации.

Замкнутая форма

В простейшем случае существует математическая формула в замкнутой форме, определяющая нашу функцию в категориях функций, реализованных в библиотеке. В частности, так обстоит дело для ϕ — в библиотеку Java `Math` включены методы для вычисления функций экспоненты и квадратного корня (и константа для π), поэтому статический метод `pdf()`, соответствующий математическому определению, реализуется достаточно просто (см. программу 2.1.2).

Без замкнутой формы

В остальных случаях для вычисления значений функции приходится использовать более сложный алгоритм. Именно так обстоит дело с вычислением Φ — для этой

Листинг 2.1.2. Гауссовы функции

```
public class Gaussian
{  // Реализация функций гауссова (нормального) распределения.
   public static double pdf(double x)
   {
      return Math.exp(-x*x/2) / Math.sqrt(2*Math.PI);
   }

   public static double cdf(double z)
   {
      if (z < -8.0) return 0.0;
      if (z >  8.0) return 1.0;
      double sum = 0.0;
      double term = z;
      for (int i = 3; sum != sum + term; i += 2)
      {
         sum  = sum + term;
         term = term * z * z / i;
      }
      return 0.5 + pdf(z) * sum;
   }

   public static void main(String[] args)
   {
      double z     = Double.parseDouble(args[0]);
      double mu    = Double.parseDouble(args[1]);
      double sigma = Double.parseDouble(args[2]);
      StdOut.printf("%.3f\n", cdf((z - mu) / sigma));
   }
}
```

sum	накопленная сумма
term	текущее слагаемое

Этот код реализует функцию плотности вероятности (pdf) и функцию распределения (cdf) для гауссова закона распределения, не реализованные в библиотеке Java Math. Реализация pdf() следует прямо из определения, а реализация cdf() использует ряды Тейлора и вызывает pdf() (см. далее текст и упражнение 1.3.38).

```
% java Gaussian 820 1019 209
0.171
% java Gaussian 1500 1019 209
0.989
% java Gaussian 1500 1025 231
0.980
```

функции выражения в замкнутой форме не существует. Такие алгоритмы иногда следуют напрямую из аппроксимаций с использованием рядов Тейлора, но вообще разработка реализаций математических функций с разумной точностью — искусство, к которому следует относиться серьезно, с использованием всех знаний, накопленных математиками за несколько прошедших веков. Было изучено много подходов к вычислению Φ. Например, аппроксимация отношения Φ и ϕ с использованием ряда Тейлора создает эффективную основу для вычисления функции:

$$\Phi(z) = 1/2 + \phi(z)\ (\ z + z^3 / 3 + z^5 / (3 \times 5) + z^7 / (3 \times 5 \times 7) + \ldots\ )$$

Эта формула напрямую преобразуется в код Java статического метода `cdf()` в листинге 2.1.2. Для малых (соответственно больших) z это значение чрезвычайно близко к 0 (соответственно 1), поэтому код просто возвращает 0 (соответственно 1); в противном случае выполняется суммирование ряда Тейлора до схождения суммы.

Выполнение программы `Gaussian` из командной строки с приведенными в примере аргументами (математическое ожидание 1019 и среднеквадратичное отклонение 209) показывает, что около 17% экзаменуемых не проходит порог для спортсменов и всего около 1% заслуживает академической стипендии. В год, когда математическое ожидание было равно 1025, а среднеквадратическое отклонение — 231, лишь около 2% преодолели порог получения академической стипендии[1].

Вычисления с математическими функциями всегда играли центральную роль в науке и технике. Во многих приложениях необходимые функции выражаются в контексте функций библиотеки Java `Math`, как было показано на примере `pdf()`, или же в контексте легко вычисляемых аппроксимаций с использованием рядов Тейлора, как в случае с `cdf()`. Поддержка таких вычислений сыграла центральную роль в эволюции компьютерных систем и языков программирования. Многочисленные примеры такого рода встречаются на сайте книги и в тексте книги.

Использование статических методов для структурирования кода

Помимо вычисления математических функций, процесс отображения значений входных параметров на выходное значение играет важную роль как обобщенный метод структурирования программной логики в любых вычислениях. Это всего лишь частный случай чрезвычайно важного принципа, которым должен руководствоваться каждый хороший программист: *когда вы можете четко разделить задачи в своей программе — сделайте это.*

Функции — естественная и универсальная форма выражения вычислительных задач. В самом деле, то представление программы Java «в перспективе», о котором говорилось в разделе 1.1, было эквивалентно функции: по сути, программа Java рассматривалась как функция, преобразующая аргументы командной строки в выходную строку. Это представление проявляется на многих уровнях вычислений. В частности, длинные программы обычно более естественно выражаются в виде функций, чем в виде последовательности операций присваивания, условных переходов и циклов Java. Это позволяет вам улучшить структуру программ, оформляя в виде функций те действия, для которых это уместно.

---

[1] Как можно заметить, в этом примере программа не обрабатывает весь массив исходных данных, а при известном (заданном) законе и некоторых заранее вычисленных характеристиках распределения случайных величин вычисляет другие его характеристики. — *Примеч. науч. ред.*

Например, программа `Coupon` (листинг 2.1.3) представляет собой модификацию программы `CouponCollector` (листинг 1.4.2) с улучшенным разделением компонентов. Анализируя код из листинга 1.4.2, можно выделить в нем три самостоятельные задачи:

- для заданного *n* вычислить случайное значение;
- для заданного *n* провести эксперимент со сбором купонов;
- получить *n* из командной строки, вычислить и вывести результат.

Программа `Coupon` изменяет структуру кода `CouponCollector`, исходя из построения вычислений на базе этих трех функций. При такой организации кода мы можем изменить `getCoupon()` (например, взять случайные числа из другого распределения) или `main()` (например, взять несколько входных значений или провести несколько экспериментов), не беспокоясь о том, как эти изменения отразятся на работе `collectCoupons()` в целом.

Статические методы изолируют реализации всех компонентов эксперимента друг от друга, или *инкапсулируют* их. Обычно программа состоит из нескольких независимых компонентов; это обстоятельство подчеркивает преимущества от разделения их на разные статические методы. Мы еще рассмотрим эти преимущества подробнее после изучения нескольких других примеров, но вы, безусловно, и так понимаете, что вычисления в программах лучше разбить на функции, подобно тому как основная идея текста лучше выражается при разбиении на абзацы. *Всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это.*

Передача аргументов и возврат значений

А теперь мы более подробно рассмотрим особенности механизма передачи аргументов и возврата значений из функций в языке Java. На концептуальном уровне эти механизмы очень просты, и все же не жалейте времени на то, чтобы досконально разобраться в них, потому что их эффект действительно принципиален. Понимание механизмов передачи аргументов и возвращаемых значений — ключ к изучению любого нового языка программирования.

Передача по значению

Параметры используются в теле функции точно так же, как и локальные переменные. Между параметром и локальной переменной существует только одно различие: Java вычисляет аргумент, переданный в точке вызова, и инициализирует параметр полученным значением. Этот способ передачи называется *передачей по значению*. Метод работает со *значениями* своих аргументов, а не с самими аргументами. Как следствие, изменение значения параметра внутри статического метода не влияет на вызывающий код. (Чтобы не усложнять примеры, мы не будем изменять параметры в коде программ, приводимых в книге.) Некоторые среды программирования предпочитают альтернативный способ, называемый *передачей по ссылке*, при котором метод работает непосредственно с аргументами вызывающего кода.

Листинг 2.1.3. Задача о собирателе купонов (новая версия)

```
public class Coupon
{
   public static int getCoupon(int n)
   {  // Получение случайного целого числа от 0 до n-1.
      return (int) (Math.random() * n);
   }
   public static int collectCoupons(int n)
   {  // Генерирование случайных купонов до того,
      // как будет получен каждый номинал,
      // и возвращение количества собранных купонов.
      boolean[] isCollected = new boolean[n];
      int count = 0, distinct = 0;
      while (distinct < n)
      {
         int r = getCoupon(n);
         count++;
         if (!isCollected[r])
            distinct++;
         isCollected[r] = true;
      }
      return count;
   }

   public static void main(String[] args)
   {  // Генерирование n разных купонов.
      int n = Integer.parseInt(args[0]);
      int count = collectCoupons(n);
      StdOut.println(count);
   }
}
```

`n`	количество номиналов (от 0 до n – 1)
`isCollected[i]`	купон i был получен?
`count`	количество собранных купонов
`distinct`	количество разных собранных купонов
`r`	случайный купон

Эта версия листинга 1.4.2 демонстрирует правильный стиль программирования с использованием инкапсуляции вычислений с помощью статических методов. Этот код работает так же, как CouponCollector, но в нем код лучше разделяется на три компонента: генерирование случайного целого числа от 0 до n–1, проведение эксперимента с собиранием купонов и управление вводом/выводом.

```
% java Coupon 1000
6522
% java Coupon 1000
6481
```

```
% java Coupon 10000
105798
% java Coupon 1000000
12783771
```

Статический метод может получить массив в аргументе или вернуть массив в вызвавший его код. Эта возможность является проявлением объектно-ориентированного характера Java (см. главу 3). Мы рассматриваем ее в текущем контексте, потому что эти базовые механизмы понятны и просты в использовании; с ними вы сможете создавать компактные решения многих задач, естественным образом возникающие при использовании массивов для обработки больших объемов данных.

Массивы как аргументы

Передавая методу в качестве аргумента массив, мы реализуем функцию, которая работает с произвольным количеством однотипных значений. Например, следующий метод вычисляет среднее арифметическое массива значений `double`:

```
public static double mean(double[] a)
{
   double sum = 0.0;
   for (int i = 0; i < a.length; i++)
      sum += a[i];
   return sum / a.length;
}
```

Мы использовали массивы в аргументах с самой первой программы, написанной в книге. Фрагмент

```
public static void main(String[] args)
```

определяет `main()` как статический метод, который получает массив строк в аргументе и не возвращает ничего. Виртуальная машина Java, выполняя программу, собирает в массив строки, введенные после имени программы в команде `java`, и вызывает `main()` с передачей ему этого массива в аргументе. (Большинство программистов присваивают этому параметру имя `args`, хотя подойдет любое имя.) В методе `main()` программа работает с этим массивом точно так же, как с любым другим массивом.

Побочные эффекты при работе с массивами

Статические методы, получающие массив в аргументе, часто создают побочные эффекты — изменяют значения элементов массива. Классический пример такого рода — метод, меняющий местами два элемента. Ниже приведена адаптированная версия кода, представленного в начале раздела 1.4:

```
public static void exchange(String[] a, int i, int j)
{
   String temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```

Эта реализация естественным образом вытекает из представления массивов в языке Java. Параметр метода `exchange()` содержит ссылку на массив, а не копию содержимого массива: когда вы передаете массив в аргументе метода, метод получает возможность изменять элементы исходного массива. Второй классический пример статического метода, который получает массив в аргументе и производит побочный эффект, — случайная перестановка элементов массива с использованием версии алгоритма, описанного в разделе 1.4 (а также методов `exchange()` и `uniform()`, упоминавшихся ранее в этом разделе):

```
public static void shuffle(String[] a)
{
   int n = a.length;
   for (int i = 0; i < n; i++)
      exchange(a, i, i + uniform(n-i));
}
```

нахождение максимума среди элементов массива	`public static double max(double[] a)`
	`{`
	`   double max = Double.NEGATIVE_INFINITY;`
	`   for (int i = 0; i < a.length; i++)`
	`      if (a[i] > max) max = a[i];`
	`   return max;`
	`}`

скалярное произведение	`public static double dot(double[] a, double[] b)`
	`{`
	`   double sum = 0.0;`
	`   for (int i = 0; i < a.length; i++)`
	`      sum += a[i] * b[i];`
	`   return sum;`
	`}`

перестановка двух элементов массива	`public static void exchange(String[] a, int i, int j)`
	`{`
	`   String temp = a[i];`
	`   a[i] = a[j];`
	`   a[j] = temp;`
	`}`

вывод одномерного массива (и его длины)	`public static void print(double[] a)`
	`{`
	`   StdOut.println(a.length);`
	`   for (int i = 0; i < a.length; i++)`
	`      StdOut.println(a[i]);`
	`}`

чтение двумерного массива с элементами типа double по строкам	`public static double[][] readDouble2D()`
	`{`
	`   int m = StdIn.readInt();`
	`   int n = StdIn.readInt();`
	`   double[][] a = new double[m][n];`
	`   for (int i = 0; i < m; i++)`
	`      for (int j = 0; j < n; j++)`
	`         a[i][j] = StdIn.readDouble();`
	`   return a;`
	`}`

Примеры реализаций типичных функций, которые получают массивы в качестве аргументов или возвращают их

В разделе 4.2 будут рассмотрены аналогичные методы сортировки массива (перестановки элементов так, чтобы они располагались в заданном порядке). Во всех этих примерах отражен тот базовый факт, что при передаче массивов в Java используется механизм *передачи по значению* в отношении ссылки на массив

и механизм *передачи по ссылке* в отношении элементов массива. В отличие от аргументов примитивных типов, изменения, вносимые методом в элементы массива, отражаются во всей программе, вызывающей метод. Метод, получающий массив в аргументе, не может изменить адрес массива в памяти, его длину и тип элементов, но метод может изменить содержимое массива — присвоить его элементам другие значения.

Массивы как возвращаемые значения

Метод, который сортирует, переставляет или иным образом изменяет массив, переданный в аргументе, не обязан возвращать ссылку на этот массив, потому что он изменяет содержимое самого переданного массива, а не его копию. Однако во многих ситуациях удобно вернуть из статического метода массив как возвращаемое значение. В первую очередь это статические методы, которые используют массивы для передачи нескольких однотипных значений в вызвавшую метод программу. Например, следующий статический метод создает и возвращает массив с данными, используемыми `StdAudio` (см. листинг 1.5.7): он содержит значения, полученные при дискретизации синусоидальной волны с заданной частотой (в герцах) и продолжительностью (в секундах) на стандартной частоте дискретизации 44 100 Гц.

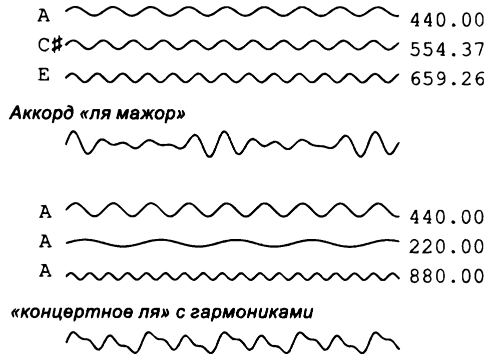
```
public static double[] tone(double hz, double t)
{
   int SAMPLING_RATE = 44100;
   int n = (int) (SAMPLING_RATE * t);
   double[] a = new double[n+1];
   for (int i = 0; i <= n; i++)
      a[i] = Math.sin(2 * Math.PI * i * hz / SAMPLING_RATE);
   return a;
}
```

В этом коде длина возвращаемого массива зависит от продолжительности: для заданной продолжительности `t` длина массива составит приблизительно `44100*t`. С такими статическими методами вы можете писать код, который рассматривает звуковую волну как единое целое (массив, содержащий данные оцифровки), как вы вскоре увидите в листинге 2.1.4.

Пример: наложение звуковых волн

Как упоминалось в разделе 1.5, для создания звука, близкого к звучанию реального музыкального инструмента, наша простая звуковая модель нуждается в улучшении. Существует много возможных улучшений; со статическими методами мы можем систематически применять эти улучшения для получения звуковых волн, намного более сложных, чем простые синусоиды из раздела 1.5. Для демонстрации эффективного использования статических методов при решении интересных вычислительных задач мы рассмотрим программу, которая по своей функциональности практически аналогична программе `PlayThatTune` (листинг 1.5.7), но вносит

дополнительные тона на одну октаву выше и ниже основного для создания более реалистичного звука.



Наложение волн для создания составных звуков

Аккорды и гармоники

Такие ноты, как «концертное ля», представляют собой «чистый» звук, который не воспринимается нами как музыкальный, потому что те звуки, которые мы привыкли слышать, содержат много других компонентов. Звук гитарной струны отражается от деревянной части инструмента, от стен комнаты, в которой вы находитесь, и т. д. Подобные эффекты могут рассматриваться как модификация базовой синусоидальной волны. Например, многие музыкальные инструменты создают дополнительные гармоники (те же ноты в других октавах с меньшей громкостью) и позволяют играть аккорды (несколько нот одновременно). Для объединения нескольких звуков применяется *наложение*: волны просто суммируются, а результат масштабируется, чтобы все значения оставались в диапазоне от –1 до +1. Таким образом, при наложении синусоидальных волн разной частоты можно получить произвольные волны произвольной сложности. Одним из выдающихся достижений математики в XIX веке была формулировка идеи о том, что любая плавная периодическая функция может быть выражена как сумма синусоидальных и косинусоидальных волн, известная как *ряд Фурье*. Эта математическая идея соответствует представлению о том, что мы можем создавать разнообразные звуки при помощи музыкальных инструментов и наших голосовых связок и что все звуки образуются наложением различных периодических кривых. Любой звук соответствует некоторой кривой, любая кривая соответствует некоторому звуку, и мы можем создавать сколь угодно сложные кривые посредством наложения.

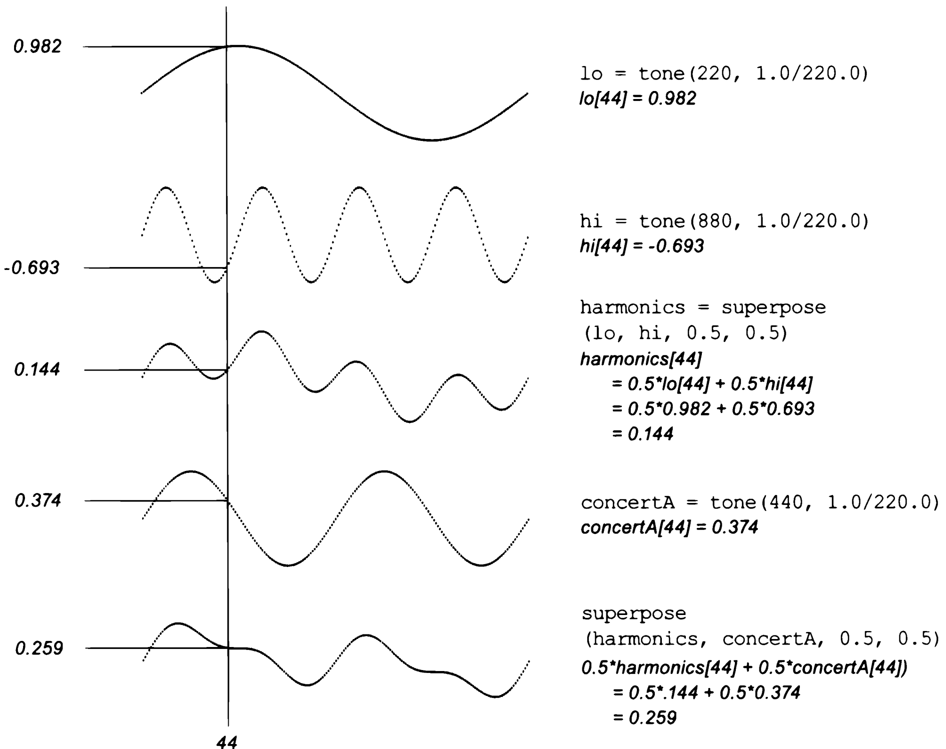
Взвешенное наложение

Так как звуковые волны представляются массивами чисел, определяющими значения в точках дискретизации, наложение реализуется достаточно просто: нужно просуммировать значения во всех точках дискретизации для получения

результата наложения, а затем заново масштабировать их. Для большей гибкости управления наложением каждой из суммируемых волн назначается относительный весовой коэффициент; эти веса имеют положительные значения, а их сумма равна 1. Например, если мы хотим, чтобы первый звук втрое превосходил по эффекту второй звук, первому звуку назначается вес 0,75, а второму — 0,25. Если одна волна представлена массивом `a[]` с относительным весом `awt`, а другая — массивом `b[]` с относительным весом `bwt`, их взвешенная сумма вычисляется следующим кодом:

```
double[] c = new double[a.length];
for (int i = 0; i < a.length; i++)
   c[i] = a[i]*awt + b[i]*bwt;
```

Условия положительности весов и равенстве суммы 1 гарантируют, что при суммировании будет сохранено наше правило о том, что все звуковые значения должны находиться в диапазоне от –1 до +1.



Добавление гармоник к «концертному ля»
(1/220 секунды на частоте дискретизации 44 100 Гц)

Листинг 2.1.4. Программа PlayThatTune (обновленная версия)

```
public class PlayThatTuneDeluxe
{
   public static double[] superpose(double[] a, double[] b,
                                    double awt, double bwt)
   {  // Взвешенное наложение a и b.
      double[] c = new double[a.length];
      for (int i = 0; i < a.length; i++)
         c[i] = a[i]*awt + b[i]*bwt;
      return c;
   }

   public static double[] tone(double hz, double t)
   {  /* см. текст */  }

   public static double[] note(int pitch, double t)
   {  // Воспроизведение ноты заданной высоты с гармониками.
      double hz = 440.0 * Math.pow(2, pitch / 12.0);
      double[] a  = tone(hz, t);
      double[] hi = tone(2*hz, t);
      double[] lo = tone(hz/2, t);
      double[] h  = superpose(hi, lo, 0.5, 0.5);
      return superpose(a, h, 0.5, 0.5);
   }

   public static void main(String[] args)
   {  // Чтение и воспроизведение звука с гармониками.
      while (!StdIn.isEmpty())
      {  // Чтение и воспроизведение ноты с гармониками.
         int pitch = StdIn.readInt();
         double duration = StdIn.readDouble();
         double[] a = note(pitch, duration);
         StdAudio.play(a);
      }
   }
}
```

`hz`	частота
`a[]`	чистый тон
`hi[]`	верхняя гармоника
`lo[]`	нижняя гармоника
`h[]`	тон с гармониками

Этот код улучшает звуки, создаваемые программой из листинга 1.5.7; для этого он создает гармоники при помощи статических методов, отчего звук становится более реалистичным по сравнению с чистым тоном.

```
% more elise.txt
7 0.25
6 0.25
7 0.25
6 0.25
...
```

```
% java PlayThatTuneDeluxe < elise.txt
```

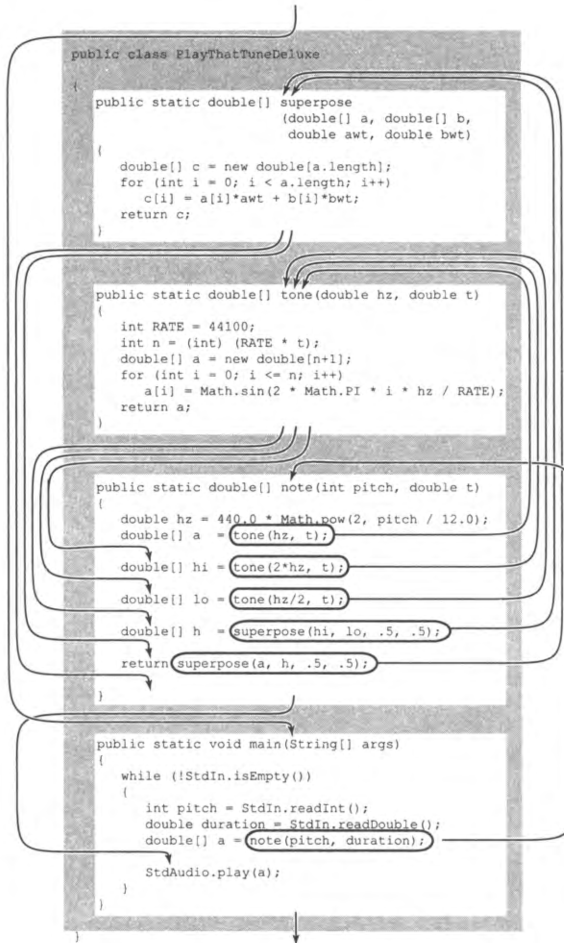
Программа в листинге 2.1.4 применяет эти концепции для создания более реалистичного звука по сравнению с программой из листинга 1.5.7. Для этого она выделяет четыре задачи, оформленные в виде функций.

- Создать чистый тон для заданной частоты и продолжительности.
- Выполнить наложение двух заданных звуковых волн с их относительными весами.
- Для заданной высоты звука и продолжительности создать ноту с гармониками.
- Прочитать и воспроизвести серию пар «высота/продолжительность» из стандартного ввода.

Все эти задачи хорошо подходят для реализации в виде функций, причем все эти функции в дальнейшем используют друг друга. Каждая функция определяется четко и однозначно. Все эти функции (как и `StdAudio`) представляют звук в виде последовательности чисел с плавающей точкой, хранящейся в массиве; числа соответствуют данным дискретизации звуковой волны на частоте 44 100 Гц.

До настоящего момента использование функций отчасти объяснялось удобством записи. Например, программная логика в листингах 2.1.1–2.1.3 достаточно проста — каждая функция вызывается всего в одном месте кода. С другой стороны, программа `PlayThatTuneDeluxe` (листинг 2.1.4) является убедительным примером эффективности использования функций для структурирования вычислений, потому что каждая функция вызывается по несколько раз. Например, функция `note()` вызывает функцию `tone()` трижды, а функцию `sum()` дважды. Без функций нам бы пришлось включить в программу несколько копий кода `tone()` и `sum()`. Функции, как и циклы, дают простой, но принципиально важный результат: одна последовательность команд (находящихся в определении метода) выполняется многократно в ходе выполнения программы — по одному разу для каждого вызова функции в программной логике `main()`.

Функции (статические методы) важны еще и тем, что они предоставляют средства для расширения языка Java в программах. После того как вы реализуете и отладите функции (такие, как `harmonic()`, `pdf()`, `cdf()`, `mean()`, `abs()`, `exchange()`, `shuffle()`, `isPrime()`, `uniform()`, `superpose()`, `note()` и `tone()`), вы сможете использовать их почти так же, как если бы они были встроены в Java. Этот гибкий механизм расширения открывает путь в совершенно новый мир программирования. До этого программы Java рассматривались как последовательности строк кода; теперь программу Java следует рассматривать как набор статических методов, вызывающих друг друга. Привычная передача управления от строки к строке присутствует и в статических методах, но для программы в целом используется более высокий уровень передачи управления, определяемый вызовами и возвратами из статических методов. Такой подход позволяет вам мыслить в контексте действий, необходимых приложению, вместо простых арифметических операций с примитивными типами, встроенными в Java.



Передача управления между несколькими статическими методами

Всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это. Примеры этого раздела (и программы в оставшейся части книги) наглядно демонстрируют, почему вам стоит придерживаться этого принципа. Со статическими методами вы можете:

- разделить длинную последовательность команд на независимые части;
- повторно использовать код без копирования;
- работать с концепциями более высокого уровня (например, со звуковыми волнами).

Такой код создает меньше проблем в понимании, сопровождении и отладке, чем длинная программа, состоящая исключительно из операций присваивания, условных переходов и циклов Java. В следующем разделе будет рассмотрена идея использования статических методов, определенных в других программах, которая откроет перед вами путь на новый уровень программирования.

Вопросы и ответы

В. Что произойдет, если опустить ключевое слово `static` при определении статического метода?

О. Как обычно, лучший способ найти ответ на такой вопрос — попробовать и посмотреть, что произойдет. Опустив модификатор `static` в методе `harmonic()` в программе `Harmonic`, вы получите следующий результат:

```
Harmonic.java:15: error: non-static method harmonic(int)
cannot be referenced from a static context
        double value = harmonic(arg);
                               ^
1 error
```

Нестатические методы отличаются от статических методов; вы узнаете о них в главе 3.

В. Что произойдет, если после команды `return` будет следовать код?

О. После достижения команды `return` управление немедленно возвращается в точку вызова, поэтому любой код после команды `return` бесполезен. Java распознает такую ситуацию как ошибку времени компиляции[1] и сообщает о недостижимом коде.

В. Что произойдет, если опустить строку `return`?

О. Ничего страшного, если метод имеет возвращаемый тип `void`. В этом случае управление возвращается в точку вызова. Если же возвращаемый тип отличен от `void` и если какой-либо путь в коде метода не завершается командой `return`, Java выдает сообщение об ошибке.

В. Зачем нужно использовать возвращаемый тип `void`? Почему бы просто не опустить возвращаемый тип?

О. Так требует Java; мы должны повиноваться. Сомнение в правильности решений, принятых создателями языка программирования, — первый шаг на пути к созданию новых языков.

---

[1] Строго говоря, это правильнее расматривать как предупреждение: наличие недостижимого кода не мешает программе выполняться. Но это же может быть признаком неправильно написанной программы, то есть наличия *логической* ошибки. — *Примеч. науч. ред.*

В. Можно ли вернуть управление из функции типа `void` с помощью `return`? Если да, то какое возвращаемое значение использовать?

О. Можно. Используйте операцию `return;` без возвращаемого значения.

В. От всей этой истории с побочными эффектами и передачей массивов в аргументах голова идет кругом. Это действительно так важно?

О. Да. Управление побочными эффектами — одна из самых важных задач программиста в больших системах. Безусловно, вы не пожалеете о времени, потраченном на выяснение различий между передачей по значению (для аргументов примитивного типа) и передачей по ссылке (для аргументов-массивов). Этот же механизм используется и для других типов данных, как вы узнаете в главе 3.

В. Тогда почему бы не устранить саму возможность побочных эффектов, передавая по ссылке все аргументы, включая массивы?

О. Представьте большой массив — допустим, с миллионами элементов. Есть ли смысл копировать все эти значения в статический метод, который поменяет местами всего два из них?

По этой причине многие языки программирования поддерживают передачу массива функциям без копирования его элементов. В качестве примера исключения можно назвать `Matlab`.

В. В каком порядке Java обрабатывает вызовы методов?

О. Независимо от приоритета операторов и ассоциативности Java обрабатывает подвыражения (включая вызовы методов) и списки аргументов слева направо. Например, при обработке выражения

```
f1() + f2() * f3(f4(), f5())
```

Java вызывает методы в порядке `f1()`, `f2()`, `f4()`, `f5()` и `f3()`. Это особенно важно для методов, создающих побочные эффекты. Каноны хорошего стиля программирования не рекомендуют писать код, зависящий от порядка обработки.

Упражнения

2.1.1. Напишите статический метод `max3()`, который получает три аргумента типа `int` и возвращает значение наибольшего из них. Добавьте перегруженную функцию, которая делает то же самое для трех значений `double`.

2.1.2. Напишите статический метод `odd()`, который получает три аргумента типа `boolean` и возвращает `true`, если истинно нечетное число аргументов, или `false` в противном случае.

2.1.3. Напишите статический метод `majority()`, который получает три аргумента типа `boolean` и возвращает `true`, если как минимум два значения аргументов ис-

тинны, или `false` в противном случае. Не используйте операцию условного перехода `if`.

2.1.4. Напишите статический метод `eq()`, который получает в аргументах два массива с элементами `int`. Метод должен возвращать `true`, если массивы имеют одинаковую длину, а все соответствующие пары элементов равны, или `false` в противном случае.

2.1.5. Напишите статический метод `areTriangular()`, который получает три аргумента типа `double` и возвращает `true`, если они могут быть сторонами треугольника (то есть ни одно из значений не больше суммы двух других). См. упражнение 1.2.15.

2.1.6. Напишите статический метод `sigmoid()`, который получает аргумент x типа `double` и возвращает значение типа `double`, вычисленное по формуле $1/(1+e^{-x})$.

2.1.7. Напишите статический метод `sqrt()`, который получает аргумент типа `double` и возвращает квадратный корень из этого числа. Используйте метод Ньютона (см. листинг 1.3.6) для вычисления результата.

2.1.8. Приведите трассировку вызова для команды `java Harmonic 3 5`.

2.1.9. Напишите статический метод `lg()`, который получает аргумент `n` типа `double` и возвращает логарифм `n` по основанию 2. В решении можно использовать библиотеку Java `Math`.

2.1.10. Напишите статический метод `lg()`, который получает аргумент `n` типа `int` и возвращает наибольшее целое число, не большее логарифма `n` по основанию 2. В решении *нельзя* использовать библиотеку Java `Math`.

2.1.11. Напишите статический метод `signum()`, который получает аргумент `n` типа `int` и возвращает `-1`, если `n` меньше `0`; `0`, если `n` равно 0; или `+1`, если `n` больше `0`.

2.1.12. Имеется следующий метод `duplicate()`:

```
public static String duplicate(String s)
{
   String t = s + s;
   return t;
}
```

Что делает следующий фрагмент кода?

```
String s = "Hello";
s = duplicate(s);
String t = "Bye";
t = duplicate(duplicate(duplicate(t)));
StdOut.println(s + t);
```

2.1.13. Имеется следующий статический метод `cube()`:

```
public static void cube(int i)
{
   i = i * i * i;
}
```

Сколько раз будет выполнен следующий цикл `for`?

```
for (int i = 0; i < 1000; i++)
   cube(i);
```

Ответ: ровно 1000 раз. Вызов `cube()` не влияет на вызывающий его код. Он изменяет значение локального параметра `i`, но это изменение никак не влияет на `i` в цикле `for` — это совершенно другая переменная. Если заменить вызов `cube(i)` командой `i = i * i * i;` (как вы, возможно, предполагали), то цикл будет выполнен 5 раз, а `i` в начале каждой из пяти итераций будет принимать значения `0`, `1`, `2`, `9` и `730` соответственно.

2.1.14. Следующая формула ***контрольной суммы*** широко применяется банками и компаниями, выпускающими кредитные карты, для проверки номеров счетов:

$$d_0 + f(d_1) + d_2 + f(d_3) + d_4 + f(d_5) + \ldots = 0 \pmod{10},$$

где d_i — цифры номера счета, а $f(d)$ — сумма цифр $2d$ (например, $f(7) = 5$, потому что $2 \times 7 = 14$, а $1 + 4 = 5$). Например, номер 17 327 — валидный, потому что $1 + 5 + 3 + 4 + 7 = 20$, а это число делится на 10. Реализуйте функцию f и напишите программу, которая получает целое число из 10 цифр в аргументе командной строки и выводит 11-значное число, являющееся валидным номером счета: сначала идут 10 цифр полученного числа, а за ними цифра контрольной суммы.

2.1.15. Для двух звезд с углами *склонения* и *прямого восхождения* (d_1, a_1) и (d_2, a_2) стягиваемый ими угол задается формулой

$$2 \arcsin((\sin 2(d/2) + \cos(d_1)\cos(d_2)\sin^2(a/2))^{1/2}),$$

где a_1 и a_2 — углы в диапазоне от −180 до 180°, d_1 и d_2 — углы в диапазоне от −90 до 90°, $a = a_2 - a_1$ и $d = d_2 - d_1$. Напишите программу, которая получает склонение и прямое восхождение двух звезд как аргументы командной строки и выводит стягиваемый ими угол. *Подсказка*: будьте внимательны с преобразованием градусов в радианы.

2.1.16. Напишите статический метод `scale()`, который получает аргумент — массив с элементами `double` — и в качестве побочного эффекта масштабирует элементы массива, чтобы каждый элемент находился в диапазоне от 0 до 1 (из каждого элемента вычитается минимальное из значений элементов, после чего каждый элемент делится на разность между максимумом и минимумом). Используйте метод `max()`, приведенный в таблице в тексте; напишите и используйте парный метод `min()`.

2.1.17. Напишите статический метод `reverse()`, который получает в качестве аргумента массив строк и возвращает новый массив, в котором строки переставлены в обратном порядке. (Не изменяйте порядок строк в массиве-аргументе.) Напишите также статический метод `reverseInplace()`, который получает в качестве аргумента массив строк и создает побочный эффект, переставляя в обратном порядке строки в массиве-аргументе.

2.1.18. Напишите статический метод `readBoolean2D()`, который читает двумерную матрицу `boolean` (с размерами) из стандартного ввода и возвращает полученный двумерный массив.

2.1.19. Напишите статический метод `histogram()`, который получает в качестве аргументов массив `a[]` с элементами `int` и целое число `m` и возвращает массив длины `m`, каждый `i`-й элемент которого содержит количество вхождений целого числа `i` в `a[]`. Если предположить, что все значения в `a[]` лежат в диапазоне от `0` до `m-1`, сумма значений в возвращенном массиве должна быть равна `a.length`.

2.1.20. На основе фрагментов кода, приведенных в этом разделе и в разделе 1.4, разработайте программу, которая получает целое число `n` как аргумент командной строки и выводит `n` наборов из пяти карт, разделенных пустыми строками. Карты сдаются из перетасованной колоды, по одной на строку. При выводе используются названия карт (например, `Ace of Clubs`).

2.1.21. Напишите статический метод `multiply()`, который получает в качестве аргументов две квадратные матрицы и вычисляет их произведение (обе матрицы имеют одинаковую размерность). Дополнительное задание: сделайте так, чтобы ваша программа работала в любых ситуациях, когда количество столбцов в первой матрице равно количеству строк во второй матрице.

2.1.22. Напишите статический метод `any()`, который получает в качестве аргумента массив `boolean` и возвращает `true`, если хотя бы один из элементов массива истинен, и `false` в противном случае. Напишите также статический метод `all()`, который получает в качестве аргумента массив значений `boolean` и возвращает `true`, если все элементы массива истинны, и `false` в противном случае.

2.1.23. Разработайте версию `getCoupon()`, которая лучше моделирует ситуацию, когда один из купонов встречается редко. Выберите одно из `n` значений случайным образом: это значение должно возвращаться с вероятностью `1/(1000n)`, а все остальные значения с равными вероятностями. Дополнительное задание: как это изменение влияет на ожидаемое количество купонов, которые необходимо собрать в задаче о собирателе купонов?

2.1.24. Измените программу `PlayThatTune`, чтобы она добавляла гармоники в двух октавах от каждой ноты, веса которых равны половине веса однооктавных гармоник.

Упражнения повышенной сложности

2.1.25. *Задача о дне рождения.* Разработайте класс со статическими методами для изучения задачи о дне рождения (см. упражнение 1.4.38).

2.1.26. *Функция Эйлера.* Функция Эйлера (тотиент) играет важную роль в теории чисел: $\Phi(n)$ определяется как количество положительных целых чисел, меньших либо равных n, взаимно-простых с n (не имеющих общих делителей с n, кроме 1). Напишите класс со статическим методом, который получает целочисленный ар-

гумент n и возвращает $\Phi(n)$, и метод `main()`, который получает целочисленный аргумент командной строки, вызывает метод с этим аргументом и выводит полученное значение.

2.1.27. *Гармонические числа.* Напишите программу `Harmonic`, которая содержит три статических метода `harmoinc()`, `harmoincSmall()` и `harmonicLarge()` для вычисления гармонических чисел. Метод `harmonicSmall()` должен только вычислять сумму (как в программе 1.3.5), метод `harmonicLarge()` должен использовать аппроксимацию $H_n = \log_e(n) + \gamma + 1/(2n) - 1/(12n^2) + 1/(120n^4)$ (число $\gamma = 0{,}577215664901532...$ называется *константой Эйлера*), а метод `harmonic()` должен вызывать `harmonicSmall()` для `n < 100` и `harmonicLarge()` в противном случае.

2.1.28. *Проверка формулы Блэка — Шоулза.* Формула Блэка — Шоулза определяет теоретическое значение цены на европейские опционы по акциям, по которым не выплачиваются дивиденды, при текущей цене акций s, цене исполнения x, безрисковой процентной ставке с непрерывным начислением r, волатильности v и времени до истечения срока опциона (в годах) t. Значение Блэка — Шоулза определяется формулой $s\,\Phi(a) - x\,e^{-rt}\,\Phi(b)$, где $\Phi(z)$ — *гауссова функция распределения*, $a = (\ln(s/x) + (r + \sigma^2/2)\,t)\,/\,(\sigma\sqrt{t}\,)$ и $b = a - \sigma\sqrt{t}$. Напишите программу, которая получает s, r, σ и t из командной строки и выводит значение Блэка — Шоулза.

2.1.29. *Пики Фурье.* Напишите программу, которая получает аргумент командной строки `n` и рисует график функции

$$(\cos(t) + \cos(2\,t) + \cos(3\,t) + ... + \cos(n\,t))\,/\,n$$

для 500 равномерно распределенных точек дискретизации t от -10 до 10 (в радианах). Запустите свою программу для $n = 5$ и $n = 500$. *Примечание*: вы заметите, что сумма сходится к пику (0 везде, кроме одного значения). Это свойство лежит в основе доказательства того, что любая плавная функция может быть выражена в виде суммы синусоид.

2.1.30. *Календарь.* Напишите программу `Calendar`, которая получает два целых числа `m` и `y` как аргументы командной строки и выводит календарь на месяц `m` года `y`, как в следующем примере:

```
% java Calendar 2 2009
February 2009
 S  M Tu  W Th  F  S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

Подсказка: используйте программу `LeapYear` (листинг 1.2.4) и упражнение 1.2.29.

2.1.31. *Метод Хорнера.* Напишите класс `Horner` с методом `evaluate()`, который получает в качестве аргументов число с плавающей точкой `x` и массив `p[]` и возвращает результат вычисления многочлена, коэффициентами которого являются элементы `p[]`:

$$p(x) = p_0 + p_1x^1 + p_2x^2 + \ldots + p_{n-2}\,x^{n-2} + p_{n-1}\,x^{n-1}.$$

Используйте *метод Хорнера* — эффективный способ проведения вычислений, в основе которого лежит следующая группировка:

$$p(x) = p_0 + x\,(p_1 + x\,(p_2 + \ldots + x\,(p_{n-2} + x\,p_{n-1}))\ldots).$$

Напишите тестовую программу со статическим методом `exp()`, который использует `evaluate()` для приближенного вычисления e^x, используя первые n слагаемых разложения в ряд Тейлора $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$. Эта программа должна получать аргумент командной строки `x` и сравнивать ваш результат с вычисленным при вызове `Math.exp(x)`.

2.1.32. *Аккорды.* Разработайте версию `PlayThatTune`, которая поддерживает работу с аккордами (и с включением гармоник). Разработайте формат ввода, который позволит задавать разную продолжительность для каждого аккорда и разные весовые коэффициенты для каждой ноты в аккорде. Создайте тестовые файлы для проверки вашей программы на разных аккордах и гармониках. Создайте версию мелодии «К Элизе», в которой используется ваша программа.

2.1.33. *Закон Бенфорда.* Американский астроном Саймон Ньюком заметил одну странность в сборнике логарифмических таблиц: начальные страницы были истрепаны гораздо сильнее, чем последние. Он предположил, что ученые выполняли больше вычислений с числами, начинавшимися на 1, чем с числами, начинавшимися на 8 и 9, и выдвинул предположение, что в общем случае начальной цифрой со значительно большей вероятностью окажется 1 (приблизительно 30%), чем цифра 9 (менее 4%). Это явление, называемое *законом Бенфорда*, часто используется в качестве статистического теста. Например, бухгалтеры-криминалисты ФРС используют его для выявления налогового мошенничества. Напишите программу, которая читает последовательность целых чисел из стандартного ввода и выводит сводку с указанием того, сколько раз каждая из цифр 1–9 является начальной цифрой введенного числа. Разбейте вычисления на статические методы. Используйте свою программу для проверки закона на таблицах с данными, найденных на вашем компьютере или в Интернете. Затем напишите программу, которая могла бы обмануть специалистов из ФРС: ваша программа должна генерировать случайные величины от \$1,00 до \$1000,00 с таким же распределением, которое вы наблюдали.

2.1.34. *Биномиальное распределение.* Напишите функцию

```
public static double binomial(int n, int k, double p)
```

для вычисления вероятности выпадения ровно k «орлов» при n бросков неправильной монеты («орел» выпадает с вероятностью p) по формуле:

$$f(n, k, p) = p^k(1 - p)^{n-k}\, n!/(k!(n - k)!)$$

Подсказка: чтобы избежать переполнения, вычислите $x = \ln f(n, k, p)$ и верните e^x. В коде `main()` получите n и p из командной строки и убедитесь в том, что сумма

по всем значениям k от 0 до n равна (приблизительно) 1. Также сравните каждое вычисленное значение с нормальным приближением

$$f(n, k, p) \approx \phi(np, np(1-p))$$

(см. упражнение 2.2.1).

2.1.35. *Собирание купонов с биномиальным распределением.* Разработайте версию `getCoupon()`, использующую функцию `binomial()` из предыдущего упражнения, для возвращения купонов в соответствии с биномиальным распределением с $p = 1/2$. *Подсказка*: сгенерируйте случайное число x с равномерным распределением от 0 до 1, после чего верните наименьшее значение k, для которого сумма $f(n, j, p)$ для всех $j < k$ превышает x. *Дополнительное задание*: разработайте гипотезу для описания поведения функции собирателя купонов под действием этого предположения.

2.1.36. *Почтовые штрихкоды.* Штрихкоды, используемые почтовой системой США для доставки почты, определяются следующим образом: каждая цифра ZIP-кода кодируется последовательностью из трех полос половинной высоты и двух полос полной высоты. Штрихкод начинается и завершается полосой полной высоты (ограничители) и включает контрольную цифру (после ZIP-кода из 5 или 9 цифр), вычисляемую как остаток от деления суммы исходных цифр на 10. Реализуйте следующие функции.

- Нарисовать полосу половинной или полной высоты с использованием `StdDraw`.
- Для заданной цифры вывести представляющую ее последовательность полос.
- Вычислить контрольную цифру.

Также реализуйте тестовое приложение, которое получает ZIP-код из пяти (или девяти) цифр как аргумент командной строки и выводит соответствующий ему почтовый штрихкод.

2.2. Библиотеки и клиенты[1]

Все программы, которые вы писали до настоящего момента, состояли из единственного файла .java с кодом на Java. В больших программах хранить весь код в одном

[1] Здесь клиентами называются программы и отдельные методы, использующие другие методы; более подробно эта концепция разбирается ниже. Таким образом, клиентами будут считаться вызывающий (головной) модуль, тестовое (демонстрационное) приложение и т. п. Это пока не имеет прямого отношения к архитектуре «клиент-сервер», типичной для сетевых приложений. — *Примеч. науч. ред.*

файле неэффективно и нецелесообразно. К счастью, в Java очень легко включить в один файл ссылку на метод, определенный в другом файле. Эта возможность имеет два важных последствия для нашего стиля программирования.

Во-первых, с ней становится возможным *повторное использование кода*. Программа может использовать уже написанный код, прошедший отладку, — и не копированием исходного текста, а простым включением в программу обращения к нему. Возможность определения кода, пригодного для повторного использования, стала важнейшим аспектом современного программирования. По сути, речь идет о расширении Java — вы можете определять и использовать собственные операции с данными.

Во-вторых, с ней становится возможным *модульное программирование*. Вы можете не только разделить программу на статические методы, как описано в разделе 2.1, но и хранить эти методы в разных файлах, группируя их в соответствии с потребностями приложения. Модульное программирование играет такую важную роль прежде всего потому, что оно позволяет *независимо* разрабатывать, компилировать и отлаживать отдельные части больших программ. Каждый завершенный фрагмент остается в собственном файле для последующего использования, и вам не нужно беспокоиться о технических подробностях его строения. Мы разрабатываем библиотеки статических методов для других программ; при этом каждая библиотека хранится в отдельном файле, а ее методы используются в других программах. Кстати, вы уже использовали их — примерами служат библиотека Java `Math` и наши библиотеки `Std*` для выполнения ввода/вывода. Еще важнее, что, как вы вскоре увидите, собственные библиотеки определяются без малейших затруднений. Возможность определения библиотек и их использования в разных программах исключительно важна для того, чтобы вы могли строить программы для решения сложных задач.

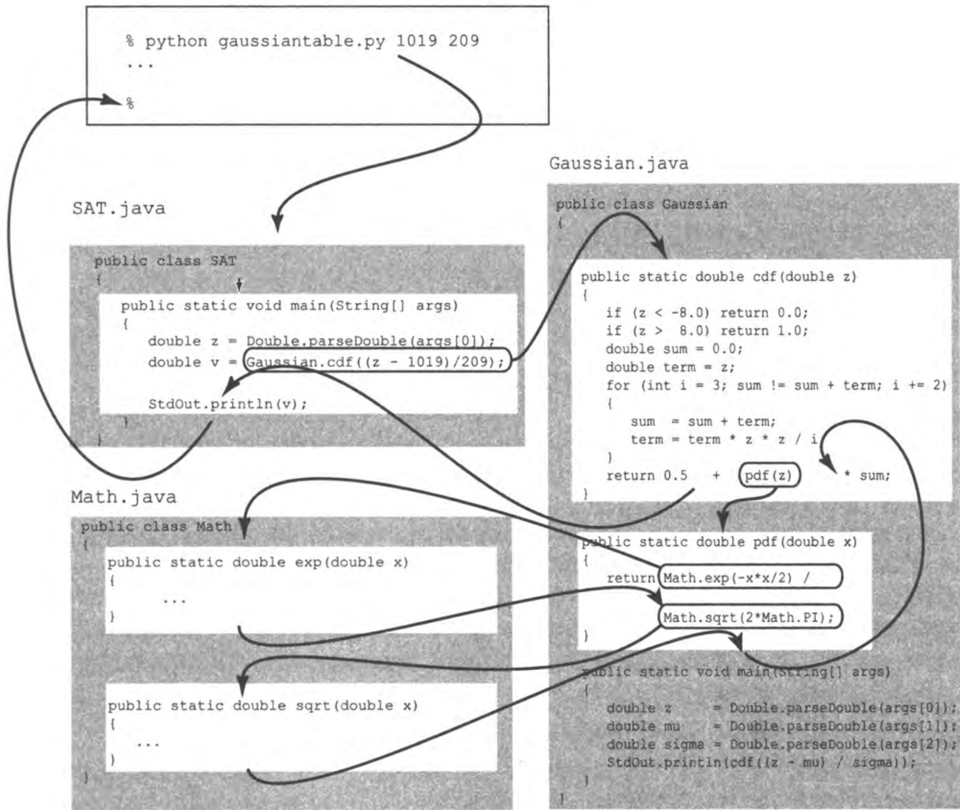
Итак, в разделе 2.1 вы перешли от представления о программе Java как о последовательности строк кода к представлению о ней как о классе, содержащем набор статических методов (одним из которых является `main()`). Теперь все готово к следующему шагу: после этого раздела вы сможете представлять программу Java как набор *классов*, каждый из которых представляет собой независимый модуль с набором методов. Так как каждый метод может вызывать метод другого класса, весь ваш код превращается в сеть методов, вызывающих друг друга и сгруппированных в классах. Располагая этой возможностью, можно переходить к управлению сложностью в программировании; для этого вы разбиваете задачи программирования на классы, которые можно реализовать и тестировать независимо.

Использование статических методов в других программах

Для обращения к статическому методу, определенному в другом классе, используется тот же механизм, который мы уже использовали для вызова таких методов, как `Math.sqrt()` и `StdOut.println()`.

- Позаботьтесь о том, чтобы оба класса были доступны для Java (например, поместив их в один каталог на вашем компьютере).

- Чтобы вызвать метод, поставьте перед его именем имя класса и разделитель-точку.



Передача управления в модульной программе

Например, можно написать простое тестовое (*«клиентское»*) приложение SAT.java, которое получает результаты вступительного экзамена *z* из командной строки и выводит процент студентов, набравших менее *z* баллов в заданном году (в котором среднее количество баллов было равно 1019, а среднеквадратическое отклонение составило 209). Для этого программа SAT.java должна вычислить $\Phi((z-1{,}019)/209)$, а метод `cdf()` из Gaussian.java (программа 2.1.2) прекрасно справляется с этой задачей. Все, что для этого нужно, — поместить файл Gaussian.java в один каталог с SAT.java и указать имя класса перед вызовом `cdf()`. Более того, любой другой класс в этом каталоге может использовать статические методы, определяемые в `Gaussian`, включая вызовы `Gaussian.pdf()` или `Gaussian.cdf()`. Библиотека `Math` всегда доступна в Java, поэтому любой класс может вызывать `Math.sqrt()` и `Math.exp()` без предварительной подготовки. Файлы Gaussian.java, SAT.java и Math.java реализуют классы Java, взаимодействующие друг с другом: `SAT` вызывает метод

из файла `Gaussian`, который вызывает другой метод из `Gaussian`, который, в свою очередь, вызывает два метода из `Math`.

Использование в проекте нескольких файлов, каждый из которых содержит независимый класс с разными методами, означает еще одно фундаментальное изменение в стиле программирования. Обычно этот подход к разработке называется *модульным программированием*: вы независимо разрабатываете и отлаживаете методы приложения, а затем используете их в любой момент в будущем. В этом разделе будут приведены наглядные примеры, которые помогут вам привыкнуть к такому подходу. Тем не менее, прежде чем переходить к примерам, стоит разобраться с некоторыми подробностями и терминами.

Ключевое слово public

Начиная с самой первой программы `HelloWorld` мы помечали все статические методы ключевым словом `public`. Этот модификатор указывает, что метод может использоваться любой другой программой, имеющей доступ к файлу. Методы также могут помечаться ключевым словом `private` (кроме того, существует еще несколько категорий), но пока вам это просто не нужно. Различные варианты рассматриваются в разделе 3.3.

Каждый модуль является классом

Модулем называется весь код, хранящийся в отдельном классе. В языке Java каждый модуль представляет собой класс Java, хранящийся в файле с именем, совпадающим с именем класса и имеющим расширение .java. В этой главе классы представляют собой обычные наборы статических методов (один из которых — метод `main()`). Общая структура класса Java гораздо сложнее, более подробно она описана в главе 3.

Файл .class

Когда вы компилируете программу (командой `javac`, за которой следует имя класса), компилятор Java создает файл с именем класса и расширением .class. Этот файл содержит код программы на *промежуточном* языке, более подходящем для выполнения компьютером. Если у вас имеется файл .class, вы сможете использовать методы этого модуля в своих программах, даже если вы не располагаете исходным кодом из соответствующего файла .java (но если в коде вдруг обнаружится ошибка — тут уж ничего не поделаешь!).

Компиляция при необходимости

На стадии компиляции программы Java обычно компилирует все, что необходимо откомпилировать для запуска этой программы. Если в программе `SAT` вызывается метод `Gaussian.cdf()`, то при выполнении команды `javac SAT.java` команда также проверит, изменялся ли файл Gaussian.java с момента последней компиляции (для этого его время последнего изменения сравнивается с временем создания

файла Gaussian.class). И если файл изменился, то компилятор также компилирует Gaussian.java! Если задуматься, становится ясно, что такой подход весьма разумен — ведь если вы найдете ошибку в файле Gaussian.java (и исправите ее), все классы, вызывающие методы из `Gaussian`, должны использовать новые версии.

Множественные методы main()

Также стоит обратить внимание еще на один нюанс: метод `main()` может присутствовать более чем в одном классе. В нашем примере каждый из классов `SAT` и `Gaussian` содержит собственный метод `main()`. Если вы вспомните правило выполнения программ, вы увидите, что путаница исключена: когда вы вводите команду `java` с именем класса, то Java передает управление машинному коду, который соответствует методу `main()`, определенному именно в этом классе. Как правило, метод `main()` включается в каждый класс для тестирования и отладки его методов. Если вы хотите выполнить класс `SAT`, вводится команда `java SAT`; если вы хотите отладить программу `Gaussian`, вводится команда `java Gaussian` (с соответствующими аргументами командной строки).

Если рассматривать каждую программу, которую вы пишете, как код, который можно будет использовать в будущих программах, вы вскоре обзаведетесь множеством полезных инструментов. С точки зрения модульного программирования любое решение вычислительной задачи повышает потенциал вашей вычислительной среды.

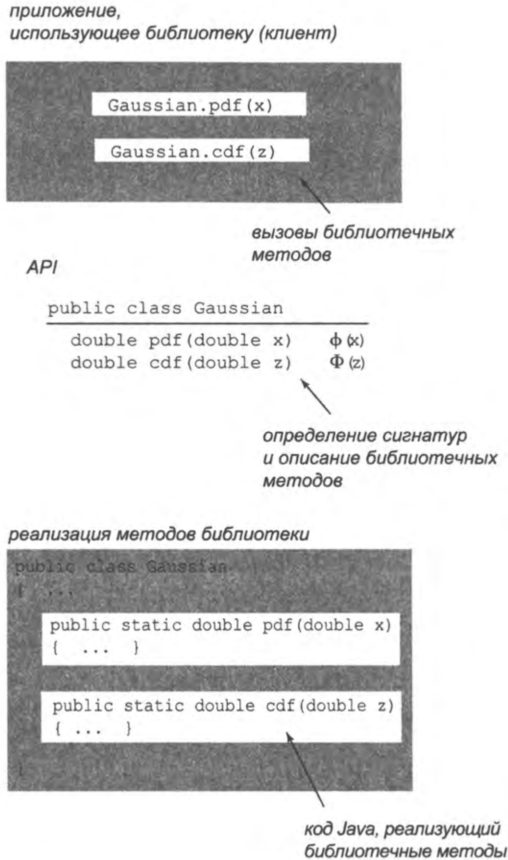
Представьте, например, что в будущем приложении вам понадобилось вычислить Ф. Почему бы просто не скопировать код реализации `cdf()` из `Gaussian`? Такое решение будет работать, но у вас появятся две копии кода, а это усложнит сопровождение. Если позднее вы захотите исправить или доработать код, вам придется вносить изменения в обеих копиях. Сравните с другим вариантом: просто вызвать `Gaussian.cdf()`. Наши методы — реализации и примеры их использования — будут быстро множиться, поэтому вы должны стремиться к тому, чтобы каждый метод существовал только в одном экземпляре.

С этого момента при написании каждой программы всегда старайтесь понять, как разбить ваши вычисления на несколько частей удобного размера и реализовать каждую часть так, словно кто-то собирается использовать ее в будущем. Чаще всего этим «кто-то» будете вы сами, и себя же вы будете благодарить за время и усилия, сэкономленные на переписывании и повторной отладке кода.

Библиотеки

Модули, методы которых предназначены прежде всего для использования из других программ, называют *библиотеками*. Важное обстоятельство при программировании на Java — существование буквально тысяч готовых библиотек, которыми вы можете пользоваться в своих программах. В книге будет приводиться информация о тех библиотеках, которые могут представлять для вас интерес, но подробное обсуждение откладывается на будущее, потому что многие библиотеки рассчитаны на

более опытных программистов. Вместо этого нас пока будет интересовать еще более важная идея: создание *пользовательских библиотек*. Эти библиотеки представляют собой наборы взаимосвязанных методов, предназначенных для использования из других программ. Ни одна библиотека Java не вместит все методы, которые могут когда-либо понадобиться при всех возможных вычислениях, поэтому возможность создания собственной библиотеки методов чрезвычайно важна для построения сложных приложений.



Библиотека — основные элементы

Клиенты

Далее мы будем называть *«клиентами»* методы или целые программы, вызывающие библиотечные методы. Если класс содержит метод, который является клиентом метода из другого класса, то первый класс называется клиентом второго класса. В нашем примере `SAT` является клиентом `Gaussian`. Класс может иметь несколько

клиентов. Например, все написанные вами программы, в которых вызывались методы `Math.sqrt()` или `Math.random()`, были клиентами `Math`.

API

Программисты обычно мыслят понятиями *контракта* между клиентом и реализацией — четкой спецификации того, что должен делать метод. Когда вы пишете и приложение-клиент, и реализацию, вы фактически заключаете контракт с самим собой, что полезно из соображений упрощения отладки. Но что еще важнее, этот подход обеспечивает возможность повторного использования кода. Вы можете писать программы, которые являются клиентами `Std*`, `Math` и других встроенных классов Java, из-за неформального контракта (описания того, что должна делать библиотека, на естественном языке) в сочетании с точными определениями сигнатур методов, доступных для использования. Вся эта информация в совокупности называется *программным интерфейсом*, или *API* (Application Programming Interface). Этот же механизм эффективно работает и в отношении пользовательских библиотек. API позволяет любому клиенту использовать библиотеку без изучения кода реализации — так же, как вы делали с методами `Math` и `Std*`. Главный принцип проектирования API — *предоставить клиенту те методы, которые ему нужны, и не предоставлять лишних*. API с чрезмерным количеством методов усложняет реализацию; API, в котором не хватает важных методов, может оказаться неудобным или бесполезным для клиентов.

Реализации

Реализацией называют код Java, реализующий методы API. По правилам этот код хранится в файле с именем библиотеки и расширением .java. Любая программа Java представляет собой реализацию некоторого API, и любой API бесполезен без реализации. Разрабатывая реализацию, мы стремимся к соблюдению условий контракта. Часто существует много вариантов реализации, и отделение клиентского кода от кода реализации дает нам возможность переключаться на обновленные, усовершенствованные реализации.

Рассмотрим для примера функции гауссова распределения. Эти функции не входят в библиотеку Java `Math`, но они играют важную роль в приложениях, поэтому есть смысл поместить их в библиотеку, где к ним можно будет обращаться из будущих программ-клиентов, и сформулировать соответствующий API:

public class Gaussian	
`double pdf(double x)`	$\phi(x)$
`double pdf(double x, double mu, double sigma)`	$\phi(x, \mu, \sigma)$
`double cdf(double z)`	$\Phi(z)$
`double cdf(double z, double mu, double sigma)`	$\Phi(z, \mu, \sigma)$

API нашей библиотеки статических методов для гауссова распределения

API включает не только функции гауссова распределения с одним аргументом, рассматривавшиеся ранее (см. листинг 2.1.2), но и версии с тремя аргументами (в которых клиент задает математическое ожидание и среднеквадратическое отклонение распределения), встречающиеся во многих статистических вычислениях. Реализация функций с тремя аргументами выглядит достаточно тривиально (см. листинг 2.2.1).

Сколько информации должен содержать API? Этот неоднозначный вопрос является предметом бурных дискуссий среди программистов и преподавателей компьютерных дисциплин. Вы можете попытаться включить в API как можно больше информации, но (как и для любого контракта!) объем информации, который будет эффективно работать, ограничен. В этой книге мы придерживаемся принципа, который напоминает наш основной принцип проектирования: *предоставляйте программистам клиентских приложений только ту информацию, которая им необходима, и не более*. Такой подход предоставляет существенно больше гибкости, чем альтернативное решение с передачей подробной информации о реализациях. В самом деле, любая дополнительная информация равносильна неявному расширению контракта, что нежелательно. У многих программистов есть дурная привычка разбираться в работе чужих методов по коду их реализации. Это может привести к появлению клиентского кода, который зависит от особенностей поведения, не указанных в API (не *документированных*), и поэтому не будет работать с новой реализацией. Реализации изменяются чаще, чем это может показаться. Например, каждый новый выпуск Java содержит много новых реализаций библиотечных функций.

Часто сначала появляется именно реализация. Допустим, у вас имеется рабочий модуль, который, как выясняется позднее, пригодится и в некоторой другой задаче, и вы просто начинаете использовать его методы в других программах. В такой ситуации стоит в какой-то момент более тщательно сформулировать API. Может оказаться, что методы проектировались без учета возможности повторного использования, и тогда стоит сформулировать API для такого использования (как было сделано для `Gaussian`).

В оставшейся части этого раздела рассматриваются примеры библиотек и клиентов. Это преследует две основные цели. Во-первых, они расширяют вашу среду программирования, делая ее пригодной для разработки все более сложных клиентских программ. Во-вторых, они служат примерами, которыми вы можете руководствоваться при разработке библиотек для собственных целей.

Случайные числа

Мы уже написали несколько программ, использующих `Math.random()`, но в коде часто встречались специфические идиомы для преобразования случайных значений типа `double` в диапазоне от 0 до 1, возвращаемых `Math.random()`, к тому типу случайных чисел, которые нам нужны (например, случайные значения типа `boolean`

Листинг 2.2.1. Библиотека для работы со случайными числами

```
public class StdRandom
{
   public static int uniform(int n)
   {  return (int) (Math.random() * n);  }

   public static double uniform(double lo, double hi)
   {  return lo + Math.random() * (hi - lo);  }

   public static boolean bernoulli(double p)
   {  return Math.random() < p;  }

   public static double gaussian()
   {  /* См. упражнение 2.2.17. */  }

   public static double gaussian(double mu, double sigma)
   {  return mu + sigma * gaussian();  }

   public static int discrete(double[] probabilities)
   {  /* См. упражнение 1.6.2. */  }

   public static void shuffle(double[] a)
   {  /* См. упражнение 2.2.4. */  }

   public static void main(String[] args)
   {  /* См. текст. */  }
}
```

Методы этой библиотеки генерируют случайные числа разных типов: случайные неотрицательные целые числа, меньшие заданного; числа, равномерно распределенные в заданном диапазоне; случайные биты (Бернулли); случайные числа со стандартным распределением Гаусса; случайные числа с распределением Гаусса для заданного математического ожидания и среднеквадратичного отклонения; случайные целые числа, распределенные в соответствии с заданным дискретным распределением.

```
% java StdRandom 5
90 26.36076 false 8.79269 0
13 18.02210 false 9.03992 1
58 56.41176 true  8.80501 0
29 16.68454 false 8.90827 0
85 86.24712 true  8.95228 0
```

или значения `int` в заданном диапазоне). Демонстрируя эффективное повторное использование кода реализации этих идиом, мы с этого момента будем использовать библиотеку `StdRandom` из листинга 2.2.1. `StdRandom` использует механизм перегрузки для генерирования случайных чисел с различным распределением. Используйте эту библиотеку так же, как вы используете стандартные библиотеки ввода/вывода (см. первый вопрос в разделе «Вопросы и ответы» в конце раздела 2.1). Как обычно, мы приводим сводку методов API для библиотеки `StdRandom`:

public class StdRandom	
void setSeed(long seed)	инициализирует генератор для получения воспроизводимых результатов
int uniform(int n)	целое число в диапазоне от 0 до n–1
double uniform(double lo, double hi)	число с плавающей точкой в диапазоне от lo до hi
boolean bernoulli(double p)	true с вероятностью p, false в противном случае
double gaussian()	распределение Гаусса, математическое ожидание 0, среднеквадратическое отклонение 1
double gaussian(double mu, double sigma)	распределение Гаусса, математическое ожидание mu, среднеквадратическое отклонение sigma
int discrete(double[] p)	i с вероятностью p[i]
void shuffle(double[] a)	случайным образом переставляет элементы массива a[]

API нашей библиотеки статических методов для работы со случайными числами

Эти методы достаточно знакомы пользователям, чтобы краткого описания API было достаточно. Собирая все методы, использующие `Math.random()` для генерирования случайных чисел разных типов в одном файле (StdRandom.java), мы можем сосредоточить все методы, генерирующие случайные числа, в одном файле (и повторно использовать код этого файла) вместо того, чтобы дублировать их код во всех программах, использующих эти методы. Более того, каждая программа, использующая один из таких методов, более «прозрачна» для понимания, чем вызывающая непосредственно `Math.random()`, потому что ее намерение использовать `Math.random()` однозначно выражено выбором метода из `StdRandom`.

Проектирование API

Мы делаем определенные предположения относительно значений, передаваемых каждому из методов `StdRandom`. Например, предполагается, что клиенты вызывают `uniform(n)` только для положительных целых `n`, `bernoulli(p)` — только для значений p в диапазоне от 0 до 1 и `discrete()` — только для массива с элементами от 0 до 1 и суммой 1. Все эти предположения являются частью контракта между клиентом и реализацией.

Библиотеки следует проектировать так, чтобы контракт был ясным и однозначным, а пользователь не увязал в технических подробностях. Как и во многих задачах из области программирования, хорошее проектирование API часто становится результатом многих итераций проб и ошибок и попыток работы с разными вариантами. Проектирование API должно проводиться очень тщательно, потому что при изменении API может возникнуть необходимость в изменении всех клиентов и всех реализаций. Цель проектирования — сформулировать, чего может ожидать клиент именно от API, *независимо от кода*. Такой подход дает программисту свободу в изменении кода и, возможно, в выборе реализации, которая достигает желаемого эффекта с большей эффективностью или большей точностью.

Тестирование модуля

И хотя мы реализовали `StdRandom`, не имея в виду никакое конкретное законченное приложение, у программистов существует хорошее правило: включать в код модуля тестовый головной метод `main()`, который не используется при обращении к библиотеке из клиентского класса, но упрощает отладку и тестирование ее методов. Каждый раз, когда вы создаете библиотеку, рассмотрите возможность включения метода `main()` для тестирования и отладки модуля. Правильная организация тестирования модуля может стать серьезной задачей программирования сама по себе (например, вопрос о том, как лучше протестировать, обладают ли генерируемые `StrRandom` числа такими же характеристиками, как и действительно случайные числа, до сих пор обсуждается экспертами). Как минимум всегда включайте метод `main()`, который:

- задействует весь код;
- предоставляет некоторые гарантии того, что код работает;
- получает аргумент из командной строки для расширенного тестирования.

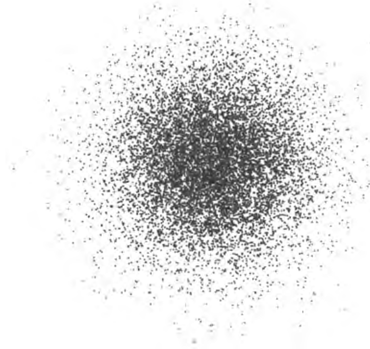
Далее этот метод `main()` должен выполнять более обстоятельное тестирование по мере расширения библиотеки. Например, в исходном варианте код `StdRandom` может выглядеть так (тестирование `shuffle()` остается для самостоятельного упражнения):

```
public static void main(String[] args)
{
   int n = Integer.parseInt(args[0]);
   double[] probabilities = { 0.5, 0.3, 0.1, 0.1 };
   for (int i = 0; i < n; i++)
   {
      StdOut.printf(„ %2d „ , uniform(100));
      StdOut.printf(„%8.5f „, uniform(10.0, 99.0));
      StdOut.printf(„%5b „  , bernoulli(0.5));
      StdOut.printf(„%7.5f „, gaussian(9.0, 0.2));
      StdOut.printf(„%2d „  , discrete(probabilities));
      StdOut.println();
   }
}
```

Если включить этот код в файл StdRandom.java и вызвать метод `main()` так, как показано в листинге 2.2.1, вывод не приносит никаких сюрпризов: целые числа в первом столбце с равной вероятностью принимают любые значения от 0 до 99; числа во втором столбце равномерно распределяются в диапазоне от 10,0 до 99,0; третий столбец содержит около половины истинных значений; числа в четвертом столбце в среднем равны 9,0 и не удаляются от 9,0 на слишком большое расстояние; и последний столбец содержит приблизительно 50% нулей, 30% единиц, 10% двоек и 10% троек. Если в одном из столбцов что-то пошло не так, можно ввести команду `java StdRandom 10` или `100`, чтобы получить больше результатов. В этом конкретном случае мы можем (и должны) провести более тщательное тестирование в специ-

альном тестовом приложении и убедиться в том, что числа обладают большинством свойств действительно случайных чисел из перечисленных распределений (см. упражнение 2.2.3). Один из эффективных способов тестирования основан на написании тестовых клиентов, использующих `StdDraw`, так как визуализация данных может наглядно показать, что программа работает так, как положено. Например, на диаграмме со множеством точек, координаты x и y которых взяты из разных распределений, часто образуется узор, наглядно демонстрирующий важные свойства распределения. И что еще важнее, ошибка в коде генератора случайных чисел с большой вероятностью немедленно проявится на такой диаграмме.

```
public class RandomPoints
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      for (int i = 0; i < n; i++)
      {
         double x = StdRandom.gaussian(.5, .2);
         double y = StdRandom.gaussian(.5, .2);
         StdDraw.point(x, y);
      }
   }
}
```



Тестовое приложение-клиент для `StdRandom`

Стрессовое тестирование

Широко используемые библиотеки, такие как `StdRandom`, должны также подвергаться *стрессовому тестированию*, в ходе которого мы убеждаемся в том, что если приложение-клиент выполнит что-то, противоречащее контракту или не предусмотренное им, то это не приведет к сбою в библиотеке. Библиотеки Java уже подвергались такому стрессовому тестированию; разработчикам пришлось тщательно анализировать каждую строку кода и задаваться вопросом, может ли какое-либо состояние стать причиной проблемы. Что должен делать метод `discrete()`, если сумма массивов элемента отлична от 1? А если в аргументе передается массив

нулевой длины? Что должна делать версия `uniform()` с двумя аргументами, если один или оба аргумента содержат `NaN`? `Infinity`? Чем больше вопросов такого рода вы придумаете, тем лучше. Такие случаи иногда называются *граничными случаями*. С опытом многие программисты учатся распознавать их на ранней стадии, чтобы избежать неприятных хлопот с отладчиком позднее. Как и прежде, лучше всего реализовать стресс-тест в виде отдельного клиента.

Ввод и вывод массивов

Вы уже видели (и еще неоднократно увидите) множество примеров, когда для хранения и обработки данных используются массивы. А это означает, что будет полезно создать библиотеку, дополняющую `StdIn` и `StdOut` статическими методами для чтения массивов примитивных типов из стандартного потока ввода и записи их в стандартный поток вывода. Следующий API предоставляет такие методы:

public class StdArrayIO	
`double[] readDouble1D()`	читает одномерный массив с элементами типа double
`double[][] readDouble2D()`	читает двумерный массив с элементами типа double
`void print(double[] a)`	выводит одномерный массив с элементами типа double
`void print(double[][] a)`	выводит двумерный массив с элементами типа double

Примечание 1. Формат для одномерных версий: целое значение n, за которым следуют n значений.
Примечание 2. Формат для двумерных версий: два целых значения m и n, за которыми следуют m × n значений по строкам.
Примечание 3. Также включены методы для int и boolean.

API для нашей библиотеки статических методов ввода и вывода массивов

Первые два замечания под таблицей отражают необходимость выбора некоторого формата файла. Для простоты и гармонии мы решили, что все значения в стандартном вводе должны включать размеры массива и следовать в указанном порядке. Методы `read*()` ожидают получать данные в этом формате, и методы `print()` выводят результаты в этом формате. Третье примечание в нижней части таблицы указывает, что `StdArrayIO` в действительности содержит 12 методов — по четыре для каждого из типов `int`, `double` и `boolean`. Методы `print()` перегружены (они имеют одно имя `print()`, но разные типы аргументов), но имена методов `read*()` должны быть разными: они состоят из имени типа (записанного с прописной буквы, как в `StdIn`) и суффикса `1D` или `2D`.

Реализация этих методов напрямую следует из кода, работающего с массивами, который рассматривался в разделах 1.4 и 2.1, как показано в программе `StdArrayIO` (листинг 2.2.2). Объединение всех этих статических методов в один файл — StdArrayIO.java — позволяет легко использовать этот код повторно и избавляет от необходимости отвлекаться на технические подробности чтения и вывода массивов, когда вы будете позднее писать клиентские программы.

Листинг 2.2.2. Библиотека ввода/вывода массивов

```
public class StdArrayIO
{
   public static double[] readDouble1D()
   {  /* См. упражнение 2.2.11.  */  }

   public static double[][] readDouble2D()
   {
      int m = StdIn.readInt();
      int n = StdIn.readInt();
      double[][] a = new double[m][n];
      for (int i = 0; i < m; i++)
         for (int j = 0; j < n; j++)
            a[i][j] = StdIn.readDouble();
      return a;
   }

   public static void print(double[] a)
   {  /* См. упражнение 2.2.11.  */  }

   public static void print(double[][] a)
   {
      int m = a.length;
      int n = a[0].length;
      System.out.println(m + " " + n);
      for (int i = 0; i < m; i++)
      {
         for (int j = 0; j < n; j++)
            StdOut.prinf("%9.5f ", a[i][j]);
         StdOut.println();
      }
      StdOut.println();
   }

   // Методы для других типов похожи (см. сайт книги).
   public static void main(String[] args)
   {  print(readDouble2D());  }
}
```

Библиотека статических методов упрощает чтение одномерных и двумерных массивов из стандартного потока ввода и вывод их в стандартный поток вывода. В формат файлов включены данные о размерах массивов (см. текст). В примере выводимые значения усечены.

```
% more tiny2D.txt
4 3
  .000  .270  .000
  .246  .224 -.036
  .222  .176  .0893
 -.032  .739  .270
```

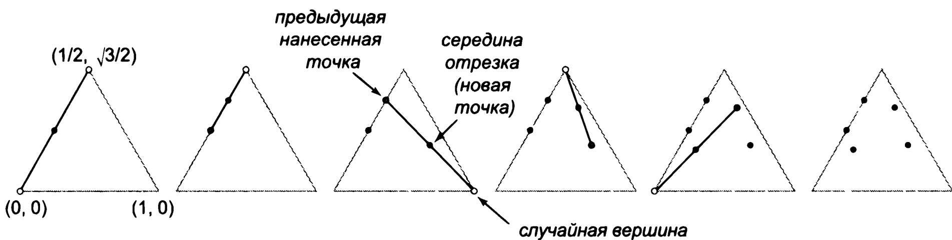
```
% java StdArrayIO < tiny.txt
4 3
  0.00000   0.27000   0.00000
  0.24600   0.22400  -0.03600
  0.22200   0.17600   0.08930
 -0.03200   0.73900   0.27000
```

Итерационные вычисления с функциями

Ученые обнаружили, что даже простые математические вычисления могут порождать сложные и интересные изображения. Вооружившись библиотеками StdRandom, StdDraw и StdArrayIO, мы попробуем изучить некоторые из них.

Треугольник Серпинского

Начнем с простого процесса: нанесите точку на одну из сторон равностороннего треугольника. Затем выберите одну из трех вершин случайным образом и нанесите новую точку на середине отрезка между только что нанесенной точкой и этой вершиной. Продолжайте повторять ту же операцию. Каждый раз выбирается случайная вершина треугольника для построения отрезка, середина которого станет следующей точкой фигуры. Так как мы делаем выбор случайно, множество точек должно обладать некоторыми характеристиками случайных точек, и это обстоятельство проявляется после нескольких начальных итераций:



Случайный процесс

Чтобы изучить результаты этого процесса для большого количества итераций, мы напишем программу для нанесения точек на график по описанным правилам:

```
double[] cx = { 0.000, 1.000, 0.500 };
double[] cy = { 0.000, 0.000, 0.866 };
double x = 0.0, y = 0.0;
for (int t = 0; t < trials; t++)
{
   int r = StdRandom.uniform(3);
   x = (x + cx[r]) / 2.0;
   y = (y + cy[r]) / 2.0;
   StdDraw.point(x, y);
}
```

Координаты x и y вершин треугольника хранятся в массивах cx[] и cy[] соответственно. Метод StdRandom.uniform() используется для выбора случайного индекса r в этих массивах — выбранная вершина имеет координаты (cx[r], cy[r]). Координата x средней точки отрезка из (x, y) в эту вершину задается выражением (x + cx[r])/2.0; аналогичное вычисление дает координату y. Добавление вызова StdDraw.point() и включение этого кода в цикл завершают реализацию. Интересно,

что, несмотря на случайность, после большого количества итераций на рисунке всегда проявляется одна и та же фигура! Эта фигура называется *треугольником Серпинского* (см. упражнение 2.3.27). Вопрос о том, почему в ходе случайного процесса образуется такая правильная фигура, чрезвычайно интересен.

Случайный процесс?

Папоротник Барнсли

Еще интереснее, что изменение правил игры позволяет получить невероятно разнообразные изображения. Один из выдающихся примеров такого рода известен как *папоротник Барнсли*. Изображение генерируется таким же процессом, но на этот раз им управляют формулы из следующей таблицы. На каждом шаге формулы, используемые для обновления x и y, выбираются с заданной вероятностью (в 1% случаев используется первая пара формул, в 85% случаев — вторая пара формул и т. д.).

Вероятность	Обновление x	Обновление y
1%	x = 0,500	y = 0,16y
85%	x = 0,85x + 0,04y + 0,075	y = –0,04x + 0,85y + 0,180
7%	x = 0,20x + 0,26y + 0,400	y = 0,23x + 0,22y + 0,045
7%	x = –0,15x + 0,28y + 0,575	y = 0,26x + 0,24y + 0,086

Для итерационного применения этих правил можно было бы написать точно такой же код, какой был написан для треугольника Серпинского, но вычисления с матрицами предоставляют универсальный способ обобщения этого кода для произвольного набора правил. Имеются `m` разных преобразований, выбранных из вектора $1 \times m$ методом `StdRandom.discrete()`. Для каждого преобразования имеется формула обновления x и формула обновления y; таким образом, для коэффициентов формул используются две матрицы $m \times 3$: одна для x, другая для y. Программа `IFS` (листинг 2.2.3) реализует эту версию вычислений, управляемую данными. Эта программа открывает безграничные возможности для исследований: она выполняет итерации для любых введенных данных, содержащих вектор,

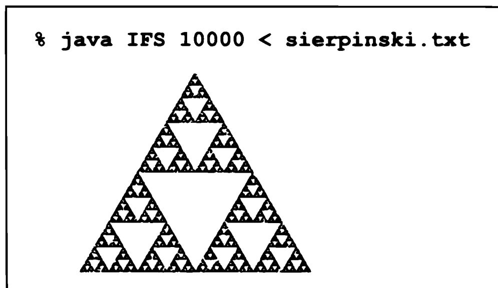
Листинг 2.2.3. Итерационные вычисления

```
public class IFS
{
   public static void main(String[] args)
   {  // Вывод результатов итерационных вычислений
      // на основе данных из StdIn.
      int trials = Integer.parseInt(args[0]);
      double[] dist   = StdArrayIO.readDouble1D();
      double[][] cx   = StdArrayIO.readDouble2D();
      double[][] cy   = StdArrayIO.readDouble2D();
      double x = 0.0, y = 0.0;
      for (int t = 0; t < trials; t++)
      {  // Нанесение на диаграмму результата 1 итерации.
         int r = StdRandom.discrete(dist);
         double x0 = cx[r][0]*x + cx[r][1]*y + cx[r][2];
         double y0 = cy[r][0]*x + cy[r][1]*y + cy[r][2];
         x = x0;
         y = y0;
         StdDraw.point(x, y);
      }
   }
}
```

`trials`	итерации
`dist[]`	вероятности
`cx[][]`	коэффициенты x
`cy[][]`	коэффициенты y
`x, y`	текущая точка

Приложение-клиент, управляемое данными, использует StdArrayIO, StdRandom и StdDraw и строит изображение фрактала, определяемого вектором $1 \times m$ (вероятности) и двумя матрицами $m \times 3$ (коэффициенты для обновления x и y соответственно), считываемыми из стандартного потока ввода. Результат отображается в виде множества точек средствами стандартной графики. Примечательно, что этой программе не обязательно знать значение m, потому что для создания и обработки матриц используется отдельный специальный метод.

```
% more sierpinski.txt
3
  .33 .33 .34
3 3
  .50 .00 .00
  .50 .00 .50
  .50 .00 .25
3 3
  .00 .50 .00
  .00 .50 .00
  .00 .50 .433
```



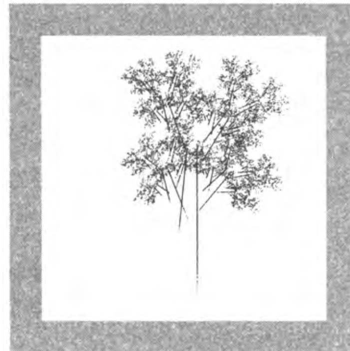
определяющий вероятности, и две матрицы, определяющие коэффициенты для x и для y. И хотя на каждом шаге выбирается случайная формула, для заданных коэффициентов будет строиться одна и та же фигура: изображение, очень похожее на папоротник, который растет в лесу, а не на результат обычного случайного процесса.

```
% more barnsley.txt
4
  .01 .85 .07 .07
4 3
  .00  .00  .500
  .85  .04  .075
  .20 -.26  .400
 -.15  .28  .575
4 3
  .00  .16  .000
 -.04  .85  .180
  .23  .22  .045
  .26  .24 -.086
```

```
% java IFS 20000 < barnsley.txt
```

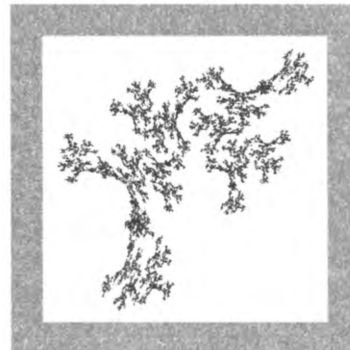
```
% more tree.txt
6
  .1 .1 .2 .2 .2 .2
6 3
  .00  .00  .550
 -.05  .00  .525
  .46 -.15  .270
  .47 -.15  .265
  .43  .26  .290
  .42  .26  .290
6 3
  .00  .60  .000
 -.50  .00  .750
  .39  .38  .105
  .17  .42  .465
 -.25  .45  .625
 -.35  .31  .525
```

```
% java IFS 20000 < tree.txt
```



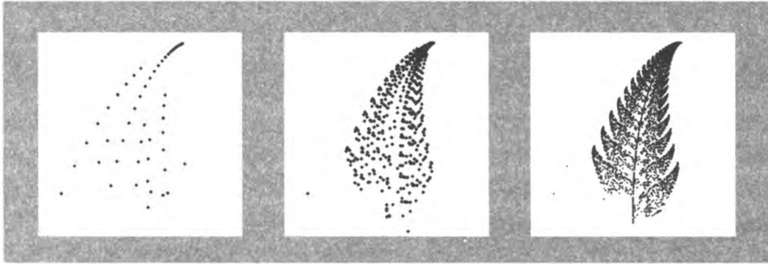
```
% more coral.txt
3
  .40 .15 .45
3 3
  .3077 -.5315  .8863
  .3077 -.0769  .2166
  .0000  .5455  .0106
3 3
 -.4615 -.2937 1.0962
  .1538 -.4476  .3384
  .6923 -.1958  .3808
```

```
% java IFS 20000 < coral.txt
```



Примеры фракталов

Тот факт, что одна и та же короткая программа, получающая несколько чисел из стандартного ввода и отображающая точки с использованием библиотеки стандартной графики (с разными данными), рисует как треугольник Серпинского, так и папоротник Барнсли (и много, много других изображений), воистину заме-



Генерирование папоротника Барнсли

чателен. Из-за простоты и привлекательного вида результатов такие вычисления применяются при синтезе реалистичных графических объектов для фильмов и компьютерных игр.

Пожалуй, еще важнее, что способность построения таких реалистичных изображений подводит нас к интересным научным вопросам: что эти вычисления говорят нам о природе? Что сама природа говорит нам об этих вычислениях?

Статистика

А теперь рассмотрим библиотеку для математических вычислений и базовых средств визуализации, которые встречаются во всевозможных научных и технических приложениях, но не реализованы в стандартных библиотеках Java. Эти вычисления связаны с задачей анализа статистических свойств множеств чисел. Например, такая библиотека может пригодиться при проведении серии научных экспериментов, возвращающих количественные результаты. Одна из самых серьезных проблем, с которыми сталкиваются современные ученые, — это организация анализа таких данных, и важность этой задачи непрерывно растет. В следующей таблице приведена сводка основных статистических методов анализа данных, которые будут включены в библиотеку:

`public class StdStats`	
`double max(double[] a)`	наибольшее значение
`double min(double[] a)`	наименьшее значение
`double mean(double[] a)`	математическое ожидание
`double var(double[] a)`	дисперсия выборки
`double stddev(double[] a)`	среднеквадратическое отклонение выборки
`double median(double[] a)`	медиана
`void plotPoints(double[] a)`	выводит точки (i, a[i])
`void plotLines(double[] a)`	выводит линии, соединяющие точки (i, a[i])
`void plotBars(double[] a)`	формирует столбцы диаграммы для точек (i, a[i])

Примечание: в библиотеку включены перегруженные реализации для других числовых типов.

API нашей библиотеки статических методов анализа данных

Основные статистические характеристики

Допустим, имеются данные измерений для n точек: $x_0, x_1, ..., x_{n-1}$. Среднее арифметическое этих измерений, также называемое *математическим ожиданием*, вычисляется по формуле $\mu = (x_0 + x_1 + ... + x_{n-1})/n$; это оценка наиболее вероятного ожидаемого значения измеряемой величины. Далее, при анализе представляют интерес минимальное и максимальное значение, а также *медиана* (значение, меньшее половины и большее другой половины всех значений). Еще одна важная характеристика — *дисперсия* выборки, которая вычисляется по формуле

$$\sigma^2 = ((x_0 - \mu)^2 + (x_1 - \mu)^2 + ... + (x_{n-1} - \mu)^2) / (n - 1),$$

и *среднеквадратическое отклонение* выборки — квадратный корень из дисперсии. Программа `StdStats` (листинг 2.2.4) содержит реализации статических методов для вычисления этих базовых статистических характеристик (медиана вычисляется сложнее других показателей — мы рассмотрим реализацию `median()` в разделе 4.2). Тестовый головной метод `main()` для `StdStats` читает числа из стандартного потока ввода в массив и вызывает каждый из методов для вывода минимума, математического ожидания, максимума и среднеквадратического отклонения:

```
public static void main(String[] args)
{
   double[] a = StdArrayIO.readDouble1D();
   StdOut.printf("      min %7.3f\n", min(a));
   StdOut.printf("     mean %7.3f\n", mean(a));
   StdOut.printf("      max %7.3f\n", max(a));
   StdOut.printf("  std dev %7.3f\n", stddev(a));
}
```

Как и в случае с `StdRandom`, здесь необходимо провести более серьезное тестирование вычислений (см. упражнение 2.2.3). Как правило, при отладке или тестировании новых методов в библиотеке вносятся соответствующие изменения в код тестов модуля, а методы тестируются поочередно. Более мощной и шире используемой библиотеке, такой как `StdStats`, нужно также тестовое приложение-клиент для проведения тщательного стрессового тестирования после внесения любых изменений. Если вас интересует, как может выглядеть такой клиент, ознакомьтесь с его кодом для `StdStats` на сайте книги. Любой опытный программист скажет вам, что время, потраченное на тесты модулей и на стрессовые тесты, многократно окупится в будущем.

Графический вывод

Одно из важных применений `StdDraw` — помощь в визуализации данных вместо использования таблиц чисел. В типичной ситуации мы проводим эксперименты, сохраняем экспериментальные данные в массиве, а затем сравниваем результаты с моделью — возможно, с математической функцией, описывающей данные. Чтобы ускорить этот процесс для типичной ситуации со значениями, разделенными интервалами равной длины, в библиотеку `StdStats` включены статические методы

Листинг 2.2.4. Библиотека анализа данных

```
public class StdStats
{
   public static double max(double[] a)
   {  // Определение максимального элемента a[].
      double max = Double.NEGATIVE_INFINITY;
      for (int i = 0; i < a.length; i++)
         if (a[i] > max) max = a[i];
      return max;
   }

   public static double mean(double[] a)
   {  // Вычисление среднего арифметического значений a[].
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
         sum = sum + a[i];
      return sum / a.length;
   }

   public static double var(double[] a)
   {  // Вычисление дисперсии для значений a[].
      double avg = mean(a);
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
         sum += (a[i] - avg) * (a[i] - avg);
      return sum / (a.length - 1);
   }

   public static double stddev(double[] a)
   {  return Math.sqrt(var(a));  }
   // Методы графического вывода приведены в листинге 2.2.5.

   public static void main(String[] args)
   {  /* См. текст. */  }
}
```

Реализации методов для вычисления максимума, минимума, дисперсии и среднеквадратичного отклонения чисел в заданном массиве. Метод вычисления минимума не приводится; методы графического вывода приведены в листинге 2.2.5; метод median() описан в упражнении 4.2.20.

```
% more tiny1D.txt
5
3.0 1.0 2.0 5.0 4.0
```

```
% java StdStats < tiny1D.txt
    min   1.000
   mean   3.000
    max   5.000
std dev   1.581
```

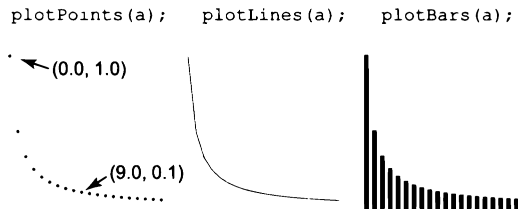
для вывода данных из массивов в наглядном виде. Листинг 2.2.5 содержит реализации методов `plotPoints()`, `plotLines()` и `plotBars()` для библиотеки `StdStats`. Эти методы выводят значения из массива, переданного в качестве аргумента, в окне графического вывода через равные промежутки, соединяя их отрезками

Листинг 2.2.5. Графическое представление значений данных в массиве

```
public static void plotPoints(double[] a)
{  // Отображение в виде точек (i, a[i]).
   int n = a.length;
   StdDraw.setXscale(-1, n);
   StdDraw.setPenRadius(1/(3.0*n));
   for (int i = 0; i < n; i++)
      StdDraw.point(i, a[i]);
}

public static void plotLines(double[] a)
{  //  Построение ломаной линии через точки (i, a[i]).
   int n = a.length;
   StdDraw.setXscale(-1, n);
   StdDraw.setPenRadius();
   for (int i = 1; i < n; i++)
      StdDraw.line(i-1, a[i-1], i, a[i]);
}

public static void plotBars(double[] a)
{  // Отображение столбцов гистограммы от (0, a[i]) до (i, a[i]).
   int n = a.length;
   StdDraw.setXscale(-1, n);
   for (int i = 0; i < n; i++)
      StdDraw.filledRectangle(i, a[i]/2, 0.25, a[i]/2);
}
```

Реализация трех методов StdStats (листинг 2.2.4) для наглядного представления данных. Они отображают точки (i, a[i]) точек, графика (ломаной линии) или гистограммы (столбчатой диаграммы) соответственно.

```
int n = 20;
double[] a = new double[n];
for (int i = 0; i < n; i++)
   a[i] = 1.0/(i+1);
```



прямой (`plotLines()`) и отображая точки (`plotPoints()`) для каждого значения или прямоугольники от оси x до значения (`plotBars()`). Во всех случаях на диаграмму наносятся точки с x-координатой `i` и y-координатой `a[i]`. Кроме того, все методы масштабируют x для заполнения окна графического вывода (чтобы точки были равномерно распределены по оси x); о масштабировании по оси y должна позаботиться программа-клиент.

Эта библиотека не претендует на роль графического пакета общего назначения, но вы без труда придумаете для нее разные расширения: разные виды точек, метки осей, цвета и многие другие функции, встречающиеся в современных системах графического представления данных. В некоторых ситуациях могут потребоваться более сложные методы.

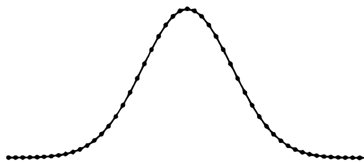
Разрабатывая библиотеку `StdStats`, мы стремились познакомить вас с анализом данных и одновременно показать, как легко создаются библиотеки для всевозможных полезных вычислений. В самом деле, библиотеки уже доказали свою полезность — мы использовали методы графического вывода для построения приведенных в книге иллюстраций с графиками функций, звуковыми волнами и результатами экспериментов. А сейчас рассмотрим несколько примеров их использования.

Построение графиков функций

Методы `StdStats.plot*()` могут использоваться для построения графиков произвольных функций: выберите интервал по оси x, на котором вы хотите построить график функции, вычислите значения функции, равномерно распределенные по этому интервалу, и сохраните их в массиве. Определите и назначьте масштаб по оси y, вызовите `StdStats.plotLines()` или другой метод `plot*()`. Например, для построения графика синуса измените масштаб по оси y так, чтобы на ней были представлены значения в диапазоне от -1 до $+1$. Масштабирование по оси x автоматически выполняется самими методами `StdStats`. Если диапазон значений функции неизвестен, проблему можно решить вызовом:

```
StdDraw.setYscale(StdStats.min(a), StdStats.max(a));
```

```
int n = 50;
double[] a = new double[n+1];
for (int i = 0; i <= n; i++)
   a[i] = Gaussian.pdf(-4.0 + 8.0*i/n);
StdStats.plotPoints(a);
StdStats.plotLines(a);
```



Построение графика функции

Плавность кривой определяется свойствами функции и количеством точек, использованных для построения графика. Как упоминалось при исходном описании `StdDraw`, количество точек должно быть достаточным для отражения резких выбросов в функции. Другой подход к построению графиков функций с точками дискретизации, разделенными интервалами разной величины, представлен в разделе 2.4.

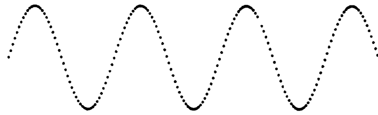
Отображение звуковой волны

И библиотека `StdAudio`, и методы графического вывода `StdStats` работают с массивами, содержащими значения отсчетов с постоянными интервалами дискретизации. В частности, для отображения звуковых волн в разделе 1.5 и в начале этого раздела мы сначала задали масштаб по оси y вызовом `StdDraw.setYscale(-1, 1)`, а затем нанесли точки на график вызовами `StdStats.plotPoints()`.

Как вы убедились, такие графики позволяют составить наглядное представление об операциях, выполняемых при обработке звука. Вы также можете создать интересные эффекты, строя графики звуковых волн при воспроизведении их в `StdAudio`, хотя эта задача достаточно сложная из-за огромного объема задействованных данных (см. упражнение 1.5.23).

```
StdDraw.setYscale(-1.0, 1.0);
double[] hi;
hi = PlayThatTune.tone(880, 0.01);
StdStats.plotPoints(hi);
```


```
StdDraw.setYscale(-1.0, 1.0);
double[] hi;
hi = PlayThatTune.tone(880, 0.01);
StdStats.plotPoints(hi);
```

Построение графика звуковой волны

Отображение результатов экспериментов

На одном изображении можно совместить сразу несколько графиков. Один из типичных примеров — сравнение экспериментальных результатов с теоретической моделью. Например, программа `Bernoulli` (листинг 2.2.6) подсчитывает количество «орлов», выпавших при `n` бросках монеты, и сравнивает результат с теоретической гауссовой функцией плотности вероятности. Известный результат теории вероятностей гласит, что эта величина имеет *биномиальное распределение*, которое очень хорошо аппроксимируется гауссовым распределением с математическим ожиданием $n/2$ и среднеквадратическим отклонением $\sqrt{n}/2$. Чем больше испытаний выполняется, тем точнее будет приближение.

Гистограмма, построенная программой `Bernoulli`, в компактной форме представляет результаты эксперимента и убедительно подтверждает теорию. Это классический пример научного подхода к прикладному программированию; такой подход часто встречается в этой книге, и вы должны использовать его при проведении экспериментов. Если существует теоретическая модель, которая может объяснить ваши результаты, то наглядное сравнение экспериментальных данных с теорией может подтвердить правильность обоих.

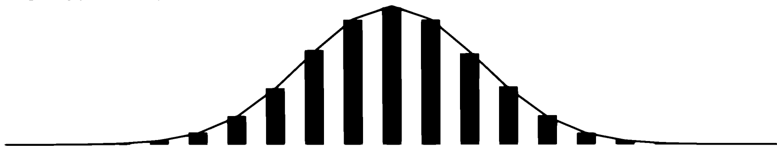
Листинг 2.2.6. Испытания Бернулли

```
public class Bernoulli
{
   public static int binomial(int n)
   {  // Моделирование n бросков монеты; возвращает количество "орлов".
      int heads = 0;
      for (int i = 0; i < n; i++)
         if (StdRandom.bernoulli(0.5)) heads++;
      return heads;
   }
   public static void main(String[] args)
   { // Проведение испытаний по схеме Бернулли,
     // построение гистограммы для результатов эксперимента
     // и графика для математической модели.
      int n = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);

      int[] freq = new int[n+1];
      for (int t = 0; t < trials; t++)
         freq[binomial(n)]++;

      double[] norm = new double[n+1];
      for (int i = 0; i <= n; i++)
         norm[i] = (double) freq[i] / trials;
      StdStats.plotBars(norm);

      double mean   = n / 2.0;
      double stddev = Math.sqrt(n) / 2.0;
      double[] phi  = new double[n+1];
      for (int i = 0; i <= n; i++)
         phi[i] = Gaussian.pdf(i, mean, stddev);
      StdStats.plotLines(phi);
   }
}
```

`n`	количество бросков в испытании
`trials`	количество испытаний
`freq[]`	результаты эксперимента
`norm[]`	нормализованные результаты
`phi[]`	гауссова модель

Эта клиентская программа, используя StdStats, StdRandom и Gaussian, наглядно доказывает, что количество «орлов», выпавших при n бросков, подчиняется распределению Гаусса.

```
% java Bernoulli 20 100000
```



Мы привели эти примеры для того, чтобы вы поняли, какие возможности открывает хорошо спроектированная библиотека статических методов для анализа данных. В упражнениях рассматриваются некоторые расширения и другие полезные идеи.

Библиотека `StdStats` пригодится при построении простых графиков. Не бойтесь экспериментировать с реализациями, изменять их или добавлять методы, создавая собственную библиотеку, которая строит графики по вашему усмотрению. В будущем, работая с постоянно расширяющимся кругом задач, вы естественным образом придете к идее разработки подобных инструментов для собственного использования.

Модульное программирование

Реализации библиотек, которые мы разработали, демонстрируют стиль программирования, называемый *модульным программированием*. Вместо того чтобы писать для каждой новой задачи новую программу, которая хранится в отдельном файле, вы разбиваете задачу на несколько меньших, более удобных подзадач, а потом реализуете и независимо отлаживаете код для решения каждой подзадачи. Хорошие библиотеки способствуют модульному программированию, позволяя разработчику определять и предоставлять решения подзадач, важных для будущих клиентов. *Всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это.* Java поддерживает такое разделение, позволяя нам независимо отлаживать классы в отдельных файлах, а позднее использовать их в программах. Традиционно программисты использовали термины «модуль» для обозначения кода, который может компилироваться и выполняться независимо; в языке Java модулем является каждый *класс*.

Программа `IFS` (листинг 2.2.3) служит примером модульной программы. Эти относительно сложные вычисления реализуются в относительно малых модулях, разрабатываемых независимо. Они используют уже привычные вам `StdRandom` и `StdArrayIO`, а также методы из `Integer` и `StdDraw`. Если бы весь код, необходимый для `IFS`, размещался в одном файле, то нам пришлось бы заниматься отладкой и сопровождением большого объема кода; при использовании же модульного программирования вы можете изучать фракталы, не сомневаясь в том, что массивы читаются правильно, а генератор случайных чисел выдает значения с правильным распределением, потому что мы уже реализовали и тестировали код этих задач в отдельных модулях.

Аналогично, программа `Bernoulli` (листинг 2.2.6) тоже является примером модульной программы. Она выполняла функции клиента для `Gaussian`, `Integer`, `Math`,

API	*Описание*
`Gaussian`	Функции для гауссова распределения
`StdRandom`	Случайные числа
`StdArrayIO`	Ввод и вывод с массивами
`IFS`	Клиент для фрактальных функций
`StdStats`	Функции для анализа данных
`Bernoulli`	Клиент для тестов распределения Бернулли

Сводка классов в этом разделе

`StdRandom` и `StdStats`. И снова мы можем с уверенностью утверждать, что методы этих модулей выдают ожидаемые результаты, потому что это системные библиотеки или библиотеки, которые мы тестировали, отладили и использовали ранее.

Для описания отношений между модулями в модульной программе часто рисуется *граф зависимостей*, в котором имена двух классов соединяются стрелкой с именем метода, если первый класс содержит вызов метода, а второй класс содержит его определение. Такие диаграммы играют важную роль, потому что понимание отношений между модулями необходимо для правильной организации разработки и сопровождения программ.

Граф зависимостей (неполный) для модулей этого раздела

Мы неоднократно подчеркиваем важность модульного программирования в этой книге, потому что оно обеспечивает ряд важных преимуществ, необходимых в современном программировании. Вот лишь некоторые из них.

- Работа с программами удобного размера — даже в больших системах.
- Отладка ограничивается небольшими фрагментами кода.
- Возможность повторного использования кода без его повторной реализации.
- Сопровождение (и доработка) кода существенно упрощается.

Важность этих преимуществ трудно переоценить, поэтому мы более подробно рассмотрим каждое из них.

Программы разумного размера

Ни одна большая задача не бывает настолько сложной, чтобы ее нельзя было разбить на несколько меньших подзадач. Если ваша программа занимает более

нескольких страниц кода, задайте себе следующие вопросы: существуют ли подзадачи, которые можно реализовать отдельно? Можно ли логично сгруппировать несколько таких подзадач в отдельной библиотеке? Могут ли другие клиенты использовать этот код в будущем? А в противоположной ситуации, если у вас появится множество мелких модулей, задайте себе другие вопросы: существует ли группа подзадач, логически принадлежащих одному модулю? Будет ли каждый модуль использоваться несколькими клиентами? Жестких правил по поводу размера модуля не существует: одна реализация критически важной абстракции может занимать всего несколько строк кода, а другая библиотека с большим количеством перегруженных методов может растянуться на сотни строк кода.

Отладка

Трассировка программы резко усложняется с ростом количества контролируемых операций и переменных, с которыми они работают. Трассировка программы с сотнями переменных требует отслеживания сотен значений, так как любая операция может повлиять на любую переменную (или оказаться под ее влиянием.) С сотнями и тысячами операций ситуация выходит из-под контроля. С модульным программированием и нашим правилом о максимально возможном ограничении области видимости переменных мы существенно сокращаем количество взаимосвязей, которые придется рассматривать во время отладки. Не менее важна идея контракта между клиентом и реализацией. Когда вы удостоверитесь в том, что реализация выполняет свою часть контракта, в ходе отладки каждого клиента уже можно рассчитывать на это.

Повторное использование кода

После того как вы реализуете такие библиотеки, как `StdStats` и `StdRandom`, вам уже не придется заботиться о написании кода для вычисления математического ожидания, среднеквадратического отклонения или генерации случайных чисел — можно просто повторно воспользоваться уже написанным кодом. Более того, вам не придется копировать этот код; любой модуль может запросто обратиться к любому `public`-методу любого другого модуля.

Сопровождение

Хорошую программу, как и хорошее литературное произведение, всегда можно усовершенствовать, а модульное программирование упрощает процесс постоянного совершенствования ваших программ Java, потому что доработка модуля улучшает все его клиенты. Например, обычно существует несколько разных подходов к решению конкретной задачи. С модульным программированием вы можете реализовать несколько подходов и опробовать их независимо друг от друга. Еще важнее другое: предположим, что в ходе разработки нового клиента в каком-то модуле будет обнаружена ошибка. При использовании модульного программирования

исправление этой ошибки фактически означает исправление ошибки сразу во всех клиентах модуля.

Если вам попадется старая программа (или новая программа, написанная старым программистом!), вы с большой вероятностью найдете в ней один огромный модуль — сплошную последовательность строк кода на много страниц, где в любой строке может быть обращение к любой переменной в программе. Старые программы такого рода часто встречаются в критических компонентах вычислительной инфраструктуры (например, на ядерных электростанциях и в некоторых банках) именно потому, что программисты, которым было поручено сопровождение кода, недостаточно хорошо понимали программу, чтобы просто переписать ее на современном языке! Современные языки с поддержкой модульного программирования (такие, как Java) способствуют предотвращению подобных ситуаций за счет раздельной разработки библиотек методов в независимых классах. Возможность совместного использования статических методов из разных файлов принципиально расширяет модель программирования в двух отношениях. Во-первых, она позволяет повторно использовать код без создания нескольких его копий. Во-вторых, она позволяет разбить программу на файлы удобного размера с возможностью независимой отладки и компиляции, а это в очередной раз подчеркивает наш исходный посыл: *всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это.*

В этом разделе мы дополнили библиотеки `Std*` из раздела 1.5 несколькими другими библиотеками, которыми вы теперь можете пользоваться: `Gaussian`, `StdArrayIO`, `StdRandom` и `StdStats`. Их применение было продемонстрировано на нескольких примерах. Центральное место в них занимают основные математические концепции, которые встречаются в любой научной или технической задаче. Наша цель — не только предоставить готовые инструменты, но и показать, как легко создать свои собственные. Первый вопрос, который чаще всего задают современные программисты, сталкиваясь со сложной задачей: «Какие инструменты мне понадобятся?» Если необходимых инструментов под рукой нет, второй вопрос: «Насколько сложно будет их реализовать?» Для хорошего программиста равно необходимы как готовность самому создать нужные ему программные инструменты, так и здравый смысл, чтобы искать готовые решения в библиотеках.

После библиотек и модульного программирования вам придется сделать еще один шаг в изучении современной модели программирования: объектно-ориентированное программирование, которое является темой главы 3. С объектно-ориентированным программированием вы сможете строить библиотеки функций с побочными эффектами (находящимися под жестким контролем вашей программы) для колоссального расширения модели программирования Java. Прежде чем переходить к объектно-ориентированному программированию, мы рассмотрим в этой главе глубокие последствия идеи о том, что метод может вызывать сам себя (раздел 2.3), и пример модульного программирования (2.4), более масштабный по сравнению с небольшими программами из этого раздела.

Вопросы и ответы

В. Я попытался использовать `StdRandom`, но получил сообщение об ошибке: `Exception in thread "main" java.lang.NoClassDefFoundError: StdRandom`. В чем дело?

О. Вы должны предоставить доступ к `StdRandom` для среды Java. См. первый вопрос в разделе «Вопросы и ответы» в конце раздела 1.5.

В. Существует ли ключевое слово, которое явно определяет класс как библиотеку?

О. Нет, подходит любой набор `public`-методов. На самом деле здесь существуют определенные концептуальные различия, потому что одно дело — создать файл .java, который вы будете компилировать и запускать, другое — создать файл .java, который вам понадобится в будущем, и третье — создать файл .java, который будет использоваться *кем-то другим*. Прежде чем браться за такую работу, которой обычно занимаются опытные системные программисты, необходимо попрактиковаться в разработке библиотек для собственного использования.

В. Как разработать новую версию библиотеки, которой я пользовался в течение некоторого времени?

О. С осторожностью. Любое изменение в API может нарушить работу любой программы, использующей эту библиотеку, поэтому лучше сохранять результаты в отдельном каталоге. Применяя такой подход, вы работаете не с самим кодом, а с его копией. Если вы вносите изменения в библиотеку с множеством клиентов, вы оцените масштаб проблем, с которыми сталкиваются компании при выпуске новых версий своих программных продуктов. Если нужно добавить в библиотеку несколько новых методов, действуйте: обычно это не создает особых проблем, хотя следует помнить, что вам, возможно, придется заниматься поддержкой этой библиотеки годами!

В. Как я узнаю, что реализация работает правильно? Почему нельзя автоматически проверить, что вызов удовлетворяет API?

О. Мы используем неформальные спецификации, потому что написание подробной спецификации почти не отличается от написания программы. Более того, один из фундаментальных принципов компьютерной теории утверждает, что даже это не решит базовую проблему, потому что в общем случае проверить, что две программы выполняют одни и те же вычисления, невозможно.

Упражнения

2.2.1. Добавьте в программу `Gaussian` (листинг 2.1.2) реализацию статического метода `pdf(x, mu, sigma)` с тремя аргументами, указанного в API. Метод вычисляет гауссову функцию плотности вероятности с заданным математическим ожиданием μ и среднеквадратическим отклонением σ по формуле $\phi(x, \mu, \sigma) = \phi((x - \mu) / \sigma)/\sigma$.

Также добавьте реализацию гауссовой функции распределения `cdf(z, mu, sigma)`, основанную на формуле $\Phi(z, \mu, \sigma) = ((z - \mu)/\sigma)$.

2.2.2. Напишите библиотеку статических методов, которая реализует гиперболические функции на основании определений $\sinh(x) = (e^x - e^{-x}) / 2$ и $\cosh(x) = (e^x - e^{-x})/2$. Функции $\tanh(x)$, $\coth(x)$, $\mathrm{sech}(x)$ и $\mathrm{csch}(x)$ определяются по аналогии со стандартными тригонометрическими функциями.

2.2.3. Напишите тестовую программу-клиент для `StdStats` и `StdRandom`, которая проверяет, что методы обеих библиотек работают именно так, как ожидается. Получите аргумент командной строки *n*, сгенерируйте *n* случайных чисел с использованием каждого из методов `StdRandom` и выведите их статистику. Дополнительное задание: обоснуйте полученные результаты, сравнив их с результатами, предполагаемыми на основании анализа.

2.2.4. Добавьте в `StdRandom` метод с именем `shuffle()`, который получает массив значений `double` в аргументе и переставляет эти элементы в случайном порядке. Реализуйте тестовую клиентскую программу, которая проверяет, что каждая перестановка массива генерируется примерно одинаковое число раз. Добавьте перегруженные методы, получающие массивы целых чисел и строк.

2.2.5. Разработайте клиентскую программу для стрессового тестирования `StdRandom`. Обратите особое внимание на метод `discrete()`. Например, равна ли сумма всех вероятностей 1?

2.2.6. Напишите статический метод, который получает в качестве аргументов значения `ymin` и `ymax` типа `double` (причем `ymin` строго меньше `ymax`) и массив `a[]` типа `double` и использует библиотеку `StdStats` для линейного масштабирования значений в массиве `a[]`так, чтобы все они лежали в диапазоне от `ymin` до `ymax`.

2.2.7. Напишите клиентскую программу для `Gaussian` и `StdStats`, который анализирует эффект от изменения математического ожидания и среднеквадратического отклонения для гауссовой функции плотности вероятности. Постройте один график с гауссовыми распределениями, имеющими фиксированное математическое ожидание с разными среднеквадратичными отклонениями, и другой график с гауссовыми распределениями, имеющими фиксированное среднеквадратическое отклонение с разным математическим ожиданием.

2.2.8. Добавьте в `StdRandom` метод `exp()`, который получает аргумент λ и возвращает случайное число из *экспоненциального распределения* с параметром λ. *Подсказка*: если x — случайное число с равномерным распределением от 0 до 1, то $-\ln x/\lambda$ — случайное число из экспоненциального распределения с параметром λ.

2.2.9. Добавьте в `StdRandom` статический метод `maxwellBoltzmann()`, который возвращает случайное значение из распределения *Максвелла — Больцмана* с параметром σ. Чтобы получить такое значение, верните квадратный корень из суммы квадратов трех случайных чисел с гауссовым распределением с математическим ожиданием 0 и среднеквадратическим отклонением σ. Распределению Максвелла — Больцмана подчиняется скорость молекул идеального газа.

2.2.10. Измените программу `Bernoulli` (листинг 2.2.6) так, чтобы гистограмма перерисовывалась после каждого эксперимента и вы могли бы следить за тем, как он сходится к гауссовому распределению. Затем добавьте аргумент командной строки и перегруженную реализацию `binomial()`, которая позволяет определить вероятность выпадения «орла» на несимметричной монете p, и проведите эксперименты, чтобы получить представление о распределении для несимметричной монеты. Обязательно проверьте значения p, близкие к 0 и близкие к 1.

2.2.11. Разработайте полную реализацию `StdArrayIO` (реализуйте все 12 методов, перечисленные в API).

2.2.12. Напишите библиотеку `Matrix`, которая реализует следующий API:

public class Matrix	
`double dot(double[] a, double[] b)`	Скалярное произведение векторов
`double[][] multiply(double[][] a, double[][] b)`	Произведение матриц
`double[][] transpose(double[][] a)`	Транспонирование
`double[] multiply(double[][] a, double[] x)`	Умножение матрицы на вектор
`double[] multiply(double[] x, double[][] a)`	Умножение вектора на матрицу

(См. раздел 1.4.) Для тестирования используйте следующий код, который выполняет те же вычисления, что и программа `Markov` (листинг 1.6.3):

```
public static void main(String[] args)
{
   int trials = Integer.parseInt(args[0]);
   double[][] p = StdArrayIO.readDouble2D();
   double[] ranks = new double[p.length];
   rank[0] = 1.0;
   for (int t = 0; t < trials; t++)
       ranks = Matrix.multiply(ranks, p);
   StdArrayIO.print(ranks);
}
```

Математики и ученые используют для таких задач проверенные библиотеки или специализированные языки для работы с матрицами. За подробностями об использовании таких библиотек обращайтесь на сайт книги.

2.2.13. Напишите тестовую программу-клиент для библиотеки `Matrix`, которая реализует версию программы `Markov`, описанную в разделе 1.6, но использует возведение матрицы в квадрат вместо многократного умножения вектора на матрицу.

2.2.14. Перепишите программу `RandomSurfer` (листинг 1.6.2) с использованием библиотек `StdArrayIO` и `StdRandom`.

Часть решения

```
...
double[][] p = StdArrayIO.readDouble2D();
int page = 0; // Начинаем со страницы 0.
```

```
int[] freq = new int[n];
for (int t = 0; t < trials; t++)
{
   page = StdRandom.discrete(p[page]);
   freq[page]++;
}
...
```

Упражнения повышенной сложности

2.2.15. *Игральные кости Зихермана.* Имеются два шестигранных кубика: грани одного помечены цифрами 1, 3, 4, 5, 6 и 8, а грани другого — 1, 2, 2, 3, 3 и 4. Сравните вероятности выпадения каждого значения суммы для такой пары с вероятностями выпадения тех же значений для обычной пары кубиков. Используйте библиотеки `StdRandom` и `StdStats`.

2.2.16. *Крэпс.* Ниже приведены правила ставок в игре «крэпс». Бросьте два кубика; пусть x — их сумма.

- Если x равно 7 или 11, вы выиграли.
- Если x равно 2, 3 или 12, вы проиграли.

В остальных случаях бросайте два кубика до тех пор, пока не выпадет x или 7.

- Если сумма равна x, вы выиграли.
- Если сумма равна 7, вы проиграли.

Напишите модульную программу для оценки вероятности выигрыша. Измените программу так, чтобы она поддерживала шулерские игральные кости, у которых вероятность выпадения 1 передается как аргумент командной строки, вероятность выпадения 6 равна 1/6 минус переданная вероятность, а значения 2–5 равновероятны. *Подсказка*: используйте метод `StdRandom.discrete()`.

2.2.17. *Гауссовы случайные значения.* Реализуйте функцию `gaussian()` в `StdRandom` (листинг 2.2.1), используя формулу Бокса — Мюллера (см. упражнение 1.2.27). Затем рассмотрите альтернативное решение — *метод Марсальи*, основанный на генерировании случайной точки на единичном круге и использовании разновидности формулы Бокса — Мюллера (см. обсуждение `do-while` в конце раздела 1.3).

```
public static double gaussian()
{
   double r, x, y;
   do
   {
      x = uniform(-1.0, 1.0);
      y = uniform(-1.0, 1.0);
      r = x*x + y*y;
   } while (r >= 1 || r == 0);
   return x * Math.sqrt(-2 * Math.log(r) / r);
}
```

Для каждого решения сгенерируйте 10 миллионов случайных значений из гауссова распределения. Измерьте, какое из решений работает быстрее.

2.2.18. *Динамическая гистограмма.* Предположим, стандартный поток ввода содержит последовательность значений `double`. Напишите программу, которая получает из командной строки целое число `n` и два значения `lo` и `hi` типа `double` и использует библиотеку `StdStats` для построения гистограммы количества чисел, попадающих в каждый из `n` интервалов, определяемых делением `(lo, hi)` на `n` равных отрезков. Используйте свою программу для добавления в упражнение 2.2.3 кода, строящего гистограмму распределения чисел, сгенерированных каждым методом. Значение `n` должно передаваться в командной строке.

2.2.19. *Стрессовое тестирование.* Разработайте программу-клиент для стрессового тестирования `StdStats`. Работайте в паре с коллегой: один человек пишет код, другой его тестирует.

2.2.20. *Задача о разорении игрока.* Разработайте программу-клиент, использующую `StdRandom`, для анализа задачи о разорении игрока (см. программу 1.3.8 и упражнения 1.3.24–1.3.25). *Примечание*: определить статический метод для этого эксперимента сложнее, чем в программе `Bernoulli`, потому что метод не может вернуть два значения.

2.2.21. *IFS.* Поэкспериментируйте с разными вариантами входных данных программы `IFS` для построения собственных узоров, похожих на треугольник Серпинского, папоротник Барнсли или другие примеры, приведенные в тексте. Для начала поэкспериментируйте с незначительными изменениями входных данных.

2.2.22. *Матричная реализация IFS.* Напишите версию `IFS`, использующую статический метод `multiply()` из программы `Matrix` (см. упражнение 2.2.12) вместо формул для вычисления новых значений `x0` и `y0`.

2.2.23. *Библиотека для изучения свойств целых чисел.* Разработайте библиотеку на основании рассмотренных в этой книге функций для вычисления характеристик целых чисел. Включите функции для проверки того, является ли заданное целое число простым; для проверки того, являются ли два числа взаимно простыми; для факторизации (нахождения всех множителей) заданного целого числа; для вычисления наибольшего общего делителя и наименьшего общего кратного двух целых чисел; для вычисления функции Эйлера (упражнение 2.1.26); любых других функций, которые, на ваш взгляд, могут оказаться полезными. Включите перегруженные реализации для значений `long`. Создайте API, программу-клиент для стрессового тестирования, и программы для решения некоторых из предшествующих упражнений.

2.2.24. *Библиотека для работы с музыкой.* Разработайте библиотеку на базе функций `PlayThatTune` (листинг 2.1.4), которая может использоваться при написании клиентских программ для создания и обработки музыки.

2.2.25. *Машины для голосования.* Разработайте клиентскую программу `StdRandom` (с соответствующими статическими методами) для анализа следующей задачи: в выборах участвуют 100 миллионов избирателей, 51% голосует за кандидата A,

и 49% голосуют за кандидата B. Однако машины для голосования несовершенны, и в 5% случаев они выдают неправильный результат. Если все ошибки совершаются независимо и случайно, достаточно ли 5% ошибки для того, чтобы результат выборов мог быть определен неправильно? Какой процент ошибок не повлияет на результат?

2.2.26. *Покер.* Напишите программу-клиент, использующую `StdRandom` и `StdStats` (с соответствующими статическими методами), для оценки вероятности прихода основных покерных комбинаций (пара, две пары, тройка, 3+2, «флеш»[1]) на основе моделирования. Разделите свою программу на статические методы и обоснуйте свое решение. Дополнительное задание: добавьте в список комбинаций «стрит» (5 карт, номиналы которых составляют непрерывную последовательность) и «стрит-флеш» (стрит одной масти).

2.2.27. *Анимация диаграмм.* Напишите программу, которая получает аргумент командной строки `m` и строит столбчатую диаграмму `m` последних значений типа `double` в стандартном вводе. Используйте тот же прием создания анимации, который мы использовали в программе `BouncingBall` (листинг 1.5.6): стирание, перерисовка, отображение и короткая пауза. Каждый раз, когда программа читает новое число, она должна перерисовывать всю столбчатую диаграмму. Так как при постепенном сдвиге поступающих данных изменения в изображении становятся относительно небольшими, ваша программа будет использовать «окно» фиксированного размера, перемещающееся по последовательности входных данных. Используйте программу для представления очень большого файла данных, изменяющегося со временем (например, биржевых котировок).

2.2.28. *Библиотека для графического вывода массивов.* Разработайте собственные методы графического вывода — лучше тех, что представлены в `StdStats`. Подключите фантазию! Попробуйте создать библиотеку, которая пригодится в ваших будущих проектах.

2.3. Рекурсия

Раз функция может вызывать другие функции, логично предположить, что она также может вызывать саму себя. Механизм вызова функций в Java и в большинстве современных языков программирования предоставляет такую возможность; это называется *рекурсией*. В этом разделе мы рассмотрим примеры элегантных и эффективных рекурсивных решений для различных задач. Рекурсия — мощный прием программирования, часто используемый в этой книге. Рекурсивные программы часто получаются более компактными и понятными, чем их нерекурсивные аналоги. Немногие программисты осваивают рекурсию настолько хорошо, чтобы уверенно применять ее в работе постоянно, однако элегантное рекурсивное реше-

[1] Пять карт одной масти. — *Примеч. пер.*

ние задачи — маленькое жизненное удовольствие, доступное любому программисту (даже вам!).

Однако рекурсия — нечто большее, чем один из приемов программирования. Во многих случаях рекурсия является удобным способом описания реального мира. Например, рекурсивное дерево (на иллюстрации), напоминающее реальное, имеет естественное рекурсивное описание. Многие, многие явления хорошо объясняются рекурсивными моделями. В частности, рекурсия играет важнейшую роль в теории вычислений. Она предоставляет простую вычислительную модель, которая охватывает любые вычисления на любых компьютерах; она упрощает организацию и анализ программного кода и играет ключевую роль в многочисленных критически важных вычислительных приложениях, от комбинаторного поиска до древовидных структур данных, используемых при обработке информации, и быстрых преобразований Фурье для обработки сигналов.

Рекурсивная модель реального мира

В частности, рекурсию необходимо освоить еще и потому, что она предоставляет прямой и очевидный механизм построения простых математических моделей, которые могут использоваться для доказательства важных свойств вашей программы. Метод, который для этого будет использоваться, называется *математической индукцией*. Обычно в этой книге мы стараемся обходить математические подробности стороной, но в этом разделе вы увидите, что этот подход заслуживает вашего внимания.

Ваша первая рекурсивная программа

Аналогом программы «Hello, World» для рекурсии является функция вычисления *факториала*, определяемая для положительных целых чисел n формулой

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1.$$

Другими словами, $n!$ вычисляется как произведение положительных целых чисел, меньших или равных n. Вообще говоря, $n!$ легко вычисляется в цикле `for`, но еще проще воспользоваться следующей рекурсивной функцией:

```
public static long factorial(int n)
{
   if (n == 1) return 1;
   return n * factorial(n-1);

}
```

Функция вызывает сама себя. Реализация очевидным образом приводит к нужному результату. Чтобы убедиться в этом, достаточно заметить, что `factorial()` возвращает $1 = 1!$ для n, равного 1, и если функция правильно вычисляет значение

$$(n-1)! = (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

Functions and Modules

2.3 Recursion

The idea of calling one function from another immediately suggests the possibility of a function calling itself. The function-call mechanism in Python and in most modern programming languages supports this possibility, which is known as recursion. In this section, we will study examples of elegant and efficient recursive solutions to a variety of problems. Once you get used to the idea, you will see that recursion is a powerful general-purpose programming technique with many attractive properties. It is a fundamental tool that we use often in this book. Recursive programs are often more compact and easier to understand than their nonrecursive counterparts. Few programmers become sufficiently comfortable with recursion to use it in everyday code, but solving a problem with an elegantly crafted recursive program is a satisfying experience that is certainly accessible to every programmer (even you!).

2.3.1	Euclid's algorithm	295
2.3.2	Towers of Hanoi	298
2.3.3	Gray code	303
2.3.4	Recursive graphics	305
2.3.5	Brownian bridge	307

Programs in this section

A recursive model of the natural world

Recursion is much more than a programming technique. In many settings, it is a useful way to describe the natural world. For example, the recursive tree (to the left) resembles a real tree, and has a natural recursive description. Many, many phenomena are well explained by recursive models. In particular, recursion plays a central role in computer science. It provides a simple computational model that embraces everything that can be computed with any computer; it helps us to organize and to analyze programs; and it is the key to numerous critically important computational applications, ranging from combinatorial search to tree data structures that support information processing to the fast Fourier transform for signal processing.

One important reason to embrace recursion is that it provides a straightforward way to build simple mathematical models that we can use to prove important facts about our programs. The proof technique that we use to do so is known as mathematical induction. Generally, we avoid going into the details of mathematical proofs in this book, but you will see in this section that it is worthwhile to understand that point of view and to make the effort to convince yourself that recursive programs have the intended effect.

Рекурсия

то она также правильно вычислит и значение

$$n! = n \times (n-1)!$$
$$= n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

Чтобы вычислить `factorial(5)`, рекурсивная функция умножает 5 на `factorial(4)`; чтобы вычислить `factorial(4)`, она умножает 4 на `factorial(3)` и т. д. Процесс повторяется, пока не будет вызвана функция `factorial(1)`, которая возвращает непосредственно значение 1. Рекурсивные алгоритмы можно трассировать точно так же, как и любую другую последовательность вызовов функций. Так как все вызовы интерпретируются как независимые копии кода, факт их «вложенности» значения не имеет.

```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

Трассировка вызовов для factorial(5)

В нашей реализации `factorial()` можно выделить два компонента, присутствующих в каждой рекурсивной функции. Во-первых, *базовый случай* возвращает значение без последующих рекурсивных вызовов. Он делает это для одного или для нескольких специальных входных значений, для которых функция вычисляется без рекурсии. Для `factorial()` базовым случаем является $n = 1$. Во-вторых, центральное место в рекурсивной функции занимает *свертка*: результат функции для одного (или нескольких) аргументов связывается с результатом этой же функции, вычисленным для другого (или нескольких других) аргумента. Для `factorial()` свертка определяется формулой `n*factorial(n-1)`. Эти два компонента должны присутствовать во всех рекурсивных функциях. Кроме того, последовательность значений аргументов должна сходиться к базовому случаю. Для `factorial()` значение n уменьшается на 1 с каждым вызовом, поэтому последовательность значений аргументов сходится к базовому случаю $n = 1$.

```
 1 1
 2 2
 3 6
 4 24
 5 120
 6 720
 7 5040
 8 40320
 9 362880
10 3628800
11 39916800
12 479001600
13 6227020800
14 87178291200
15 1307674368000
16 20922789888000
17 355687428096000
18 6402373705728000
19 121645100408832000
20 2432902008176640000
```

Значения n! в формате long

Возможно, маленькая программа, такая как `factorial()`, станет яснее, если вынести свертку в условие `else`. Однако применение этого соглашения во всех рекурсивных программах приведет к излишнему усложнению больших программ, потому что большую часть кода (для свертки) придется размещать в фигурных скобках после `else`. Вместо этого мы используем другую схему: базовый случай всегда размещается сначала и завершается командой `return`, а оставшаяся часть кода посвящена свертке.

Эта реализация `factorial()` сама по себе не особенно полезна: $n!$ растет настолько быстро, что умножение вскоре приводит к переполнению типа `long` и при уже $n > 20$ выдаются некорректные значения. Однако рекурсия хорошо подходит для множества других функций. Например, рекурсивная функция

```
public static double harmonic(int n)
{
   if (n == 1) return 1.0;
   return harmonic(n-1) + 1.0/n;
}
```

вычисляет n-е гармоническое число (см. листинг 1.3.5) при малых n по следующей формуле:

$$\begin{aligned} H_n &= 1 + 1/2 + \ldots + 1/n \\ &= (1 + 1/2 + \ldots + 1/(n-1)) + 1/n \\ &= H_{n-1} + 1/n. \end{aligned}$$

Этот метод позволяет эффективно вычислить всего в нескольких строках кода значение любой конечной суммы (или произведения), для которой существует компактная формула. На самом деле такие рекурсивные функции представляют собой замаскированные циклы, но рекурсия помогает лучше понять суть выполняемых вычислений.

Математическая индукция

Рекурсивное программирование напрямую связано с *математической индукцией* — логическим методом, широко применяемым в доказательствах, относящихся к натуральным числам.

Доказательство методом математической индукции того, что некоторое утверждение, относящееся к целому n, истинно для любых n, состоит из двух шагов.

- *Базовый случай* (основание индукции): доказательство того, что утверждение истинно для некоторого конкретного значения или значений n (чаще всего это 0 или 1).
- *Индукционный переход* (центральная часть доказательства): предположите, что утверждение истинно для всех положительных целых чисел, меньших n, и используйте этот факт для доказательства того, что оно истинно для n.

Такое доказательство демонстрирует, что утверждение истинно для бесконечно большого количества значений n: вы начинаете с базового случая, а затем доказываете, что утверждение истинно для каждого большего значения n (одно за другим).

Обычно знакомство с методом индукции начинается с доказательства того, что сумма положительных целых чисел, меньших либо равных n, определяется формулой $n(n + 1) / 2$. Другими словами, мы пытаемся доказать, что следующее равенство истинно для всех $n \geq 1$:

$$1 + 2 + 3 \ldots + (n - 1) + n = n\,(n + 1) / 2.$$

Равенство очевидно истинно для $n = 1$ (базовый случай), потому что $1 = 1(1 + + 1) / 2$. Если предположить, что оно истинно для всех положительных целых чисел, меньших n, то оно заведомо истинно для $n - 1$, поэтому

$$1 + 2 + 3 \ldots + (n - 1) = (n - 1)\,n / 2.$$

Мы можем прибавить n к обеим сторонам равенства и упростить правую часть для получения нужной формулы (индукционный переход).

Каждый раз, когда вы пишете рекурсивную программу, примените математическую индукцию, чтобы убедиться в том, что программа приводит к желаемому эффекту. Соответствие между индукцией и рекурсией очевидно. Различие в терминологии указывает на изменение направленности: в рекурсивной программе мы стремимся выполнить вычисления, сводя текущую задачу к более простой, поэтому используем термин «свертка»; в доказательстве методом индукции мы стремимся установить истинность утверждения для более сложной задачи, поэтому используется термин «индукционный переход».

При написании рекурсивных программ обычно не нужно записывать полное формальное доказательство того, что оно приводит к желаемому результату, но наша работа всегда зависит от существования такого доказательства. Мы часто апеллируем к неформальному индукционному доказательству, чтобы убедиться в том, что рекурсивная программа работает именно так, как задумано. Например, мы ограничились неформальным доказательством того, что `factorial()` вычисляет произведение всех положительных целых чисел, меньших или равных n.

Алгоритм Евклида

Наибольший общий делитель (НОД) двух положительных целых чисел — наибольшее целое число, на которое без остатка делятся оба числа. Например, наибольшим общим делителем 102 и 68 является число 34, потому что и 102, и 68 делятся на 34 и не делятся без остатка ни на какое число, большее 34. Возможно, вы встречались с понятием наибольшего общего делителя, когда изучали сокращение дробей. Например, дробь 68/102 сокращается до 2/3 делением числителя и знаменателя на 34, их наибольший общий делитель. Вычисление НОД больших чисел — важная задача, встречающаяся во многих областях, в том числе и в знаменитой криптографической системе RSA.

Листинг 2.3.1. Алгоритм Евклида

```
public class Euclid
{
   public static int gcd(int p, int q)
   {
      if (q == 0) return p;
      return gcd(q, p % q);
   }
   public static void main(String[] args)
   {
      int p = Integer.parseInt(args[0]);
      int q = Integer.parseInt(args[1]);
      int divisor = gcd(p, q);
      StdOut.println(divisor);
   }
}
```

p, q	аргументы
divisor	наибольший общий делитель

Программа выводит наибольший общий делитель двух чисел, переданных как аргументы командной строки, с использованием рекурсивной реализации алгоритма Евклида.

```
% java Euclid 1440 408
24
% java Euclid 314159 271828
1
```

Для эффективного вычисления наибольшего общего делителя можно воспользоваться следующим свойством, истинным для положительных целых p и q:

Если $p > q$, то НОД чисел p и q равен НОД чисел q и p % q.

Чтобы убедиться в этом, сначала заметим, что НОД p и q равен НОД q и $p - q$, потому что и p, и q делятся на это число в том и только в том случае, если q и $p - q$ делятся на него. Из тех же соображений q и $p - 2q$, q и $p - 3q$ и т. д. имеют тот же НОД, а один из способов вычисления p % q основан на вычитании q из p до тех пор, пока не получится число, меньшее q.

```
gcd(1440, 408)
   gcd(408, 216)
      gcd(216, 192)
         gcd(192, 24)
            gcd(24, 0)
               return 24
            return 24
         return 24
      return 24
   return 24
```

Трассировка вызовов для gcd()

Статический метод `gcd()` из программы `Euclid` (листинг 2.3.1) — компактная рекурсивная функция, у которой свертка базируется на этом свойстве. В базовом случае значение q равно 0, при этом $gcd(p,0) = p$. Чтобы убедиться в том, что свертка сходится к базовому случаю, заметим, что значение второго аргумента уменьшается при каждом рекурсивном вызове, так как p % $q < q$. Если $p < q$, то первый рекурсивный вызов фактически меняет порядок двух аргументов. Значение второго аргумента уменьшается по крайней мере в 2 раза при каждом втором рекурсивном вызове, поэтому последовательность значений аргумента быстро сходится к базовому случаю (см. упражнение 2.3.11). Рекурсивное решение

задачи вычисления наибольшего общего делителя, называемое *алгоритмом Евклида*, принадлежит к числу старейших известных алгоритмов — ему более 2000 лет.

Ханойские башни

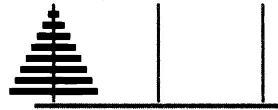
Ни одно обсуждение рекурсии не было бы полным без упоминания древней задачи о *ханойских башнях*. Пусть имеются три стержня и *n* дисков, которые надеваются на стержни. Диски отличаются по размерам и изначально находятся на одном из стержней; самый большой (диск *n*) находится внизу, а меньший (диск 1) находится наверху. Требуется переместить все *n* дисков на другой стержень с соблюдением следующих правил.

- За один раз перемещается только один диск.
- Больший диск никогда не кладется на меньший.

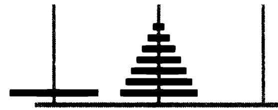
По легенде, некая группа монахов перекладывает 64 золотых диска на 3 алмазных стержнях, и когда все диски будут перемещены, наступит конец света. Но смогут ли монахи в принципе справиться с этой задачей без нарушения правил?

Решением задачи должна стать последовательность инструкций по перемещению дисков. Предполагается, что стержни выстроены в ряд и в каждой инструкции по перемещению диска указывается номер стержня и направление перемещения — налево или направо. Если диск лежит на левом стержне, то инструкция по перемещению налево означает переход к правому стержню; и наоборот, если диск лежит на правом стержне, инструкция по перемещению направо означает переход к левому стержню. Если все диски лежат на одном стержне, есть всего два возможных хода (перемещение самого маленького диска налево или направо); в противном случае существуют три возможных хода (перемещение самого маленького диска налево или направо или допустимый ход с двумя другими стержнями.) Выбор одной из этих возможностей для достижения цели — задача, требующая планирования. Рекурсия как раз и предоставляет такой план, основанный на следующей идее: сначала верхние $n - 1$ дисков перемещаются на пустой стержень, после чего самый большой диск перемещается на другой пустой стержень (там, где ему не будут мешать меньшие диски). Операция завершается перемещением $n - 1$ дисков на самый большой диск.

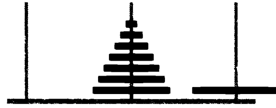
Исходная позиция



Перемещение n – 1 дисков направо (рекурсивное)



Перемещение самого большого диска налево (с переходом на правый стержень)



Перемещение n – 1 дисков направо (рекурсивное)

Рекурсивная схема решения задачи о ханойских башнях

Листинг 2.3.2. Ханойские башни

```
public class TowersOfHanoi
{
   public static void moves(int n, boolean left)
   {
      if (n == 0) return;
      moves(n-1, !left);
      if (left) StdOut.println(n + " left");
      else      StdOut.println(n + " right");
      moves(n-1, !left);
   }
   public static void main(String[] args)
   {  // Прочитать n, вывести инструкции для перемещения n дисков влево.
      int n = Integer.parseInt(args[0]);
      moves(n, true);
   }
}
```

n	количество дисков
left	направление перемещения стопки

Рекурсивный метод moves() выводит описания ходов для перемещения n дисков налево (если аргумент left равен true) или направо (аргумент left равен false).

```
% java TowersOfHanoi 1
1 left
% java TowersOfHanoi 2
1 right
2 left
1 right
% java TowersOfHanoi 3
1 left
2 right
1 left
3 left
1 left
2 right
1 left
```

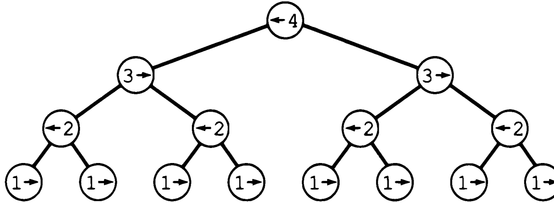
```
% java TowersOfHanoi 4
1 right
2 left
1 right
3 right
1 right
2 left
1 right
4 left
1 right
2 left
1 right
3 right
1 right
2 left
1 right
```

Программа `TowersOfHanoi` (листинг 2.3.2) является прямой реализацией этой рекурсивной стратегии. Она получает аргумент командной строки `n` и выводит решение задачи для `n` дисков. Рекурсивная функция `moves()` выводит последовательность ходов для перемещения стопки дисков на левый (если аргумент `left` равен `true`) или на правый (если аргумент `left` равен `false`.) Для этого она действует по только что описанной схеме.

Дерево вызовов функций

Чтобы лучше понять поведение модульной программы с множеством рекурсивных вызовов (такой, как `TowersOfHanoi`), мы воспользуемся специальным визуальным

представлением — *деревом вызовов функций*. А именно каждый вызов метода представляется узлом дерева, который изображается в виде кружка со значениями аргументов этого вызова. Под каждым узлом рисуются узлы дерева, соответствующие каждому вызову из этого метода (по порядку слева направо), и соединительные линии. Поскольку такая схема включает в себя узлы для всех вызовов функции, она представляет полную информацию для понимания поведения программы.



Дерево вызовов для функции `moves` ***(4,*** `true`***)***
в программе `TowersOfHanoi`

Деревья вызовов могут использоваться для анализа поведения любой модульной программы, но они особенно полезны для выявления закономерностей поведения рекурсивных программ. Например, дерево, соответствующее вызову `move()` в программе `TowersOfHanoi`, строится просто. Сначала нарисуйте узел дерева, помеченный значениями аргументов командной строки. Первый аргумент содержит количество дисков в перемещаемой стопке (и метку перемещаемого диска), второй — направление перемещения диска. Для наглядности мы обозначаем направление (значение `boolean`) стрелкой, направленной влево или вправо, потому что значение в программе интерпретируется именно так — как направление перемещения диска. Затем нарисуйте ниже два узла с количеством дисков, уменьшенным на 1, и противоположным направлением. Продолжайте делать это, пока на концах всех «ветвей» не окажутся узлы-«листья» со значением первого аргумента, равным 1. Эти узлы соответствуют вызовам `moves()`, не приводящим к дальнейшим рекурсивным вызовам.

Проанализируйте дерево вызовов, приведенное ранее в этом разделе, и сравните его с реальной трассировкой, приведенной на иллюстрации. Вы увидите, что дерево рекурсии — всего лишь компактное представление трассировки. В частности, читая метки узлов слева направо, вы получите ходы, выполняемые в процессе решения задачи.

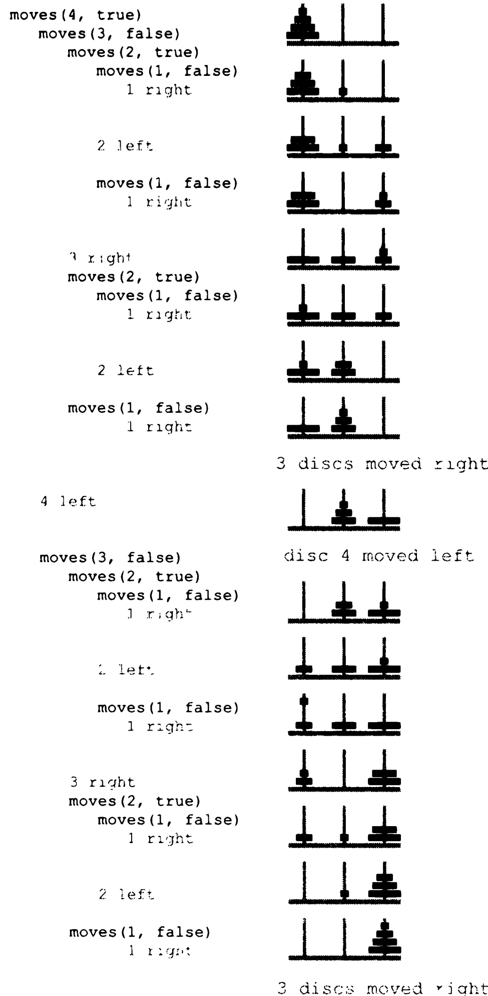
Кроме того, при анализе дерева можно заметить несколько закономерностей, включая следующие две:

- через один ход, начиная с первого, перемещается наименьший диск;
- этот диск всегда движется в одном направлении.

Эти наблюдения актуальны, потому что они определяют решение задачи, не требующее рекурсии (и даже компьютера): каждым нечетным ходом (включая первый

и последний) перемещается наименьший диск, а каждый нечетный ход является единственно допустимым на этот момент, в котором не задействован наименьший диск. Используя индукцию, можно доказать, что этот метод дает тот же результат, что и рекурсивная программа. Вероятно, монахи, начавшие перекладывать диски сотни лет назад, еще до изобретения компьютера, используют именно этот метод.

Деревья очень важны для понимания рекурсии, потому что дерево по своей природе является рекурсивным объектом. Как абстрактная математическая модель, дерево играет важную роль во многих областях, а в главе 4 будет рассмотрено применение деревьев как вычислительной модели структурирования данных для эффективной обработки.



Трассировка вызовов функций для move (4, true)

Экспоненциальное время выполнения

Одно из преимуществ рекурсии заключается в том, что для нее часто удается разрабатывать математические модели для описания поведения рекурсивных программ и изучения их свойств. Так, в задаче о ханойских башнях можно оценить время, оставшееся до конца света (если предположить, что легенда правдива). Почему это так важно? Во-первых, вы узнаете, что до конца света еще далеко, а во-вторых, получите информацию, которая позволит вам избежать написания программ, которые не могут завершиться в обозримом будущем.

Математическая модель задачи о ханойских башнях проста: если определить функцию $T(n)$ как число дисков, перемещаемых программой `TowersOfHanoi` для решения задачи с n дисками, то рекурсивная реализация алгоритма подразумевает, что функция $T(n)$ подчиняется следующему условию:

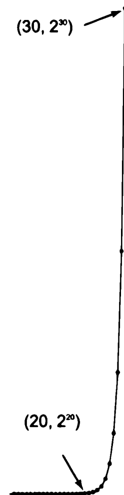
$$T(n) = 2T(n - 1) + 1 \text{ для } n > 1, \text{ при } T(1) = 1.$$

В дискретной математике подобные формулы называются *рекуррентными соотношениями*. Рекуррентные соотношения естественным образом встречаются при анализе рекурсивных программ. Часто они могут использоваться для определения нужной характеристики в замкнутой форме. Для $T(n)$ по исходным значениям $T(1) = 1$, $T(2) = 3$, $T(3) = 7$ и $T(4) = 15$ можно предположить, что $T(n) = 2^n - 1$. Рекуррентное соотношение позволяет доказать этот факт посредством математической индукции.

- Базовый случай: $T(1) = 2^n - 1 = 1$.
- Индукционный переход: для $T(n - 1) = 2^{n-1} - 1$, $T(n) = 2\ (2^{n-1} - 1) + 1 = 2^n - 1$.

Следовательно, по индукции $T(n) = 2^n - 1$ для всех $n > 0$. Минимально возможное количество перемещений также удовлетворяет тому же рекуррентному соотношению (см. упражнение 2.3.11).

Зная значение $T(n)$, можно оценить время, необходимое для выполнения всех перемещений. Если монахи перекладывают один диск в секунду, для решения задачи с 20 дисками им потребуется более недели, с 30 дисками — более 34 лет и с 40 дисками — более 348 веков (и то если они ни допустят ни единой ошибки). Для задачи с 64 дисками потребуется более 5,8 миллиарда веков. Впрочем, конец света состоится еще позднее, потому что монахи наверняка не пользуются программой 2.3.2 и не могут перекладывать диски с такой быстротой или так быстро решить, какой диск должен перекладываться следующим.



Экспоненциальный рост

Даже компьютеры не справляются с экспоненциальным ростом. Для компьютера, способного выполнять до мил-

лиарда операций в секунду, выполнение 2^{64} операций потребует много веков, а выполнение 2^{1000} операций не под силу ни одному компьютеру. Это очень принципиальный момент: применяя рекурсию, вы можете легко и быстро писать простые и короткие программы с экспоненциальной трудоемкостью выполнения, которые попросту не будут завершаться за сколько-нибудь реальное время при больших n. Новичкам этот принципиальный факт часто кажется невероятным, поэтому вам стоит ненадолго задуматься над ним. Чтобы убедиться в том, что дело обстоит именно так, исключите трассировку из программы `TowersOfHanoi` и выполните ее для возрастающих значений n, начиная с 20. Вы легко убедитесь, что при каждом увеличении n на 1 время выполнения удваивается, и вам быстро надоест дожидаться завершения. Если для некоторого значения n программа выполняется за час, то для $n + 5$ придется ожидать больше суток, для $n + 10$ — больше месяца, а для $n + 20$ — больше века (здесь уж никакого терпения не хватит). Вашему компьютеру может просто не хватить производительности для выполнения какой-нибудь написанной вами короткой программы на Java, внешне кажущейся совсем простой! *Остерегайтесь программ, время для выполнения которых может расти экспоненциально.*

Разработчика часто интересует оценка времени выполнения его программ. В разделе 4.1 мы обсудим применение только что описанного подхода для оценки времени выполнения других программ.

Коды Грея

Задача о ханойских башнях отнюдь не «игрушечная». Она тесно связана с базовыми алгоритмами для работы с числами и объектами. В качестве примера рассмотрим *коды Грея* — математическую абстракцию, находящую множество практических применений.

Драматург Сэмюэл Беккет (вероятно, наиболее известный по пьесе «В ожидании Годо») написал пьесу «Четверо», которая обладает следующим свойством: действие начинается с пустой сцены, а персонажи входят и выходят по одному, при этом каждое подмножество персонажей на сцене встречается ровно один раз. Как Беккет сгенерировал план постановки для этой пьесы?

код	*подмножество*	*перемещение*
0 0 0 0	empty	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

Представление кода Грея

Один из способов представления подмножества n разных объектов — использование строки из n бит. В задаче Беккета используется строка из 4 бит; биты нумеруются справа налево и соответствуют персонажам, а значение 1 обозначает присутствие персонажа на сцене. Например, строка `0 1 0 1` представляет сцену, где присут-

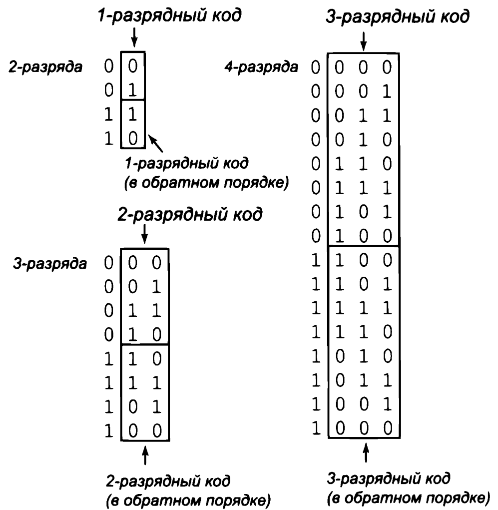
ствуют персонажи 3 и 1. Такое представление быстро доказывает основополагающий факт: у множества из n объектов существует ровно 2^n разных подмножества. В пьесе четыре персонажа, поэтому количество разных сцен равно $2^4 = 16$. Требуется сгенерировать план постановки.

n-разрядный код Грея представляет собой список из 2^n неповторяющихся двоичных чисел, в котором каждый элемент списка находится на расстоянии[1] ровно одного бита от своего предшественника. Коды Грея непосредственно применимы в задаче Беккета, потому что изменение значения бита с 0 на 1 соответствует включению персонажа в подмножество (выходу на сцену); изменение бита с 1 на 0 — исключению его из подмножества (уходу со сцены).

Как генерируются коды Грея? Здесь эффективно работает рекурсивный план, очень похожий на тот, который использовался для задачи о ханойских башнях. *n-разрядный рефлексивный код Грея* определяется рекурсивно следующим образом:

- $(n-1)$-разрядный код, с добавлением 0 перед каждым словом, за которым следует
- $(n-1)$-разрядный код *в обратном порядке*, с добавлением 1 перед каждым словом.

0-разрядный код определяется как пустой, так что 1-разрядный код состоит из 0 и 1. Из этого рекурсивного определения можно путем индукции убедиться в том, что n-разрядный рефлексивный код Грея обладает требуемым свойством: соседние кодовые слова отличаются значением только одного бита. Это утверждение истинно по индукционной гипотезе, кроме разве что последнего кодового слова в первой половине и первого кодового слова во второй половине: эта пара отличается только в позиции первого бита.



2-, 3- и 4-разрядные коды Грея

После некоторых размышлений рекурсивное определение приводит нас к реализации программы `Beckett` (листинг 2.3.3), генерирующей план постановки пьесы Беккета. Эта программа очень похожа на `TowersOfHanoi`. В самом деле, если не считать терминологии, отличаются только значения вторых аргументов в рекурсивных вызовах!

[1] Так называемое *кодовое расстояние*, или *расстояние Хемминга*, — количество разрядов, значения которых для двух соседних кодовых слов различны. Кодовое расстояние между соседними словами кода Грея всегда равно 1. — *Примеч. науч. ред.*

Листинг 2.3.3. Код Грея

n	количество актеров
enter	инструкция

```
public class Beckett
{
   public static void moves(int n, boolean enter)
   {
      if (n == 0) return;
      moves(n-1, true);
      if (enter) StdOut.println("enter " + n);
      else       StdOut.println("exit  " + n);
      moves(n-1, false);
   }

   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      moves(n, true);
   }
}
```

Эта рекурсивная программа генерирует инструкции для актеров в постановке Беккета (позиции битов, изменяющихся в рефлексивном коде Грея). Позиция изменяющегося бита определяется точно так же, как и в функции ruler, а актеры, разумеется, выходят на сцену и уходят с нее поочередно.

```
% java Beckett 1
enter 1
% java Beckett 2
enter 1
enter 2
exit  1
% java Beckett 3
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
```

```
% java Beckett 4
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
enter 4
enter 1
enter 2
exit  1
exit  3
enter 1
exit  2
exit  1
```

Как и в случае с направлениями в `TowersOfHanoi`, инструкции выхода актеров на сцену и их ухода в программе `Beckett` избыточны, ведь инструкция `exit` возможна только в том случае, если актер находится на сцене, а инструкция `enter` — только если актер отсутствует. В самом деле, и `Beckett`, и `TowersOfHanoi` напрямую связаны с функцией формирования делений линейки, которая рассматривалась в одной из первых программ (листинг 1.2.1). Не считая операций вывода, обе они имеют в основе простую рекурсивную функцию, подобную той, которая позволяла программе `Ruler` получить результаты для любого значения, переданного как аргумент командной строки.

Коды Грея применяются во многих областях, от аналого-цифровых преобразователей до планирования экспериментов. Они использовались в кодово-импульсной связи, при минимизации логических схем, в гиперкубических архитектурах и даже для упорядочения книг на библиотечных полках.

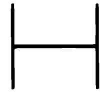
Рекурсивная графика

Простые рекурсивные алгоритмы способны строить весьма замысловатые изображения. Рекурсивные изображения не только связаны с различными практическими областями, но и предоставляют привлекательную основу для лучшего понимания свойств рекурсивных функций, потому что мы можем наблюдать за процессом постепенного формирования рекурсивного изображения.

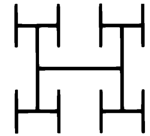
Начнем с программы `Htree` (листинг 2.3.4), которая для заданного аргумента командной строки n строит *H-дерево порядка n*, определяемое следующим образом: в базовом случае для $n = 0$ не рисуется ничего. На каждом следующем шаге свертки в единичном квадрате строятся:

- три отрезка в форме буквы H;
- четыре H-дерева порядка $n - 1$, центром каждого из которых является один из концов буквы H, с дополнительным условием, что H-деревья порядка $n - 1$ имеют половинный размер.

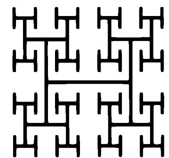
порядок 1



порядок 2



порядок 3



H-деревья

H-деревья находят много практических применений. Например, вообразите компанию, предоставляющую услуги кабельного телевидения, которая должна проложить кабель по всем домам на обслуживаемой ею территории. Одна из разумных стратегий заключается в том, чтобы использовать H-дерево для передачи сигнала по некоторому числу центров, распределенных по территории, а затем провести кабель от каждого дома к ближайшему центру. С аналогичной задачей сталкиваются разработчики электронных компонентов, которым нужно «развести» питание или сигнал по микросхеме.

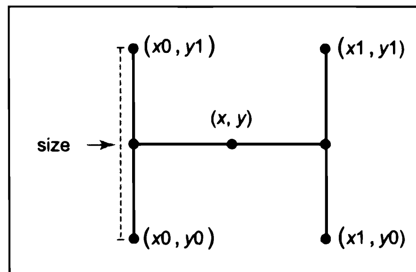
И хотя каждое изображение строится в окне фиксированного размера, H-деревья безусловно демонстрируют экспоненциальный рост. H-дерево порядка n соединяет 4^n центров, поэтому при $n = 10$ вам придется рисовать более миллиона линий, а при $n = 15$ — более миллиарда. При $n = 30$ вы заведомо не дождетесь завершения работы программы.

Если вы запустите программу `Htree` на своем компьютере с изображением, которое строится за минуту или около того, даже простое наблюдение за процессом рисования даст хорошее представление о природе рекурсивных программ, потому что вы будете видеть, в каком порядке появляются фигуры H и как из них образуются H-деревья. Еще более примечательно то, что при любом порядке рекурсивных

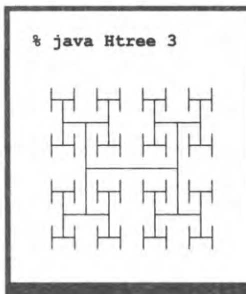
Листинг 2.3.4. Рекурсивная графика

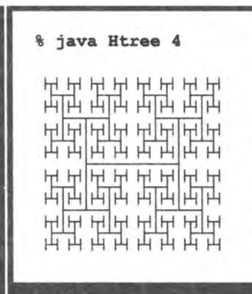
```
public class Htree
{
   public static void draw(int n, double size, double x, double y)
   {  // Построение H-дерева порядка n
      // с заданным размером и с центром x, y.
      if (n == 0) return;
      double x0 = x - size/2, x1 = x + size/2;
      double y0 = y - size/2, y1 = y + size/2;
      StdDraw.line(x0,  y, x1,  y);
      StdDraw.line(x0, y0, x0, y1);
      StdDraw.line(x1, y0, x1, y1);
      draw(n-1, size/2, x0, y0);
      draw(n-1, size/2, x0, y1);
      draw(n-1, size/2, x1, y0);
      draw(n-1, size/2, x1, y1);
   }

   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      draw(n, 0.5, 0.5, 0.5);
   }
}
```

`n`	глубина рекурсии (порядок H-дерева)
`size`	длина отрезка
`x, y`	центр



Функция draw() чертит три отрезка, каждый из которых имеет длину size, в форме буквы H с центром в точке (x, y). Затем она рекурсивно вызывает себя для каждой из четырех конечных точек, с уменьшением аргумента size вдвое и с использованием целочисленного аргумента n для управления глубиной рекурсии.





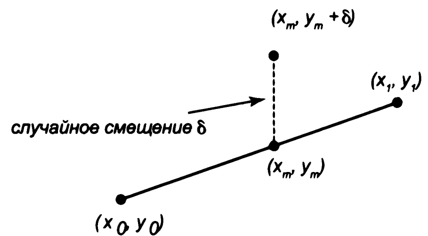
вызовов `draw()` и вызовов `StdDraw.line()` всегда строится один и тот же рисунок: проследите за тем, как порядок этих вызовов влияет на порядок появления линий в формирующемся изображении (см. упражнение 2.3.14).

Броуновский мост

Н-дерево является простым примером *фрактала* — геометрической фигуры, которую можно разделить на части, каждая из которых (приблизительно) представляет собой уменьшенную копию оригинала. Фракталы легко строятся рекурсивными программами, хотя ученые, математики и программисты изучают их с разных позиций. Мы уже несколько раз сталкивались с фракталами в этой книге — например, в программе `IFS` (листинг 2.2.3).

Изучение фракталов играет важную роль в художественном творчестве, экономическом анализе и научных исследованиях. Художники и ученые используют фракталы для построения компактных моделей сложных фигур, которые встречаются в природе и плохо описываются средствами традиционной геометрии: облаков, растений, гор, рек, человеческой кожи и многих других. Экономисты используют фракталы для моделирования графиков функций экономических характеристик.

Броуновские фракталы — математическая модель построения реалистичных фрактальных моделей для многих естественных форм. В частности, она используется в финансовой инженерии и при изучении многих природных явлений, включая морские приливы и работу нервных окончаний. Вычисление точных значений фракталов на основании этой модели может быть достаточно сложной задачей, но приближенные оценки относительно легко вычисляются с помощью рекурсивных программ.



Построение броуновского моста

Программа `Brownian` (листинг 2.3.5) строит график функции, который является приближением броуновского фрактала — так называемого *броуновского моста*. Этот график можно представить как случайное блуждание, соединяющее две точки (x_0, y_0) и (x_1, y_1) и зависящее от нескольких параметров. Реализация основана на *методе смещения средней точки*, который представляет собой рекурсивный процесс построения графика на интервале [x_0, y_0] по оси x. Базовый случай (при котором длина интервала меньше заданного допуска) сводится к вычерчиванию отрезка, соединяющего две конечные точки. На шаге свертки интервал делится на две половины. Говоря более формально, выполняются следующие действия.

- Вычисляется средняя точка (x_m, y_m) интервала.
- К y-координате y_m средней точки прибавляется случайное значение δ из гауссова распределения с математическим ожиданием 0 и заданной дисперсией.

Листинг 2.3.5. Броуновский мост

```
public class Brownian
{
   public static void curve(double x0, double y0,
                            double x1, double y1,
                            double var, double s)
   {
      if (x1 - x0 < 0.01)
      {
         StdDraw.line(x0, y0, x1, y1);
         return;
      }
      double xm = (x0 + x1) / 2;
      double ym = (y0 + y1) / 2;
      double delta = StdRandom.gaussian(0, Math.sqrt(var));
      curve(x0, y0, xm, ym+delta, var/s, s);
      curve(xm, ym+delta, x1, y1, var/s, s);
   }
   public static void main(String[] args)
   {
      double hurst = Double.parseDouble(args[0]);
      double s = Math.pow(2, 2*hurst);
      curve(0, 0.5, 1.0, 0.5, 0.01, s);
   }
}
```

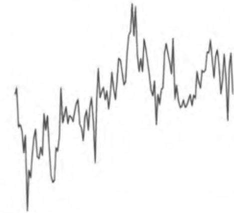
`x0, y0`	левый конец
`x0, y0`	правый конец
`xm, ym`	середина
`delta`	смещение
`var`	дисперсия
`hurst`	экспонента Херста

Вводя небольшое случайное гауссово смещение в программу, которая строит прямую линию, мы получаем фрактальные кривые. Аргумент командной строки hurst, называемый экспонентой Херста, управляет плавностью кривых.

```
% java Brownian 1
```

```
% java Brownian 0.5
```

```
% java Brownian 0.05
```



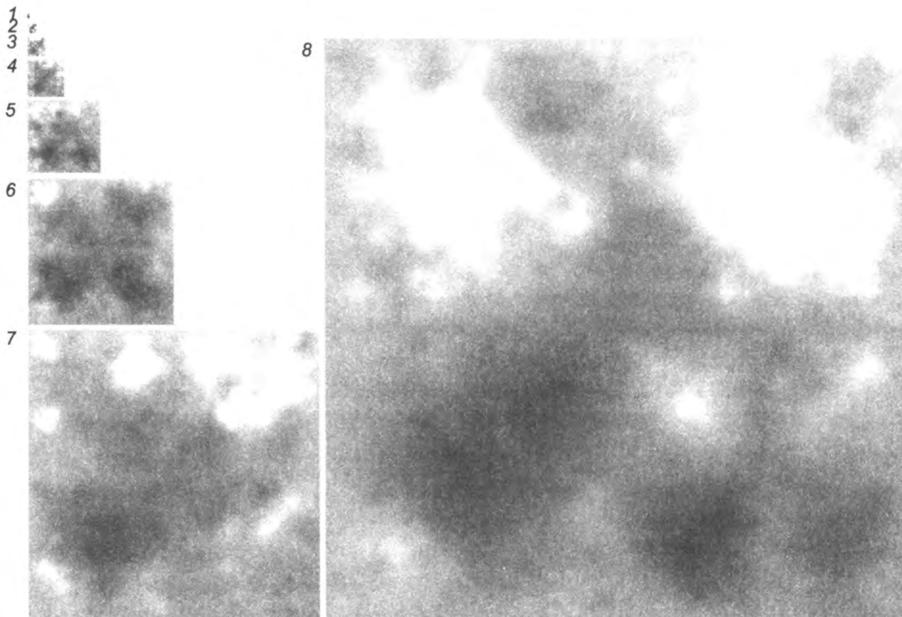
- Процедура повторяется для подынтервалов с уменьшением дисперсии на заданный коэффициент масштабирования *s*.

Форма кривой зависит от двух параметров: *нестабильность* (исходное значение дисперсии) управляет расстоянием, на которое график функции отклоняется от прямой линии, соединяющей точки, а *экспонента Херста* управляет плавностью кривой. Обозначим экспоненту Херста H и будем делить дисперсию на 2^{2H} на

каждом уровне рекурсии. Когда значение H равно 1/2 (сокращение вдвое на каждом уровне), кривая представляет собой броуновский мост — обобщение задачи о разорении игрока на непрерывный случай (см. программу 1.3.8). Если $0 < H < 1/2$, отклонения увеличиваются, а кривая получается более изломанной. Наконец, при $2 > H > 1/2$ смещения уменьшаются, а кривая получается более гладкой. Значение $2 - H$ называется *фрактальной размерностью* кривой.

Нестабильность и исходные конечные точки интервала имеют отношение к масштабу и позиционированию. Тестовый метод-клиент `main()` в `Brownian` дает возможность поэкспериментировать с экспонентой Херста. Со значениями больше 1/2 вы получите график, напоминающий силуэт гор на горизонте; со значениями меньше 1/2 получается что-то похожее на график колебаний биржевых котировок.

Расширение метода смещения средней точки для двух измерений позволяет строить фракталы, называемые *плазменными облаками*. Для рисования прямоугольных плазменных облаков используется рекурсивная схема, в которой базовым случаем является прямоугольник заданного цвета, а сверткой — рисование плазменного облака меньшего размера в каждой из четырех четвертей цветами, отличающимися от среднего случайным гауссовым смещением. Используя такие же параметры управления нестабильностью и плавностью, как в `Brownian`, можно строить на удивление реалистичные изображения облаков. Этот же код может использоваться для синтеза рельефа (значение цвета интерпретируется как высота). Разновидности этой схемы широко применяются в индустрии развлечений для построения фонов в фильмах и компьютерных играх.



Плазменные облака

Проблемы с рекурсией

Вероятно, вы уже удостоверились, что рекурсия позволяет писать компактные и элегантные программы. Но когда вы начнете строить собственные рекурсивные программы, следует помнить о некоторых часто встречающихся проблемах. Одна из таких проблем уже была рассмотрена относительно подробно — это время выполнения программ, которое может расти экспоненциально. После того как проблемы будут выявлены, они обычно довольно легко решаются, но вы должны научиться проявлять предусмотрительность и избегать их при написании рекурсивных программ.

Отсутствие базового случая

Следующая рекурсивная функция вроде бы должна вычислять гармонические числа, но в ней отсутствует базовый случай:

```
public static double harmonic(int n)
{
   return harmonic(n-1) + 1.0/n;
}
```

Если запустить программу, вызывающую эту функцию, она будет постоянно вызывать себя без возврата управления, и ваша программа никогда не завершится. Вероятно, вы уже сталкивались с бесконечными циклами: вы запускаете свою программу, и ничего не происходит (или на экран идет бесконечный поток вывода). С бесконечной рекурсией дело обстоит иначе, потому что система сохраняет информацию о каждом рекурсивном вызове (для этого используется механизм, описанный в разделе 4.3; в его основе лежит структура данных, называемая *стеком*), и в конечном итоге у нее не хватит памяти для сохранения при очередном вызове. Рано или поздно Java выдаст ошибку `StackOverflowError` во время выполнения. Работая над созданием рекурсивной программы, всегда старайтесь убедиться в том, что она приводит к желаемой цели; для этого можно воспользоваться неформальными рассуждениями, основанными на математической индукции. Иногда в таких случаях обнаруживается отсутствие базового случая.

Отсутствие гарантий сходимости

Другая распространенная ошибка — включение в рекурсивную функцию рекурсивного вызова для решения подзадачи, которая не проще исходной. Например, следующий метод переходит в бесконечный цикл рекурсии для любого значения своего аргумента (кроме 1), потому что последовательность значений аргумента не сходится к базовому случаю:

```
public static double harmonic(int n)
{
   if (n == 1) return 1.0;
   return harmonic(n) + 1.0/n;
}
```

Такие ошибки обнаруживаются достаточно легко, но более тонкие их разновидности выявить намного сложнее. Некоторые примеры такого рода приводятся в упражнениях в конце раздела.

Перерасход памяти

Если количество рекурсивных вызовов функции слишком велико, то и затраты памяти для обеспечения всех этих вызовов могут оказаться недопустимо большими, что приведет к ошибке `StackOverflowError`. Чтобы получить представление о том, сколько памяти для этого требуется, проведите небольшую серию экспериментов с рекурсивной функцией для вычислений гармонических чисел для возрастающих значений n:

```
public static double harmonic(int n)
{
   if (n == 1) return 1.0;
   return harmonic(n-1) + 1.0/n;
}
```

Значение, при котором произойдет ошибка `StackOverflowError`, дает некоторое представление о том, сколько памяти Java расходует для реализации рекурсии. С другой стороны, программа для вычисления H_n (листинг 1.3.5) с большими n выполняется с минимальными затратами памяти[1].

Лишние повторные вычисления

Искушение написать простую рекурсивную функцию для решения задачи велико, но вы всегда должны помнить о том, что функция может потребовать экспоненциального времени выполнения (хотя это и не обязательно) из-за лишних повторных вычислений. Такой эффект возможен даже с самыми простыми рекурсивными функциями, и вы, безусловно, должны научиться избегать его. Например, последовательность *чисел Фибоначчи*

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

определяется рекуррентным отношением $F_n = F_{n-1} + F_{n-2}$ для $n \geq 2$ с $F_0 = 0$ и $F_1 = 1$. Последовательность Фибоначчи обладает многими интересными свойствами и встречается во многих областях. Неопытный программист может написать рекурсивную функцию для вычисления чисел последовательности Фибоначчи так:

```
// Внимание: крайне неэффективная реализация!
public static long fibonacci(int n)
{
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

[1] Это лишний раз демонстрирует критичность в некоторых случаях экономного использования памяти за счет правильного выбора структур данных и подходов к манипулированию ими. К сожалению, перерасход памяти по описанному сценарию — характерное явление для объектно-ориентированных языков, в том числе и для Java. — *Примеч. науч. ред.*

Однако такая реализация крайне неэффективна! Новички нередко отказываются признавать этот факт и выполняют такой код, полагая, что компьютер достаточно быстр для получения ответа. Попробуйте сами; посмотрите, хватит ли производительности вашего компьютера для вычисления `fibonacci(50)`. А чтобы понять, почему эта попытка обречена на провал, посмотрите, что функция делает для вычисления `fibonacci(8)=21`. Сначала она вычисляет `fibonacci(7)=13` и `fibonacci(6)=8`. Чтобы вычислить `fibonacci(7)`, она по рекурсии *снова* вычисляет `fibonacci(6)=8` и `fibonacci(5)=5`. Ситуация ухудшается очень быстро, потому что функция оба раза вычисляет `fibonacci(6)` заново, игнорируя уже вычисленное значение `fibonacci(5)` и т. д. Собственно, при вычислении `fibonacci(n)` эта программа вычислит `fibonacci(1)` ровно F_n раз (см. упражнение 2.3.12). Потери на повторные вычисления растут по экспоненте. Например, вызов `fibonacci(200)` совершает $F_{200} > 10^{43}$ рекурсивных вызовов `fibonacci(1)`! Ни один компьютер не справится с таким объемом вычислений. *Остерегайтесь программ, время на выполнение которых может расти экспоненциально*. К этой категории относятся многие вычисления, находящие естественное выражение в виде рекурсивных функций. Не делайте так, этот путь ведет в тупик.

```
fibonacci(8)
   fibonacci(7)
      fibonacci(6)
         fibonacci(5)
            fibonacci(4)
               fibonacci(3)
                  fibonacci(2)
                     fibonacci(1)
                        return 1
                     fibonacci(0)
                        return 0
                     return 1
                  fibonacci(1)
                     return 1
                  return 2
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               return 3
            fibonacci(3)
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               fibonacci(1)
                  return 1
               return 2
            return 5
         fibonacci(4)
            fibonacci(3)
               fibonacci(2)
                  .
                  .
                  .
```

Неправильный подход к вычислению чисел Фибоначчи

Мы рассмотрим системный подход, называемый *динамическим программированием*, — элегантный способ предотвращения подобных проблем. Идея заключается в том, чтобы обойтись без лишних вычислений, присутствующих в некоторых рекурсивных функциях; для этого ранее вычисленные значения сохраняются для использования на следующих шагах — вместе того чтобы вычислять их снова и снова.

Динамическое программирование

Общий подход к реализации рекурсивных программ, называемый *динамическим программированием*, предоставляет эффективные и элегантные решения для широкого круга задач. Общая идея заключается в том, чтобы рекурсивно разделить

большую задачу на несколько меньших подзадач, сохранить ответы на каждую из этих подзадач и на завершающем этапе использовать сохраненные результаты подзадач для решения всей исходной задачи. Каждая подзадача решается только один раз (а не снова и снова), тем самым предотвращается возможность экспоненциального роста времени выполнения.

Например, если исходная задача должна вычислять n-е число Фибоначчи, будет естественно определить $n + 1$ подзадач, где подзадача i вычисляет i-е число Фибоначчи для каждого $0 \leq i \leq n$. Подзадача i легко решается, если вам уже известны решения меньших подзадач, а именно подзадач $i - 1$ и $i - 2$. Более того, решение исходной задачи представляет собой решение одной из подзадач — подзадачи n.

Нисходящее динамическое программирование

При *нисходящем динамическом программировании* результат каждой решенной подзадачи сохраняется (*кэшируется*), и если вам в следующий раз придется решать ту же подзадачу, вы можете использовать кэшированное значение вместо того, чтобы решать ее «с нуля». В нашем примере для хранения уже вычисленных чисел Фибоначчи используется массив `f[]`. В языке Java для этой цели используется *статическая переменная*, также называемая *переменной класса* или *глобальной переменной*; такие переменные объявляются за пределами какого-либо метода, что позволяет сохранять информацию между вызовами методов.

```
public class TopDownFibonacci
{
   private static long[] f = new long[92];

   public static long fibonacci(int n)
   {
      if (n == 0) return 0;
      if (n == 1) return 1;
      if (f[n] > 0) return f[n];
      f[n] = fibonacci(n-1) + fibonacci(n-2);
      return f[n];
   }
}
```


Применение нисходящего динамического программирования для вычисления чисел Фибоначчи

Нисходящее динамическое программирование позволяет избежать повторения уже выполненной работы за счет сохранения результатов вызовов. Также оно называется *мемоизацией*.

Восходящее динамическое программирование

При *восходящем динамическом программировании* вычисляются решения всех подзадач, начиная с самых «простых», и постепенно наращиваются решения к более сложным подзадачам. Чтобы воспользоваться восходящим динамическим про-

граммированием, следует упорядочить подзадачи так, чтобы каждая последующая подзадача решалась объединением решений предыдущих задач (которые уже были решены к этому моменту). В нашем примере с числами Фибоначчи это делается просто: подзадачи решаются в порядке 0, 1, 2 и т. д. К тому моменту, когда потребуется решить подзадачу i, все меньшие подзадачи уже будут решены, включая подзадачи $i - 1$ и $i - 2$.

```
public static long fibonacci(int n)
{
   int[] f = new int[n+1];
   f[0] = 0;
   f[1] = 1;
   for (int i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

Если порядок подзадач очевиден и все решения помещаются в памяти, решения, основанные на восходящем динамическом программировании, работают очень эффективно. Мы рассмотрим более сложный пример применения динамического программирования, когда порядок решения подзадач не столь очевиден (пока вы не увидите его). В отличие от чисел Фибоначчи, найти для него нерекурсивное решение по методу восходящего динамического программирования будет значительно сложнее.

Задача о поиске наибольшей общей подстроки

Рассмотрим фундаментальную задачу обработки строк, которая часто встречается в вычислительной биологии и других областях: для двух заданных строк x и y требуется определить степень их сходства. Примеров много — проверка на сходство двух цепочек ДНК, двух английских слов, двух файлов с кодом Java на повторение кода и т. д. Одной из метрик сходства является *длина наибольшей общей подстроки* (LCS, Longest Common Subsequence). Если удалить часть символов из x и y, после чего две оставшиеся строки совпадут, такая оставшаяся строка называется *общей подстрокой*. Задача о поиске LCS заключается в поиске общей подстроки двух строк максимальной длины. Например, наибольшей общей подстрокой `GGCACCACG` и `ACGGCGG ATACG` является строка `GGCAACG` длины 7.

Алгоритмы вычисления LCS используются в программах сравнения данных — например, в команде `diff` в Unix, при помощи которой программисты десятилетиями проверяли степень сходства текстовых файлов. Похожие алгоритмы играют важную роль в научных вычислениях — например, в алгоритме Смита — Уотермана в вычислительной биологии и в алгоритме Витерби в теории цифровой связи.

Рекуррентность LCS

Теперь мы опишем рекурсивную формулу, которая позволяет найти LCS двух заданных строк `s` и `t`. Пусть `m` и `n` — длины `s` и `t` соответственно. Запись `s[i..m)` используется для обозначения *суффикса* `s`, начинающегося с индекса `i`,

и t[j..n) — для суффикса t, начинающегося с индекса j. С одной стороны, если s и t начинаются с одного символа, то LCS x и y содержит этот первый символ. Таким образом, задача сводится к нахождению LCS суффиксов s[1..m) и t[1..n). С другой стороны, если s и t начинаются с разных символов, то оба символа не могут быть частью одной подстроки, поэтому одну из строк можно спокойно отбросить. В любом случае задача сводится к нахождению LCS двух строк: либо s[0..m) и t[1..n), либо s[1..m) и t[0..n) (той из них, которая будет строго короче). В общем случае, если opt[i][j] обозначает длину LCS суффиксов s[i..m) и t[j..m), следующее рекуррентное отношение выражает opt[i][j] в длинах LCS более коротких суффиксов.

```
              0                               if i = m or j = n
opt[i][j] =   opt[i+1, j+1] + 1               if s[i] = t[j]
              max(opt[i, j+1], opt[i+1, j])   в остальных случаях
```

Решение методом динамического программирования

Программа LongestCommonSubsequence (листинг 2.3.6) сначала применяет метод восходящего динамического программирования для разрешения рекуррентности. Мы создаем двумерный массив opt[i][j] для хранения длин LCS суффиксов s[i..m) и t[j..n). Изначально нижняя строка (значения для i=m) и правый столбец (значения для j=n) заполнены 0. Из рекуррентного отношения четко следует порядок остальных вычислений: все начинается с opt[m][n]. Затем при уменьшении i и/или j мы знаем, что все необходимые значения для определения opt[i][j] были вычислены, потому что для элемента opt[][] нужны элементы с большими значениями i и/или j. Метод lcs() в листинге 2.3.6 вычисляет элементы opt[][], заполняя значения по строкам снизу вверх (от $i = m - 1$ до 0) и справа налево в каждой строке (от $j = n - 1$ до 0). Также возможен альтернативный порядок с заполнением значений по столбцам справа налево и по строкам снизу вверх. На следующем рисунке (с. 292) для каждого элемента стрелками обозначены элементы, используемые для его вычисления.

И наконец, последнее усилие — получение самой подстроки, а не только ее длины. Для этого нужно отследить шаги алгоритма динамического программирования в обратном направлении, чтобы восстановить найденный путь (выделен серым цветом на диаграмме) от opt[0][0] до opt[m][n]. Чтобы определить вариант, приводящий к opt[i][j], следует рассмотреть три возможности.

- Символ s[i] равен t[j]. В этом случае должно выполняться условие opt[i][j] = = opt[i+1][j+1] + 1, а следующим символом в LCS является s[i] (или t[j]), поэтому мы включаем символ s[i] (или t[j]) в LCS и продолжаем двигаться от opt[i+1][j+1].
- LCS не содержит s[i]. В этом случае opt[i][j] = opt[i+1][j], и мы продолжаем двигаться от opt[i+1][j].
- LCS не содержит t[j]. В этом случае opt[i][j] = opt[i][j+1], и мы продолжаем двигаться от opt[i][j+1].

Листинг 2.3.6. Наибольшая общая подстрока

```
public class LongestCommonSubsequence
{
   public static String lcs(String s, String t)
   {  // Вычисление длин LCS для всех подзадач
      int m = s.length(), n = t.length();
      int[][] opt = new int[m+1][n+1];
      for (int i = m-1; i >= 0; i--)
         for (int j = n-1; j >= 0; j--)
         if (s.charAt(i) == t.charAt(j))
            opt[i][j] = opt[i+1][j+1] + 1;
         else
            opt[i][j] = Math.max(opt[i+1][j], opt[i][j+1]);
      // Формирование итоговой LCS
      String lcs = "";
      int i = 0, j = 0;
      while(i < m && j < n)
      {
         if (s.charAt(i) == t.charAt(j))
         {
            lcs += s.charAt(i);
            i++;
            j++;
         }
         else if (opt[i+1][j] >= opt[i][j+1]) i++;
         else                                 j++;
      }
      return lcs;
   }

   public static void main(String[] args)
   {  StdOut.println(lcs(args[0], args[1]));  }
}
```

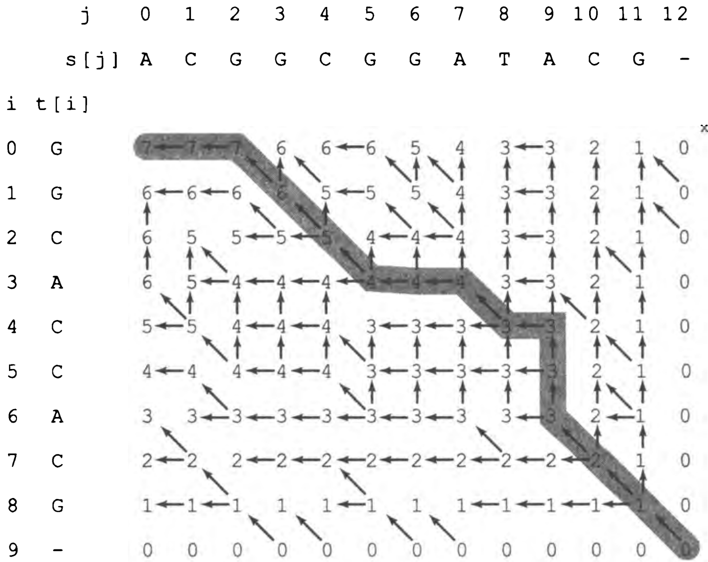
s, t	две строки
m, n	длины двух строк
opt[i][j]	длина LCS для x[i..m) и y[j..n)
lcs	наибольшая общая подстрока

Функция lcs() вычисляет и возвращает LCS двух строк x и y, используя восходящее динамическое программирование. Вызов метода s.charAt(i) возвращает символ i строки s.

```
% java LongestCommonSubsequence GGCACCACG ACGGCGGATACG
GGCAACG
```

Движение начинается с `opt[0][0]` и продолжается, пока не будет достигнут элемент `opt[m][n]`. На каждом шаге увеличивается `i` или `j` (или обе переменные сразу), поэтому процесс будет гарантированно завершен после максимум `m+n` итераций цикла `while`.

Динамическое программирование — одна из фундаментальных парадигм проектирования алгоритмов, тесно связанная с рекурсией. Если вы позднее займетесь изучением алгоритмов или технологическими исследованиями, вы наверняка больше узнаете о них. Концепция рекурсии исключительно важна для вычислений, и желание избежать повторного вычисления значений, которые уже были



Наибольшая общая подстрока GGCACCACG и ACGGCGGATACG

вычислены, выглядит вполне естественно. Впрочем, не все задачи сразу удается преобразовать к рекурсивной форме и не все рекурсивные формы подразумевают порядок, легко предотвращающий повторные вычисления. Счастливое совпадение одного с другим поначалу выглядит как маленькое чудо, как в только что рассмотренной задаче поиска LCS.

Выводы

Программисты, не использующие рекурсию, упускают две важные возможности. Во-первых, рекурсия позволяет создавать компактные решения для сложных задач. Во-вторых, в рекурсивном решении часто заложено обоснование правильности работы программы. На заре вычислительных систем затраты ресурсов, связанные с рекурсией, в некоторых системах оказывались недопустимо высокими, и многие разработчики обходили рекурсию стороной. В современных системах, и в Java в частности, рекурсия часто оказывается предпочтительным решением.

Рекурсивные функции в полной мере демонстрируют мощь тщательно сформулированной абстракции. Хотя концепция функции, способной вызывать саму себя, на первый взгляд может показаться абсурдной, рассмотренные нами примеры доказывают, что владение рекурсией крайне важно для понимания и использования вычислительных систем, а также для понимания роли вычислительных моделей при изучении природных явлений.

Рекурсия также демонстрирует, что правильность работы программы может быть объективно проверена. Между рекурсией и математической индукцией существует естественная связь. В повседневном программировании мы стремимся к обеспечению корректности кода, чтобы сберечь время и усилия, которые пришлось бы потратить на поиск ошибок. В современных приложениях с их особым вниманием к безопасности и конфиденциальности корректность работы *абсолютно необходима*. Если программист сам не может убедиться в том, что приложение работает так, как было задумано, то как в этом может быть уверен пользователь, желающий сохранить свои персональные данные в безопасности?

Рекурсия — последний аспект модели программирования, на основе которой была сформирована большая часть вычислительной инфраструктуры в тот период, когда компьютеры стали выходить на передний план в повседневной жизни в конце XX века. Программы, которые строились на основе библиотек функций с командами, работавшими с примитивными типами данных, ветвлениями, циклами и вызовами функций (в том числе и рекурсивными), способны решать важные задачи самых разных типов. В следующем разделе мы подчеркнем это обстоятельство и проанализируем эти концепции в контексте более крупного приложения. В главах 3 и 4 будет показано развитие этих базовых концепций, ориентированное на более современный стиль программирования, который сейчас занимает ведущее положение в мире.

Вопросы и ответы

В. Существуют ли ситуации, в которых перебор в цикле остается единственным способом решения задачи?

О. Нет, любой цикл можно заменить эквивалентной рекурсивной функцией, хотя для рекурсивной версии может потребоваться больше памяти.

В. Существуют ли ситуации, в которых задача может быть решена только с использованием рекурсии?

О. Нет, любую рекурсивную функцию можно заменить эквивалентным циклом. В разделе 4.3 будет показано, как компилятор генерирует код вызова функции с использованием специальной структуры данных, называемой *стеком*.

В. Что лучше выбрать — рекурсию или циклы?

О. То, что приведет к более простому, более понятному или эффективному коду.

В. Я понимаю, что в рекурсивном решении нужно беспокоиться о лишних затратах памяти и повторных вычислениях. Что-нибудь еще?

О. Будьте исключительно осторожны при создании массивов в рекурсивном коде. Затраты памяти возрастают очень быстро — как и время, необходимое для управления памятью.

Упражнения

2.3.1. Что произойдет, если вызвать `factorial()` с отрицательным значением `n`? С большим значением — скажем, 35?

2.3.2. Напишите рекурсивную функцию, которая получает аргумент — целое число n и возвращает $\ln (n!)$.

2.3.3. Приведите последовательность целых чисел, которая будет выведена вызовом `ex233(6)`:

```
public static void ex233(int n)
{
   if (n <= 0) return;
   StdOut.println(n);
   ex233(n-2);
   ex233(n-3);
   StdOut.println(n);
}
```

2.3.4. Приведите результат `ex234(6)`:

```
public static String ex234(int n)
{
   if (n <= 0) return "";
   return ex234(n-3) + n + ex234(n-2) + n;
}
```

2.3.5. Объясните, где допущена ошибка в следующей рекурсивной функции:

```
public static String ex235(int n)
{
   String s = ex235(n-3) + n + ex235(n-2) + n;
   if (n <= 0) return "";
   return s;
}
```

Ответ. Базовый случай никогда не будет достигнут, потому что он анализируется после шага свертки. Вызов `ex235(3)` приведет к вызовам `ex235(0)`, `ex235(-3)`, `ex235(-6)` и так далее вплоть до ошибки `StackOverflowError`.

2.3.6. Для заданных четырех положительных целых чисел `a`, `b`, `c` и `d` объясните, какое значение будет вычислено при вызове `gcd(gcd(a, b), gcd(c, d))`.

2.3.7. Объясните в контексте целых чисел и делителей эффект следующей функции, напоминающей функцию Евклида:

```
public static boolean gcdlike(int p, int q)
{
   if (q == 0) return (p == 1);
   return gcdlike(q, p % q);
}
```

2.3.8. Рассмотрим следующую рекурсивную функцию:

```
public static int mystery(int a, int b)
{
   if (b == 0)     return 0;
   if (b % 2 == 0) return mystery(a+a, b/2);
   return mystery(a+a, b/2) + a;
}
```

Чему равны значения `mystery(2, 25)` и `mystery(3, 11)`? Для заданных положительных целых `a` и `b` опишите, какое значение вернет `mystery(a, b)`. Затем ответьте на тот же вопрос, но замените `+` на `*`, а `return 0` — на `return 1`.

2.3.9. Напишите рекурсивную версию программы `Ruler` для вывода делений на линейке (см. листинг 1.2.1) с использованием `StdDraw`.

2.3.10. Реализуйте следующие рекуррентные соотношения (во всех случаях `T(1) = 1`). Предполагается, что `n` является степенью 2:

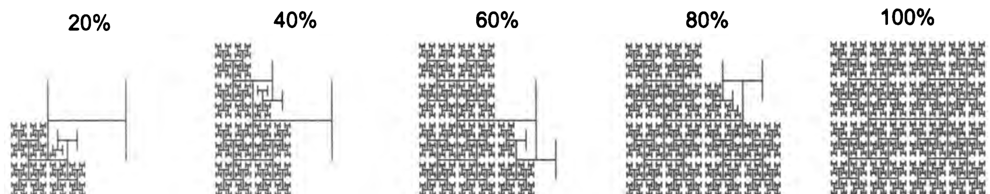
- $T(n) = T(n/2) + 1$;
- $T(n) = 2T(n/2) + 1$;
- $T(n) = 2T(n/2) + n$;
- $T(n) = 4T(n/2) + 3$.

2.3.11. Докажите методом индукции, что минимально возможное количество перемещений для решения задачи о ханойских башнях подчиняется тому же рекуррентному отношению, что и количество перемещений, использованное в нашем рекурсивном решении.

2.3.12. Докажите посредством индукции, что рекурсивная программа, приведенная в тексте, совершает ровно F_n рекурсивных вызовов `fibonacci(1)` при вычислении `fibonacci(n)`.

2.3.13. Докажите, что второй аргумент `gcd()` уменьшается по меньшей мере в 2 раза при каждом втором рекурсивном вызове, после чего докажите, что `gcd(p, q)` использует не более $2 \log_2 n + 1$ рекурсивных вызовов, где n — большее из `p` и `q`.

2.3.14. Измените программу `Htree` (листинг 2.3.4), чтобы отрисовка H-дерева происходила в режиме анимации. Затем измените порядок рекурсивных вызовов (и базовый случай), просмотрите полученную анимацию и объясните каждый результат.



Упражнения повышенной сложности

2.3.15. *Двоичное представление.* Напишите программу, которая получает положительное целое число n как аргумент командной строки и выводит его двоичное представление. Вспомните, что в листинге 1.3.7 использовался метод вычитания степеней 2. Теперь используйте другой, более простой метод: многократно делите n на 2 и выводите остатки в обратном порядке. Сначала напишите цикл `while` для выполнения этих вычислений и выведите биты в неправильном порядке. Затем используйте рекурсию для вывода битов в правильном порядке.

2.3.16. *Лист А4.* Ширина листа к его высоте в формате ISO составляет квадратный корень из 2 к 1. Формат А0 имеет площадь в 1 квадратный метр. Лист А1 представляет собой лист А0, разрезанный по вертикали на две одинаковые половины. Лист А2 представляет собой лист А1, разрезанный горизонтальной линией на две половины, и т. д. Напишите программу, которая получает целочисленный аргумент командной строки n и средствами `StdDraw` рисует деление листа А0 на 2^n частей.

2.3.17. *Перестановки.* Напишите программу `Permutations`, которая получает целое число `n` как аргумент командной строки и выводит все $n!$ перестановок из n букв, начиная с a (предполагается, что значение n не превышает 26). Перестановка n элементов представляет собой один из $n!$ вариантов упорядочения элементов. Например, при $n = 3$ результат должен выглядеть так (порядок перечисления вариантов несущественен):

```
bca cba cab acb bac abc
```

2.3.18. Перестановки размера k. Измените программу `Permutations` из предыдущего упражнения, чтобы она получала как аргументы командной строки два числа n и k и выводила все $P(n,k) = n! / (n - k)!$ перестановок, содержащих ровно k из n элементов. Для $k = 2$ и $n = 4$ результат должен выглядеть так (порядок перечисления снова несущественен):

```
ab ac ad ba bc bd ca cb cd da db dc
```

2.3.19. *Сочетания.* Напишите программу `Combinations`, которая получает целое число n в аргументе командной строки и выводит все 2^n сочетаний любого размера. *Сочетанием* называется подмножество n элементов с произвольным порядком элементов. Например, для $n = 3$ результат должен выглядеть так:

```
  a ab abc ac b bc c
```

Обратите внимание: среди комбинаций программа должна выводить пустую строку (подмножество размера 0).

2.3.20. *Сочетания длины k.* Измените программу `Combinations` из предыдущего упражнения, чтобы она получала как аргументы командной строки два числа n и k и выводила все $C(n,k) = n! / (k!(n - k)!)$ комбинаций размера k. Например, для $n = 5$ и $k = 3$ результат должен выглядеть так:

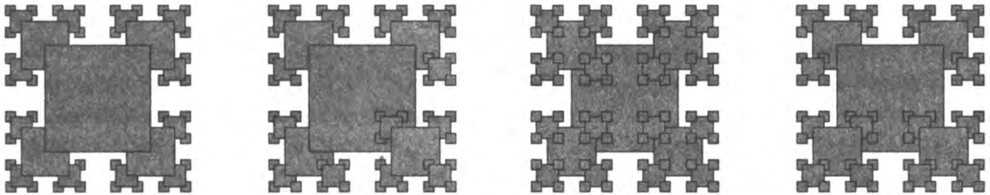
```
abc abd abe acd ace ade bcd bce bde cde
```

2.3.21. *Расстояние Хемминга.* Расстояние Хемминга между двумя строками битов длины n равно количеству позиций, в которых эти две строки различаются. Напишите программу, которая читает целое число k и битовую строку s из командной строки и выводит все битовые строки, у которых расстояние Хемминга от s не превышает k. Например, если $k = 2$, а $s = 0000$, ваша программа должна вывести следующий результат:

```
0011 0101 0110 1001 1010 1100
```

Подсказка: потребуются выборки (сочетания) k битов из строки s, состояние которых нужно изменить.

2.3.22. *Рекурсивные квадраты.* Напишите программу для построения каждого из следующих рекурсивных узоров. Отношение размеров квадратов составляет 2,2 : 1. Чтобы получить квадрат с контуром, изобразите закрашенный серый квадрат, а затем незакрашенный черный.



2.3.23. *Блины на тарелке.* Имеется стопка из n блинов разного размера. Ваша задача — упорядочить стопку так, чтобы самый большой блин находился внизу, а самый маленький — наверху. Разрешена только одна операция — перевернуть верхние k блинов, чтобы они лежали в противоположном порядке. Разработайте рекурсивный алгоритм для размещения блинов в нужном порядке, использующую не более $2n - 3$ переворачиваний.

2.3.24. *Код Грея.* Измените программу `Beckett` (листинг 2.3.3), чтобы она выводила слова кода Грея (а не просто последовательность позиций, где происходят изменения).

2.3.25. *Разновидность ханойских башен.* Рассмотрим следующую разновидность задачи о ханойских башнях. На трех стержнях нанизаны $2n$ дисков возрастающего размера. Изначально все диски нечетных размеров $(1, 3, ..., 2n - 1)$ находятся на левом стержне с возрастанием размеров сверху вниз; все диски с четным размером $(2, 4, ..., 2n)$ находятся на правом стержне. Напишите программу для вывода инструкций по перемещению нечетных дисков на правый стержень, а четных дисков на левый стержень по тем же правилам, что и в классической задаче о ханойских башнях.

2.3.26. *Анимация задачи о ханойских башнях.* Используйте `StdDraw` для анимации решения задачи о ханойских башнях. Диски должны перемещаться приблизительно 1 раз в секунду.

2.3.27. *Треугольники Серпинского.* Напишите рекурсивную программу для отрисовки треугольников Серпинского (см. листинг 2.2.3). Как и в программе `Htree`, используйте аргумент командной строки для управления глубиной рекурсии.

Треугольники Серпинского

2.3.28. *Биномиальное распределение.* Оцените количество рекурсивных вызовов, которые будут выполняться в следующем коде:

```
public static double binomial(int n, int k)
{
   if ((n == 0) && (k == 0)) return 1.0;
   if ((n < 0) || (k < 0))   return 0.0;
   return (binomial(n-1, k) + binomial(n-1, k-1))/2.0;
}
```

для вычисления `binomial(100, 50)`. Разработайте улучшенную реализацию, основанную на методе динамического программирования. *Подсказка*: см. упражнение 1.4.41.

2.3.29. *Функция Коллаца.* Рассмотрим следующую рекурсивную функцию, которая связана со знаменитой нерешенной задачей из теории чисел, известной как *задача Коллаца*, или задача $3n + 1$:

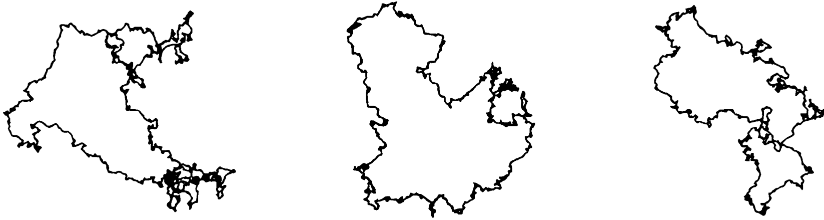
```
public static void collatz(int n)
{
   StdOut.print(n + " ");
   if (n == 1) return;
   if (n % 2 == 0) collatz(n / 2);
   else            collatz(3*n + 1);
}
```

Например, вызов `collatz(7)` выводит последовательность

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

в результате 17 рекурсивных вызовов. Напишите программу, которая получает аргумент командной строки n и возвращает значение $i < n$, обеспечивающее максимизацию количества рекурсивных вызовов для `collatz(i)`. Эта задача не решена в том смысле, что никто не знает, завершается ли функция для всех целых чисел (математическая индукция здесь не поможет, потому что один из рекурсивных вызовов получает значение аргумента большее, чем входной параметр «родительского» вызова).

2.3.30. *Броуновский остров.* Б. Мандельброт задал знаменитый вопрос: «Какова длина береговой линии Британии?» Измените программу `Brownian` и преобразуйте ее в программу `BrownianIsland`, которая рисует броуновские острова, береговая линия которых напоминает береговую линию Великобритании. Модификация проста: сначала измените `curve()` так, чтобы к координатам x и y прибавлялась случайная гауссова величина; затем измените метод `main()` так, чтобы он формировал кривую из точки в центре холста в эту же точку. Поэкспериментируйте с разными значениями параметров, чтобы ваша программа строила реалистичные изображения островов.



Броуновские острова с экспонентой Херста 0,76

2.3.31. *Плазменные облака.* Напишите рекурсивную программу для рисования плазменных облаков с использованием метода, предложенного в тексте.

2.3.32. *Странная функция.* Рассмотрите так называемую 91-функцию Маккарти:

```
public static int mcCarthy(int n)
{
   if (n > 100) return n - 10;
   return mcCarthy(mcCarthy(n+11));
}
```

Определите значение `mcCarthy(50)` без использования компьютера, а также количество рекурсивных вызовов `mcCarthy()`, требуемых для вычисления этого результата. Докажите, что базовый случай достигается для всех положительных целых `n`, или найдите значение `n`, для которого эта функция входит в бесконечный рекурсивный цикл.

2.3.33. *Рекурсивное дерево.* Напишите функцию `Tree`, которая получает число `n` как аргумент командной строки и строит следующие рекурсивные узоры для `n`, равных 1, 2, 3, 4 и 8.

2.3.34. Наибольшая палиндромная подстрока. Напишите программу `LongestPalindromicSubsequence`, которая получает строку как аргумент командной строки и находит в ней самую длинную подстроку, которая является палиндромом (одинаково читается как в прямом, так и в обратном направлении). *Подсказка*: ищите наибольшую общую подстроку исходной строки и результата ее обратной перестановки.

2.4. Пример: задача о протекании

Программные инструменты, которые рассматривались нами до настоящего момента, могут применяться для решения самых разных важных задач. Подводя итог изучению функций и модулей, мы рассмотрим пример программы для решения интересной научной задачи. В этом примере мы постараемся показать базовые элементы, рассмотренные ранее, в контексте трудностей, с которыми вы можете столкнуться при решении конкретной задачи, и продемонстрировать стиль программирования, который вы можете применять в самых разных программах.

В нашем примере известный вычислительный метод, называемый *методом Монте-Карло*, будет применяться для изучения естественной модели, называемой *протеканием*. Термин «метод Монте-Карло» широко применяется для обозначения любых вычислительных методов, применяющих случайные величины для оценки неизвестной величины посредством выполнения множественных испытаний. Мы уже использовали этот метод в других контекстах — например, в задаче о разорении игрока и задаче о собирателе купонов. Вместо того чтобы разрабатывать полноценную математическую модель или оценивать все возможные результаты эксперимента, мы полагаемся на законы вероятности.

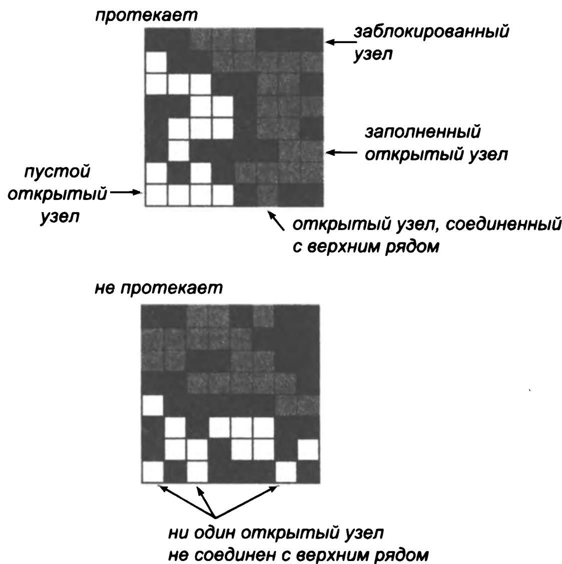
В этом примере рассматривается модель *протекания*, описывающая многие природные явления. Впрочем, нас прежде всего интересует процесс разработки модульных программ для решения вычислительных задач. Мы идентифицируем подзадачи, которые могут решаться независимо от других задач, при этом стараясь выявить ключевые абстракции и задавая себе вопросы вроде следующих: существует ли какая-то конкретная подзадача, которая поможет в решении этой задачи? какими важнейшими характеристиками обладает эта подзадача? может ли решение, которое ориентировано на эти важнейшие характеристики, пригодиться при решении других задач? Время, проведенное за поиском ответов на такие вопросы, не пропадет даром, потому что они упрощают создание, отладку и повторное использование программ, чтобы вы могли быстрее заняться решением основной задачи.

Задача о протекании

Локальные взаимодействия в системе нередко обусловливают ее глобальные свойства. Например, электротехника может интересовать композитный материал, состоящий из случайно распределенных изолирующих и металлических компонентов: какая их часть должна быть металлической, чтобы материал был проводящим? Или другой пример: геолога может интересовать пористая повехность, покрытая водой (или с залежами нефти в глубине). При каких условиях вода сможет просачиваться через основание водоема (или нефть просочится на поверхность)? Для моделирования таких ситуаций ученые определили абстрактный процесс, называемый *протеканием*. Этот процесс был всесторонне изучен; как выяснилось, он предоставляет точную модель для невероятно широкого спектра задач, помимо изолирующих материалов и пористых веществ, — в том

числе для распространения лесных пожаров и эпидемий, эволюции или анализа структуры Интернета.

Для простоты мы начнем работать в двух измерениях и смоделируем систему в виде сетки из $n \times n$ *узлов*. Каждый узел либо *заблокирован*, либо *открыт*; открытые узлы изначально пусты. *Заполненный* узел представляет собой открытый узел, который может быть соединен с открытым узлом в верхнем ряду по цепочке соседних (слева, справа, сверху, снизу) открытых узлов. Если в нижнем ряду существует заполненный узел, такая система называется *протекающей*. Другими словами, система протекает, если при заполнении всех открытых узлов, соединенных с верхним рядом, будут заполнены некоторые открытые узлы в нижнем ряду. В примере с изолирующими/металлическими материалами открытые узлы соответствуют металлическим частицам, и в «протекающей» системе существует металлический путь от верха до низа (заполненные узлы проводят электричество). В примере с пористыми веществами открытые узлы соответствуют пустому пространству, через которое может течь вода, и в протекающей системе вода заполняет все открытые узлы и протекает сверху вниз.



Примеры протекания

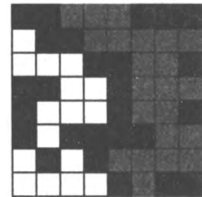
В известной научной задаче, которая интенсивно изучалась десятилетиями, ученых интересовал следующий вопрос: если узлы независимо назначаются открытыми с вероятностью p (и, следовательно, блокируются с вероятностью $1 - p$), какова вероятность того, что система будет протекать? Точное математическое решение этой задачи до сих пор не найдено. Наша цель — написать программу для изучения этой задачи.

Подготовительная стадия

Анализ задачи протекания в программах Java сталкивается с многочисленными проблемами и требует принятия многих решений, так что в итоге программа получится куда более объемной, чем короткие программы, встречавшиеся до сих пор. Мы постараемся продемонстрировать стиль программирования с *пошаговым наращиванием функциональности*, когда мы независимо разрабатываем модули, решающие части задачи, и последовательно наращиваем вычислительную инфраструктуру по ходу работы над задачей.

На первом шаге выбирается представление данных. Это решение может существенно повлиять на код, который мы напишем позднее, поэтому к нему следует отнестись как можно серьезнее. В самом деле, в ходе работы над представлением часто выясняется нечто такое, из-за чего нам приходится отправлять все наработки в мусор и начинать все заново.

протекающая система



заблокированные узлы

```
1 1 0 0 0 1 1 1
0 1 1 0 0 0 0 0
0 0 0 1 1 0 0 1
1 1 0 0 1 0 0 0
1 0 0 0 1 0 0 1
1 0 1 1 1 1 0 0
0 1 0 1 0 0 0 0
0 0 0 0 1 0 1 1
```

открытые узлы

```
0 0 1 1 1 0 0 0
1 0 0 1 1 1 1 1
1 1 1 0 0 1 1 0
0 0 1 1 0 1 1 1
0 1 1 1 0 1 1 0
0 1 0 0 0 0 1 1
1 0 1 0 1 1 1 1
1 1 1 1 0 1 0 0
```

заполненные узлы

```
0 0 1 1 1 0 0 0
0 0 0 1 1 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 1
0 0 0 0 1 1 1 1
0 0 0 0 0 1 0 0
```

Представления задачи о протекании

Для задачи о протекании эффективное представление очевидно: массив $n \times n$. Какой тип данных следует выбрать для представления элементов? Например, можно использовать целые числа: 0 — пустой узел, 1 — заблокированный узел, 2 — заполненный узел. Также можно заметить, что узлы обычно описываются в контексте вопросов: узел открыт или заблокирован? Пуст он или полон? Такой способ описания элементов наводит на мысль об использовании массивов $n \times n$, где каждый элемент равен `true` или `false`. Такие двумерные массивы называются *бинарными матрицами*. Код, в котором используются бинарные матрицы, проще кода первого варианта.

Бинарные матрицы — фундаментальные математические объекты с множеством практических применений. Язык Java не поддерживает операции с бинарными матрицами непосредственно, но вы можете использовать методы `StdArrayIO` (см. листинг 2.2.2) для их чтения и записи. Этот выбор демонстрирует основной принцип, часто встречающийся в программировании: усилия, необходимые для построения более универсального инструмента, обычно окупаются.

Со временем возникнет необходимость в работе с произвольными данными, но также следует предусмотреть возможность чтения и записи в файлы, потому что отладка программ со случайными входными данными может быть неэффективной. При каждом запуске программы появляется новый вариант входных данных; но после исправления ошибки вам хотелось бы видеть только что использованные данные, чтобы убедиться в том, что исправление прошло успешно. Соответственно лучше начать

с конкретных, хорошо проверенных и понятных наборов данных, которые хранятся в файлах в формате, совместимом с `StdArrayIO` (строка с размерами матрицы, после которой перечисляются значения 0 и 1 по строкам).

Когда вы начнете работать над новой задачей, в которой используются разные файлы, обычно лучше создать новую папку (каталог), чтобы отделить эти файлы от других, с которыми вы тоже можете продолжать работать. Например, для всех файлов этого примера можно создать папку с именем `percolation`. Работу можно начать с реализации и отладки базового кода для чтения и записи данных, создания тестовых файлов, проверки совместимости файлов с кодом и т. д., причем еще до того, как вы вообще займетесь задачей о протекании. Такой код, иногда называемый *служебным*, реализуется достаточно просто, но если вы с самого начала напишете его на должном уровне, это облегчит вам решение основной задачи.

А пока можно заняться кодом проверки бинарной матрицы на протекание. За основу целесообразно принять интерпретацию задачи как модели того, что произойдет при заполнении верхнего ряда водой: протечет она до нижнего ряда или нет? Нашим первым решением по проектированию станет создание метода `flow()`; этот метод получает в качестве аргумента бинарную матрицу `isOpen[][]`, которая определяет, какие узлы являются открытыми, и возвращает другую бинарную матрицу `isFull[][]`, которая показывает, какие узлы заполнены. О том, как будет реализован этот метод, сейчас можно вообще не беспокоиться; пока мы только решаем, как будет организован программный код. Также очевидно, что клиентский код должен использовать метод `percolates()`, который проверяет, что у массива, возвращаемого `flow()`, в нижней строке присутствует хотя бы один заполненный узел.

Все эти решения отражены в программе `Percolation` (листинг 2.4.1). Никакие интересные вычисления в этой программе не выполняются, но после запуска и отладки этого кода можно приступать к решению самой задачи. Метод, не выполняющий никаких реальных вычислений (такой, как `flow()`), иногда называется *заглушкой*. Наличие заглушки позволяет тестировать и отлаживать методы `percolates()` и `main()` в том контексте, в каком они нам понадобятся. Служебный код, содержащийся в листинге 2.4.1, обеспечивает поддержку, необходимую для разработки программы. Полностью отладив и реализовав этот код (большая часть которого нам все равно понадобится) с самого начала, мы закладываем прочную основу для построения кода, непосредственно относящегося к решению задачи. Нередко после завершения реализации служебный код удаляется (или заменяется другим кодом).

Вертикальное протекание

Если задана бинарная матрица, представляющая открытые узлы, как определить, протекает она или нет? Как будет показано позднее в этом разделе, эти вычисления напрямую связаны с одним из фундаментальных вопросов информатики. А пока рассмотрим заметно упрощенную версию задачи, которую мы назовем *вертикальным протеканием*.

Листинг 2.4.1. Служебный код Percolation

```
public class Percolation
{
   public static boolean[][] flow(boolean[][] isOpen)
   {
      int n = isOpen.length;
      boolean[][] isFull = new boolean[n][n];
      // Здесь будет вычисляться матрица isFull[][].
      return isFull;
   }

   public static boolean percolates(boolean[][] isOpen)
   {
      boolean[][] isFull = flow(isOpen);
      int n = isOpen.length;
      for (int j = 0; j < n; j++)
         if (isFull[n-1][j]) return true;
      return false;
   }

   public static void main(String[] args)
   {
      boolean[][] isOpen = StdArrayIO.readBoolean2D();
      StdArrayIO.print(flow(isOpen));
      StdOut.println(percolates(isOpen));
   }
}
```

n	размер системы (n × n)
isFul[][]	заполненные узлы
isOpen[][]	открытые узлы

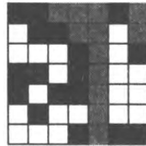
Начиная работу над задачей, мы реализуем и отлаживаем этот код, который обеспечивает выполнение служебных действий, создающих окружение для основных вычислений. Основная функция flow() возвращает бинарную матрицу с заполненными узлами (в заготовке кода этого не происходит). Вспомогательная функция percolates() проверяет нижнюю строку возвращаемой матрицы для принятия решения о том, существует ли протекание в системе. Тестовый метод-клиент main() читает бинарную матрицу из стандартного потока ввода и выводит результаты вызовов flow() и percolates() для этой матрицы.

```
% more testEZ.txt
5 5
0 1 1 0 1
0 0 1 1 1
1 1 0 1 1
1 0 0 0 1
0 1 1 1 1
```

```
% java Percolation < testEZ.txt
5 5
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
false
```

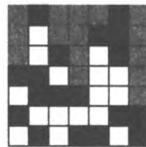
В упрощенном варианте мы ограничимся вертикальными соединительными путями. Если такой путь соединяет верхний ряд с нижним, мы говорим, что в системе существует *вертикальное протекание* по этому пути (и что в самой системе существует вертикальное протекание). Вероятно, это ограничение интуитивно понятно, если речь идет о просыпании песка сквозь бетонную плиту, но не о просачивании сквозь нее воды или об электрической проводимости. Несмотря на упрощение, вертикальное протекание как задача представляет самостоятельный интерес, потому что оно поднимает различные математические вопросы. Насколько принципиально это различие? Сколько путей вертикального протекания можно ожидать?

вертикальное протекание



узел связан с верхним рядом вертикальным путем

вертикального протекания нет



нет вертикальных узлов, соединенных с верхним рядом вертикальным путем

Вертикальное протекание

Определение узлов, которые заполняются по какому-либо пути, соединенному по вертикали с верхним рядом, выполняется достаточно просто. Нужно инициализировать верхнюю строку массива-результата верхней строкой матрицы системы; заполненные узлы при этом соответствуют открытым. Затем, двигаясь сверху вниз, мы заполняем каждую строку массива, проверяя соответствующую строку матрицы открытых узлов. Таким образом, при движении сверху вниз заполняются строки `isFull[][]`: все узлы, соответствующие узлам `isOpen[][]` и связанные по вертикали с уже заполненным узлом предыдущего ряда, помечаются значением `true`. В листинге 2.4.2 приведена реализация `flow()` для `Percolation`, которая возвращает бинарную матрицу заполненных узлов (`true`, если узел соединен с верхним рядом вертикальным путем, `false` в противном случае).

Листинг 2.4.2. Обнаружение вертикального протекания

```
public static boolean[][] flow(boolean[][] isOpen)
{  // Вычисление заполненных узлов для вертикального протекания.
   int n = isOpen.length;
   boolean[][] isFull = new boolean[n][n];
   for (int j = 0; j < n; j++)
      isFull[0][j] = isOpen[0][j];
   for (int i = 1; i < n; i++)
      for (int j = 0; j < n; j++)
         isFull[i][j] = isOpen[i][j] && isFull[i-1][j];
   return isFull;
}
```

n	размер системы (n × n)
isFul[][]	заполненные узлы
isOpen[][]	открытые узлы

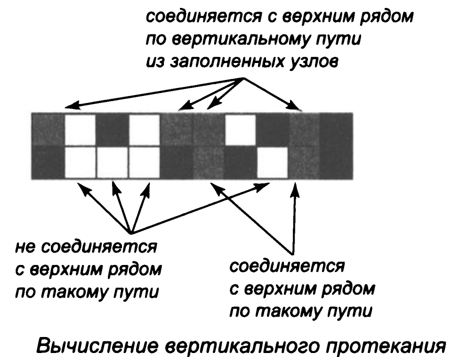
Подставляя этот метод на место заглушки в листинге 2.4.1, мы видим, что тестовый пример решается именно так, как и ожидалось (см. текст).

```
% more test5.txt
5 5
0 1 1 0 1
0 0 1 1 1
1 1 0 1 1
1 0 0 0 1
0 1 1 1 1
```

```
% java Percolation < test5.txt
5 5
0 1 1 0 1
0 0 1 0 1
0 0 0 0 1
0 0 0 0 1
0 0 0 0 1
true
```

Тестирование

Когда вы убедитесь в том, что написанный код работает в первом приближении, следует опробовать его на более широком наборе тестовых примеров и заняться некоторыми теоретическими вопросами. На этой стадии исходный служебный код уже не приносит такой пользы, как в начале, поскольку вводить большие бинарные матрицы через стандартный поток ввода и выводить их в стандартный поток вывода, особенно при большом количестве тестовых примеров, становится слишком неудобно. Вместо этого следует автоматически генерировать тестовые примеры и следить за тем, как наш код работает с ними, чтобы убедиться в его правильности. А если говорить конкретнее, для обретения уверенности и лучшего понимания кода мы ставим перед собой следующие две цели.



Вычисление вертикального протекания

- Протестировать код для больших случайных бинарных матриц.
- Оценить вероятность того, что в системе существует протекание для заданного p.

Для достижения этих целей нам понадобятся новые тестовые клиенты — более сложные по сравнению с тем служебным кодом, который использовался для общей проверки программы. Суть модульного стиля программирования заключается в создании таких клиентов в отдельных независимых классах *без изменения кода, относящегося к самой задаче о протекании.*

Визуализация данных

Если использовать для вывода `StdDraw`, мы сможем работать с матрицами существенно большего размера для этой задачи. Следующий статический метод — `Percolation` позволяет визуально отобразить содержимое бинарных матриц с делением холста `StdDraw` на квадраты, по одному для каждого узла:

```
public static void show(boolean[][] a, boolean which)
{
   int n = a.length;
   StdDraw.setXscale(-1, n);
   StdDraw.setYscale(-1, n);
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         if (a[i][j] == which)
            StdDraw.filledSquare(j, n-i-1, 0.5);
}
```

Второй аргумент `which` указывает, какие квадраты нужно закрасить — соответствующие элементам `true` или `false`. Этот метод не связан непосредственно с основными вычислениями, но возможность визуального представления данных для задач большой размерности, безусловно, будет полезной. Использование метода `show()` для отображения бинарных матриц с заблокированными и заполненными узлами разными цветами обеспечивает привлекательное визуальное представление задачи.

Метод Монте-Карло

Наш код должен правильно работать для любой бинарной матрицы. Более того, в интересующем нас научном вопросе задействованы случайные бинарные матрицы. Для этого в `Percolation` нужно добавить еще один статический метод:

```
public static boolean[][] random(int n, double p)
{
   boolean[][] a = new boolean[n][n];
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         a[i][j] = StdRandom.bernoulli(p);
   return a;
}
```

Метод генерирует бинарную матрицу $n \times n$ заданного размера n со случайным содержимым — каждый отдельный ее элемент принимает значение `true` с вероятностью p.

После отладки кода для нескольких тестовых примеров можно переходить к его тестированию на случайных данных. В реальных тестовых примерах могут обна-

ружиться новые ошибки; внимательно проверьте результаты. С другой стороны, после того как код будет отлажен для небольшой размерности, дальше можно действовать более уверенно. После исправления очевидных ошибок вам будет проще сосредоточиться на новых ошибках.

При наличии таких инструментов тестовый клиент для проверки работы кода задачи на гораздо больших наборах тестовых данных строится очень просто. Программа `PercolationVisualizer` (листинг 2.4.3) состоит из метода `main()`, который получает значения *n* и *p* из командной строки и выводит результат расчета протекания.

Такие клиенты весьма типичны. В конечном итоге мы хотим получить точную оценку вероятности протекания (вероятно, в результате выполнения большого количества испытаний), но этот простой инструмент дает возможность лучше понять задачу за счет анализа задачи на больших наборах тестовых данных (и в то же время получить уверенность в том, что код работает правильно). Прежде чем читать дальше, загрузите и выполните этот код с сайта книги для изучения процесса протекания. Если вы поэкспериментируете с выполнением `PercolationVisualizer` для умеренных значений *n* (допустим, от 50 до 100) и разных `p`, вы поймете, что с помощью этой программы можно получить ответы на некоторые вопросы о протекании. Понятно, что при низких значениях *p* протекания не будет, а при очень высоких — протекание есть почти всегда. Как система ведет себя для промежуточных значений *p*? Как изменяется поведение с ростом *n*?

Оценка вероятностей

Следующим шагом процесса разработки станет написание кода для оценки вероятности протекания в случайной системе (с размером *n* при вероятности открытого узла *p*). Эта величина будет называться ***вероятностью протекания***. Чтобы оценить ее значение, достаточно провести набор испытаний. Ситуация принципиально не отличается от бросков монеты (см. листинг 2.2.6), но вместо моделирования бросков мы генерируем случайную систему и проверяем, существует ли в ней протекание.

Программа `PercolationProbability` (листинг 2.4.4) инкапсулирует эти вычисления в методе `estimate()`, который получает три аргумента — `n`, `p` и `trials`, и возвращает оценку вероятности того, что в системе `n × n` с вероятностью открытого узла *p* существует протекание; для получения оценки генерируются `trials` экземпляров случайных систем и вычисляется, какая часть из них оказалась протекающими.

Сколько испытаний потребуется для получения точной оценки? Этот вопрос решается традиционными методами теории вероятностей и статистики, выходящими за рамки книги, однако мы можем получить некоторое представление о задаче на основании вычислительных данных. Всего из нескольких запусков `PercolationProbability` можно узнать, что если вероятность открытого узла близка к 0 или 1, большого количества испытаний не нужно, но для некоторых значений вычисление до двух знаков после запятой может потребовать до 10 000 испытаний. Чтобы исследовать ситуацию более подробно, можно изменить программу `PercolationProbability` для получения таких результатов, как в программе

Листинг 2.4.3. Клиент для визуализации

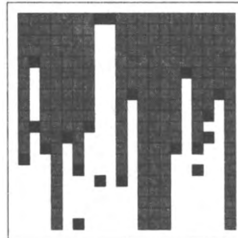
```
public class PercolationVisualizer
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      double p = Double.parseDouble(args[1]);
      StdDraw.enableDoubleBuffering();

      // Заблокированные узлы выводятся черным цветом.
      boolean[][] isOpen = Percolation.random(n, p);
      StdDraw.setPenColor(StdDraw.BLACK);
      Percolation.show(isOpen, false);

      // Полные узлы выводятся синим цветом.
      StdDraw.setPenColor(StdDraw.BOOK_BLUE);
      boolean[][] isFull = Percolation.flow(isOpen);
      Percolation.show(isFull, true);
      StdDraw.show();
   }
}
```

n	размер системы (n × n)
p	вероятность открытого узла
isOpen[][]	открытые узлы
isFul[][]	заполненные узлы

Клиент получает два числа n и p в аргументах командной строки, генерирует случайную систему n × n с вероятностью открытого узла p, определяет, какие узлы полны, и выводит результат средствами стандартной графики. На приведенных ниже диаграммах показаны результаты для вертикального протекания.

```
% java PercolationVisualizer 20 0.9
```

```
% java PercolationVisualizer 20 0.95
```



`Bernoulli` (листинг 2.2.6), с построением гистограммы точек данных для анализа распределений (см. упражнение 2.4.9). В методе `PercolationProbability.estimate()` проявляется качественное изменение в объеме выполняемых вычислений. Внезапно проведение тысяч испытаний становится абсолютно оправданным. Было бы неразумно проводить все эти испытания без предварительной отладки этих методов протекания. Кроме того, с этого момента нам придется учитывать время, необходимое для выполнения всех вычислений. Базовой методологии оценки этого времени посвящен раздел 4.1, но в данном случае структура программ достаточно проста, а вычисления происходят относительно быстро; чтобы убедиться

Листинг 2.4.4. Оценка вероятности протекания

```
public class PercolationProbability
{
   public static double estimate(int n, double p, int trials)
   {  // Генерирование случайных систем nxn; возвращается
      // эмпирическая оценка вероятности протекания.
      int count = 0;
      for (int t = 0; t < trials; t++)
      {  // Генерирование одной случайной бинарной матрицы nxn.
         boolean[][] isOpen = Percolation.random(n, p);
         if (Percolation.percolates(isOpen)) count++;
      }
      return (double) count / trials;
   }
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      double p = Double.parseDouble(args[1]);
      int trials = Integer.parseInt(args[2]);
      double q = estimate(n, p, trials);
      StdOut.println(q);
   }
}
```

n	размер системы (n × n)
p	вероятность открытого узла
trials	количество испытаний
isOpen[][]	открытые узлы
q	вероятность протекания

Метод estimate() генерирует случайные системы n × n с вероятностью открытого узла p и вычисляет, в какой части этих систем существует протекание. Этот процесс является бернуллиевским процессом, как и бросание монеты (см. листинг 2.2.6). Увеличение количества испытаний повышает точность оценки. Если значение p близко к 0 или 1, для достижения точной оценки потребуется не большое количество испытаний. Ниже приведены результаты, относящиеся к вертикальному протеканию.

```
% java PercolationProbability 20 0.05 10
0.0
% java PercolationProbability 20 0.95 10
1.0
% java PercolationProbability 20 0.85 10
0.7
% java PercolationProbability 20 0.85 1000
0.564
% java PercolationProbability 40 0.85 100
0.1
```

в этом, достаточно просто запустить программу. При выполнении T испытаний, в каждом из которых задействовано n^2 узлов, общее время выполнения `PercolationProbability.estimate()` пропорционально n^2T. Если увеличить T в 10 раз (для повышения точности), время выполнения увеличивается примерно десятикратно. Если увеличить n в 10 раз (для изучения протекания в больших системах), время выполнения увеличивается приблизительно в 100 раз.

Можно ли при помощи этой программы вычислить вероятности протекания в системе с миллиардами узлов с точностью до семи знаков? Ни один компьютер не бу-

дет достаточно быстрым, чтобы использовать `PercolationProbability.estimate()` для этой цели. Более того, в реальном научном эксперименте по протеканию значение n будет, скорее всего, намного выше. С помощью нашей модели можно рассчитывать экспериментально проверить гипотезы и для намного более крупных систем, но на уровне моделирования отдельных атомов. Поэтому упрощения такого рода в научных исследованиях совершенно необходимы.

Загрузите программу `PercolationProbability` с сайта книги, чтобы получить представление как о вероятностях протекания, так и о времени, необходимом для их вычисления. При этом вы не только узнаете больше о протекании, но и сможете проверить только что выдвинутые гипотезы о длительности моделирования для описанных моделей. Какова вероятность того, что в системе с вероятностью открытого узла p существует вертикальное протекание? Вертикальное протекание — достаточно простой случай, чтобы элементарная вероятностная модель давала точное значение этой величины, в чем можно убедиться экспериментально с помощью `PercolationProbability`. Так как мы занялись изучением вертикального протекания только для того, чтобы иметь простую отправную точку в разработке программного кода для изучения задачи протекания, мы оставляем дальнейшее изучение вертикального протекания читателю для самостоятельной работы (см. упражнение 2.4.11) и обратимся к основной задаче.

Рекурсивное решение

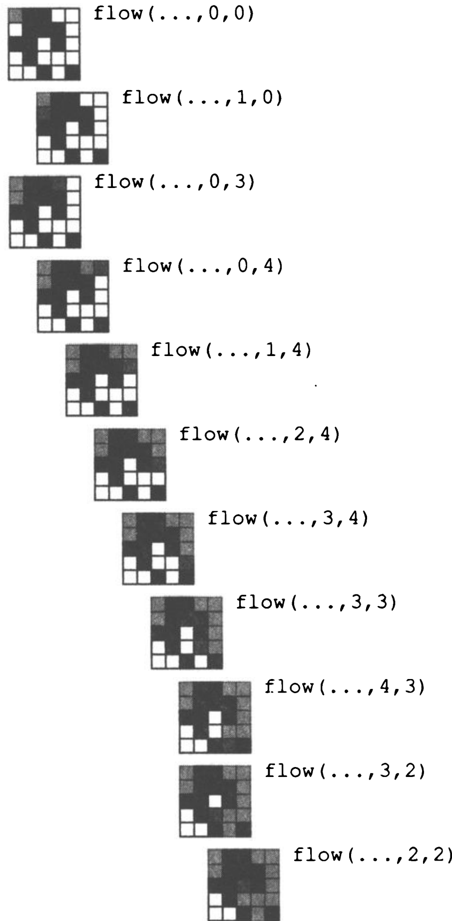
Как проверить существование протекания в системе в общем случае — когда достаточно существования любого пути, который начинается в верхнем ряду и завершается в нижнем (не только вертикального)?

Примечательно, что эта задача может быть решена компактной программой, основанной на классическом рекурсивном алгоритме *поиска в глубину*. Листинг 2.4.5 содержит реализацию метода `flow()`, который вычисляет матрицу `isFull[][]` с использованием рекурсивной версии `flow()`, получающей четыре аргумента: матрицу открытых узлов `isOpen[][]`, матрицу протекания `isFull[][]` и позицию узла, заданную индексом строки `i` и индексом столбца `j`. Базовым случаем является рекурсивный вызов, который просто возвращает управление (мы будем называть такой вызов *холостым вызовом*) по одной из следующих причин.

- Либо `i`, либо `j` выходит за границы массива.
- Узел заблокирован (элемент `isOpen[i][j]` содержит `false`).
- Узел уже помечен как заполненный (`isFull[i][j]` содержит `true`).

На шаге свертки узел помечается как заполненный, после чего выполняются рекурсивные вызовы для четырех соседей узла: `isOpen[i+1][j]`, `isOpen[i][j+1]`, `isOpen[i][j-1]` и `isOpen[i-1][j]`. Версия `flow()` с одним аргументом вызывает рекурсивный метод для каждого узла верхнего ряда. Рекурсия всегда завершается, потому что каждый рекурсивный вызов либо является холостым, либо помечает

новый узел как заполненный. Мы можем доказать посредством индукции (как это часто бывает в рекурсивных программах), что узел помечается как заполненный в том и только в том случае, если он соединен с одним из узлов верхнего ряда.



Рекурсивное решение задачи о протекании (холостые вызовы опущены)

Трассировка работы `flow()` для маленького тестового примера дает много полезной информации. Вы увидите, что она вызывает `flow()` для каждого узла, который может быть достигнут по пути из открытых узлов, начиная от верхнего ряда. Пример показывает, что простые рекурсивные программы могут скрывать в себе вычисления, весьма сложные в остальных отношениях. Этот метод является частным случаем алгоритма поиска в глубину, который находит много важных практических применений[1].

[1] Например, очень похожим образом действуют хорошо описанные в литературе алгоритмы закраски контурных изображений. — *Примеч. науч. ред.*

Листинг 2.4.5. Поиск протекания

```
public static boolean[][] flow(boolean[][] isOpen)
{  // Заполнение всех узлов, доступных от верхнего ряда.
   int n = isOpen.length;
   boolean[][] isFull = new boolean[n][n];
   for (int j = 0; j < n; j++)
      flow(isOpen, isFull, 0, j);
   return isFull;
}
public static void flow(boolean[][] isOpen,
                        boolean[][] isFull, int i, int j)
{  // Заполнение каждого узла, доступного от (i, j).
   int n = isFull.length;
   if (i < 0 || i >= n) return;
   if (j < 0 || j >= n) return;
   if (!isOpen[i][j]) return;
   if (isFull[i][j]) return;
   isFull[i][j] = true;
   flow(isOpen, isFull, i+1, j);  // Внизу.
   flow(isOpen, isFull, i, j+1);  // Справа.
   flow(isOpen, isFull, i, j-1);  // Слева.
   flow(isOpen, isFull, i-1, j);  // Вверх.
}
```

n	размер системы (n × n)
isOpen[][]	открытые узлы
isFull[][]	заполненные узлы
i, j	текущий узел (строка, столбец)

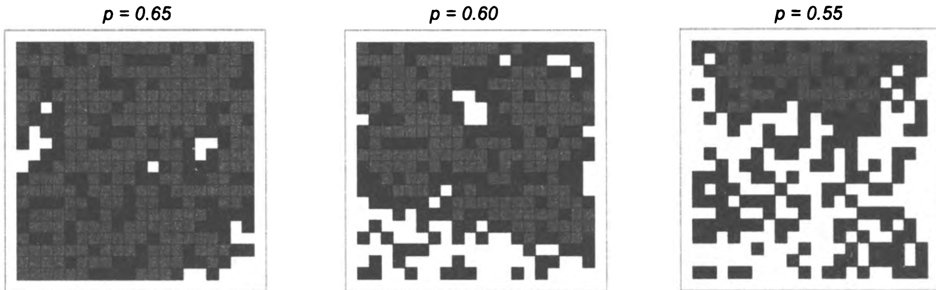
Если заменить заглушки из листинга 2.4.1 этими методами, мы получим решение задачи о протекании путем поиска в глубину. Рекурсивный метод flow() присваивает true элементу isFull[][], соответствующему любому узлу, который может быть достигнут от isOpen[i][j] по цепочке соседних открытых узлов. Метод flow() с одним аргументом вызывает рекурсивный метод для каждого узла верхнего ряда.

```
% more test8.txt
8 8
0 0 1 1 1 0 0 0
1 0 0 1 1 1 1 1
1 1 1 0 0 1 1 0
0 0 1 1 0 1 1 1
0 1 1 1 0 1 1 0
0 1 0 0 0 0 1 1
1 0 1 0 1 1 1 1
1 1 1 1 0 1 0 0
```

```
% java Percolation < test8.txt
8 8
0 0 1 1 1 0 0 0
0 0 0 1 1 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 1
0 0 0 0 1 1 1 1
0 0 0 0 0 1 0 0
true
```

Чтобы избежать конфликта с нашим решением о вертикальном протекании (листинг 2.4.2), мы можем переименовать его в `PercolationVertical` и создать новую копию `Percolation` (листинг 2.4.1), заменив в нем заглушку `flow()` двумя методами из листинга 2.4.5. После этого разработанные ранее инструменты `PercolationVisualizer` и `PercolationProbability` можно использовать для визуализации и экспериментов с новым алгоритмом. Если вы поэкспериментируете с разными значениями n и p, вы быстро получите представление о моделируемой

ситуации: в системах всегда существует протекание при высокой вероятности открытого узла p и никогда — при низких значениях p. Кроме того (особенно с ростом n), существует значение p, выше которого в системах (почти) всегда есть протекание и ниже которого протекания (почти) никогда не бывает.



Протекание становится менее вероятным с уменьшением вероятности открытого узла p

Отладив классы `PercolationVisualizer` и `PercolationProbability` для простого процесса вертикального протекания, мы сможем с большей уверенностью использовать их для изучения более общего случая протекания и быстрее перейти к изучению основной научной задачи. Обратите внимание: если вы захотите вернуться к экспериментам с вертикальным протеканием, вам придется отредактировать код `PercolationVisualizer` и `PercolationProbability`, чтобы они обращались снова к `PercolationVertical` вместо `Percolation`, или написать другие тестовые клиенты `PercolationVertical` и `Percolation`, выполняющие методы обоих классов для их сравнения.

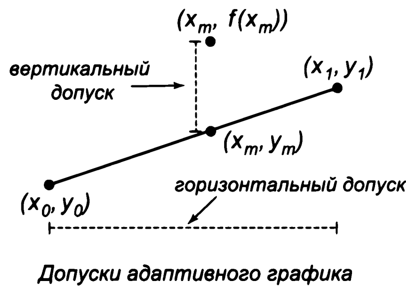
Адаптивный график

Чтобы расширить понимание процессов протекания, следующим шагом разработки станет написание программы, которая строит график вероятности протекания как функции от вероятности открытого узла p для заданного значения n. Конечно, графики проще всего строятся по математической формуле такой функции; но для протекания найти такую формулу еще никому не удалось, поэтому приходится действовать другим способом: провести моделирование и нанести результаты на график.

И тут мы немедленно сталкиваемся с многочисленными вопросами. Сколько значений p следует вычислить для получения вероятности протекания? Какие значения p следует выбрать? На какую точность нужно ориентироваться в таких вычислениях? Из всех этих решений образуется задача планирования эксперимента. Как бы нам ни хотелось мгновенно получить точную визуализацию графика для любого заданного n, затраты на вычисления могут оказаться

неприемлемыми. Например, первое, что приходит в голову, — построить график для 100 или 1000 равномерно распределенных точек средствами `StdStats` (программа 2.2.5). Но, как вы узнали после использования `PercolationProbability`, вычисление достаточно точного значения вероятности протекания для каждой точки может занять несколько секунд, так что на построение всего графика может потребоваться несколько минут, часов и даже больше. При этом совершенно очевидно, что значительная часть этого времени будет потеряна напрасно, потому что мы знаем, что для малых p значения равны 0, а для больших p они равны 1. Это время было бы лучше потратить на более точные вычисления для промежуточных значений p. Что же делать?

Программа `PercolationPlot` (листинг 2.4.6) реализует рекурсивное решение с такой же структурой, как у программы `Brownian` (листинг 2.3.5), часто применяемое в подобных ситуациях. Основная идея проста: мы выбираем максимальное расстояние, допустимое между значениями координаты x (которое мы будем называть *горизонтальным допуском*), максимальную допустимую погрешность по координате y (*вертикальный допуск*) и количество выполняемых испытаний T для каждой точки. Рекурсивный метод рисует график в заданном интервале $[x_0, x_1]$, от точки (x_0, y_0) к точке (x_1, y_1). В нашей задаче график рисуется от точки (0, 0) до точки (1, 1). В базовом случае (если расстояние между x_0 и x_1 меньше горизонтального допуска или если расстояние между линией, соединяющей две конечные точки, и значением функции в средней точке меньше допустимой погрешности) просто чертится отрезок из (x_0, y_0) в (x_1, y_1). На шаге свертки программа (рекурсивно) строит две половины графика, от (x_0, y_0) до $(x_m, f(x_m))$ и от $(x_m, f(x_m))$ до (x_1, y_1).



Допуски адаптивного графика

Код `PercolationPlot` относительно прост, и он строит симпатичный график с относительно низкими затратами ресурсов. Вы можете использовать его для анализа формы зависимости при разных значениях n или выбрать меньшие допуски, чтобы быть уверенным в том, что график близок к фактическим значениям. В принципе, можно вывести точные математические оценки качества аппроксимации, но, возможно, лучше не вдаваться в такие подробности во время экспериментов и исследований, так как наша цель ограничивается разработкой гипотезы о протекании, которую можно было бы проверить экспериментально.

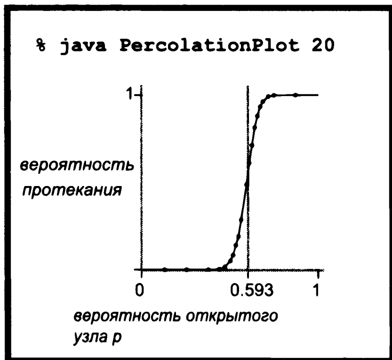
Листинг 2.4.6. Клиент, строящий адаптивный график

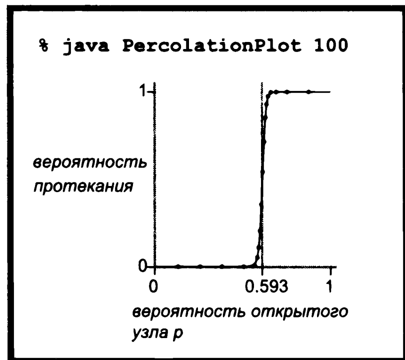
```
public class PercolationPlot
{
   public static void curve(int n,
                            double x0, double y0,
                            double x1, double y1)
   {  // Проведение экспериментов и графический
      //вывод результатов.
      double gap = 0.01;
      double err = 0.0025;
      int trials = 10000;
      double xm  = (x0 + x1)/2;
      double ym  = (y0 + y1)/2;
      double fxm = PercolationProbability.estimate(n, xm, trials)
      if (x1 - x0 < gap || Math.abs(ym - fxm) < err)
      {
         StdDraw.line(x0, y0, x1, y1);
         return;
      }
      curve(n, x0, y0, xm, fxm);
      StdDraw.filledCircle(xm, fxm, 0.005);
      curve(n, xm, fxm, x1, y1);
   }

   public static void main(String[] args)
   {  // Построение графика экспериментальной зависимости кривой
      // для системы nxn.
      int n = Integer.parseInt(args[0]);
      curve(n, 0.0, 0.0, 1.0, 1.0);
   }
}
```

n	размер системы
x0, y0	левая граница
x1, y1	правая граница

xm, ym	средняя точка
fxm	значение в средней точке
gap	горизонтальный допуск
err	вертикальный допуск
trials	количество испытаний

Эта рекурсивная программа рисует график зависимости вероятности протекания (на основании экспериментальных наблюдений) от вероятности открытого узла p (управляющая переменная) для случайных систем n × n.





И действительно, графики, построенные программой `PercolationPlot`, немедленно подтверждают гипотезу о существовании порогового значения (около 0,593): если p больше порога, то в системе почти гарантированно существует протекание; если же p меньше порога, то в системе протекание почти всегда отсутствует. С ростом n кривая приближается к ступенчатой функции, которая в пороговой точке меняет значение с 0 на 1. Это явление, называемое *фазовым переходом*, встречается во многих физических системах.

За простой формой вывода листинга 2.4.6 кроется огромный объем вычислений. Например, кривая для $n = 100$ состоит из 18 точек, каждая из которых вычисляется в результате 10 000 испытаний, а в каждом испытании задействовано n^2 узлов. Каждый узел генерируется и проверяется в программе несколькими операциями, так что график строится в результате выполнения *миллиардов* операций. Из этого наблюдения следует вынести два урока. Во-первых, вы должны быть полностью уверены в любой строке кода, которая будет выполняться миллиарды раз, поэтому наше внимание к пошаговому процессу разработки и отладки кода оправданно. Во-вторых, хотя нас могут интересовать гораздо большие системы, обработка случаев действительно большой размерности потребует дополнительного изучения теории вычислений, а именно разработки более быстрых алгоритмов, а также инфраструктуры для определения их характеристик быстродействия.

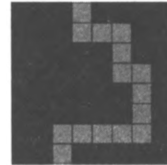
Благодаря повторному использованию всех программных наработок мы можем изучать всевозможные разновидности задачи о протекании, просто реализуя разные методы `flow()`. Например, если исключить последний рекурсивный вызов из рекур-

```
PercolationPlot.curve()
   PercolationProbability.estimate()
      Percolation.random()
         StdRandom.bernoulli()
            .
            . n² раз
            .
           StdRandom.bernoulli()
      return
      Percolation.percolates()
         flow()
         return
      return
      .
      . T раз
      .
       Percolation.random()
         StdRandom.bernoulli()
           .
           . n² раз
           .
         StdRandom.bernoulli()
      return
      Percolation.percolates()
         flow()
         return
      return
   return
   .
   . по одному для каждой точки
   .
   PercolationProbability.estimate()
      Percolation.random()
         StdRandom.bernoulli()
           .
           . n² раз
           .
         StdRandom.bernoulli()
      return
      Percolation.percolates()
         flow()
         return
      return
      .
      . T раз
      .
       Percolation.random()
         StdRandom.bernoulli().
           .
           . n² раз
           .
         StdRandom.bernoulli()
      return
      Percolation.percolates()
         flow()
         return
      return
   return
return
```

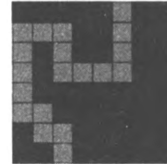
Трассировка вызовов для PercolationPlot

сивного метода `flow()` в листинге 2.4.5, программа будет проверять существование так называемого *направленного протекания*, при котором не рассматриваются пути, идущие вверх. Эта модель может сыграть важную роль в таких ситуациях, как протекание жидкости через пористый камень под воздействием силы тяжести, но, например, не в ситуации с электрической проводимостью. Запустив `PercolationPlot` для обоих методов, сможете ли вы обнаружить различия (см. упражнение 2.4.10)?

протекание существует (путь никогда не идет вверх)



протекания нет



Направленное протекание

Для моделирования реальных физических процессов, таких как протекание воды через пористые материалы, необходимо использовать трехмерные массивы. Существует ли аналогичный порог в трехмерной задаче? Если так, то какова его величина? Поиск в глубину эффективно работает для изучения этого вопроса, хотя введение еще одного измерения требует уделять еще больше внимания вычислительным затратам на моделирование (см. упражнение 2.4.18). Ученые также изучают более сложные решетчатые структуры, которые плохо моделируются многомерными массивами, — моделирование таких структур будет рассматриваться в разделе 4.5.

Изучение протекания посредством компьютерного моделирования представляет интерес, потому что вывести аналитическое выражение для вычисления пороговых значений в ряде моделей реальных явлений еще не удалось. Ученые могут определить эти значения только посредством моделирования. Они проводят эксперименты, чтобы узнать, отражает ли модель протекания те явления, которые наблюдаются в природе, — возможно, посредством уточнения модели (например, переходом на другую решетчатую структуру). Протекание — один из примеров широкого круга задач, для которых вычислительные методы (такие, как в данном случае) являются важной частью анализа.

Выводы

Вероятно, для изучения задачи о протекании и получения графиков, которые рисует программа из листинга 2.4.6, можно было бы спроектировать и написать программу из нескольких сотен строк. На заре компьютерной техники у программиста не было особого выбора — ему приходилось работать с такими программами и тратить невероятно много времени на выявление ошибок и исправление неудачных решений на стадии проектирования. Современные средства программирования, такие как Java, позволяют более эффективно тратить время: программист использует модульный стиль программирования, описанный в этом разделе, не забывая о полезных уроках, которые он усвоил в процессе обучения.

Ожидайте наличия ошибок в программе

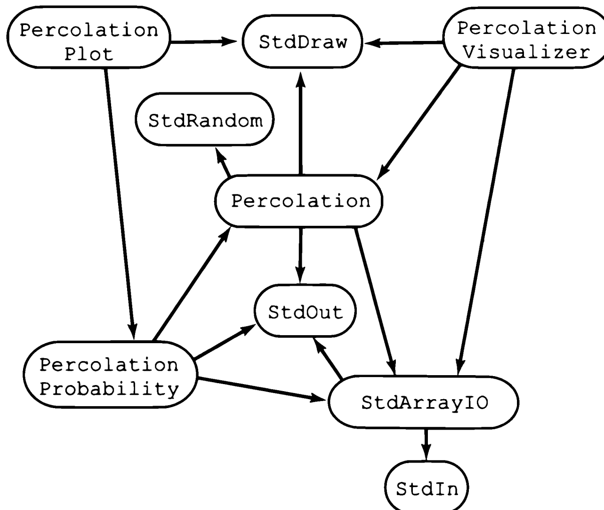
Любой нетривиальный код, написанный вами, будет содержать одну-две ошибки (а то и много больше). Если вы будете выполнять небольшие фрагменты кода с небольшими, понятными тестовыми сценариями, вам будет проще найти ошибки, а потом исправить их. А когда код будет отлажен, вы сможете использовать свою библиотеку при построении любых новых приложений.

Ограничивайте размер модулей

Вряд ли вы сможете удерживать в голове более нескольких десятков строк кода одновременно, поэтому старайтесь разбивать свой код на небольшие модули. Некоторые классы, содержащие библиотеки взаимосвязанных методов, со временем могут разрастись до сотен строк кода; в остальных случаях лучше работать с небольшими файлами.

Ограничивайте взаимодействия между модулями

В хорошо спроектированной модульной программе большинство модулей должно зависеть от минимального количества других модулей. В частности, модуль, обращающийся с вызовами к большому количеству других модулей, следует разбить на меньшие части. Особое внимание обратите на модули, которые сами вызываются многими другими модулями, — если вы внесете изменения в API такого модуля, эти изменения придется повторять во всех его клиентах.



Граф зависимостей наших примеров (без включения системных вызовов)

Выполняйте разработку с постепенным наращиванием функциональности

Запускайте и отлаживайте каждый мелкий модуль по мере его написания. В этом случае вам в любой момент времени не придется работать более чем с несколькими десятками строк непроверенного кода. Если вы поместите весь свой код в один большой модуль, трудно быть уверенным в том, что какая-то часть этого модуля свободна от ошибок. Раннее выполнение кода также заставляет задуматься о форматах ввода/вывода, природе экземпляров задачи и других подобных вопросах на более ранней стадии. Информация, которую вы получите во время обдумывания таких вопросов и отладки соответствующего кода, повысит эффективность кода, написанного на более поздних стадиях процесса.

Решите упрощенную задачу

Хоть какое-то работающее решение — лучше, чем никакого, поэтому обычно стоит начинать работу с создания самого простого кода, решающего поставленную задачу (как мы сделали в случае с вертикальным протеканием). Эта реализация становится первым шагом на пути постоянного усовершенствования и доработки кода, пока вы развиваете более полное понимание задачи на основе анализа более широкого спектра тестовых примеров и разрабатываете вспомогательный программный код, такой как наши классы `PercolationVisualizer` и `PercolationProbability`.

Поищите рекурсивное решение

Рекурсия — незаменимый инструмент в современном программировании; научитесь доверять ему. Если простота и элегантность `Percolation` и `PercolationPlot` вас в этом еще не убедили, попробуйте разработать нерекурсивную программу для проверки протекания в системе, а потом подумайте еще раз.

Создавайте свои инструменты, когда это уместно

Наш метод визуализации `show()` и метод генерирования бинарной матрицы `random()`, безусловно, пригодятся и в других приложениях — например, в методе построения адаптивного графика `PercolationPlot`. Эти методы совсем несложно включить в соответствующие библиотеки. Создавать такие методы общего назначения ничуть не сложнее (а может, и проще), чем реализовывать заново специализированные методы именно для задачи о протекании.

Повторно используйте программный код, если это возможно

Библиотеки `StdIn`, `StdRandom` и `StdDraw` упростили процесс разработки кода в этом разделе. Кроме них мы смогли повторно использовать для протекания такие программы, как `PercolationVisualizer`, `PercolationProbability` и `PercolationPlot`, хотя они были разработаны для *вертикального* протекания. Стоит вам написать несколько похожих программ, и вы обнаружите, что их версии можно повторно

использовать в других моделях Монте-Карло или других задачах анализа экспериментальных данных.

Этот пример прежде всего должен был убедить вас в том, что с модульным программированием можно сделать гораздо больше, чем без него. И хотя не существует единственно верного стиля программирования, инструменты и методология, описанные в этом разделе, позволят браться за сложные задачи, которые в традиционной реализации могли бы оказаться непосильными.

Успех модульного программирования — всего лишь начало. Современные системы программирования используют куда более гибкую модель программирования, чем модель «класс как библиотека статических методов», которую мы рассматривали. В следующих двух главах мы разовьем эту модель вместе с множеством других примеров, демонстрирующих ее полезность.

Вопросы и ответы

В. Редактирование `PercolationVisualizer` и `PercolationProbability` для переименования `Percolation` в `PercolationVertical` (или другое имя метода, который мы хотим проанализировать) создает лишние хлопоты. Нельзя ли как-то обойтись без этого?

О. Да, мы вернемся к этой ключевой проблеме в главе 3. А пока вы можете хранить реализации в разных подкаталогах, но и это может создать путаницу. В принципе некоторые расширенные возможности Java (такие, как *classpath*) могут вам помочь, но и у них есть свои недостатки.

В. Рекурсивный метод `flow()` внушает мне сомнения. Как лучше разобраться в том, что здесь происходит?

О. Опробуйте его на собственных небольших примерах, дополненных инструкциями для трассировки вызова функций. После нескольких запусков вы удостоверитесь в том, что он всегда помечает как заполненные все открытые узлы, соединенные с начальным узлом цепочкой смежных открытых узлов.

В. Существует ли простое нерекурсивное решение по выявлению всех заполненных узлов?

О. Есть несколько способов выполнения тех же базовых вычислений. Мы вернемся к задаче в разделе 4.5, когда будем рассматривать поиск в ширину. А пока попытка разработать нерекурсивную реализацию `flow()`, безусловно, станет полезным упражнением, если вас интересует эта тема.

В. Мне кажется, программа `PercolationPlot` (листинг 2.4.6) выполняет слишком много вычислений для такого простого графика. Нет ли более эффективного решения?

О. Лучшим решением была бы просто математическая формула, описывающая функцию, но за десятки лет исследований она так и не найдена. Пока ученые не найдут такую формулу, им придется использовать вычислительные эксперименты — вроде описанного в этом разделе.

Упражнения

2.4.1. Напишите программу, которая получает число n как аргумент командной строки и строит бинарную матрицу $n \times n$. Элемент в строке i и столбце j таблицы содержит `true`, если i и j являются взаимно простыми числами. Заполненная матрица выводится средствами стандартной графики (см. упражнение 1.4.16). Затем напишите аналогичную программу для вывода матрицы Адамара порядка n (см. упражнение 1.4.29). Наконец, напишите программу для вывода бинарной матрицы, у которой элемент на пересечении строки n и столбца j содержит `true`, если коэффициент x^j в $(1 + x)^i$ (биномиальный коэффициент) нечетен (см. упражнение 1.4.41). Возможно, закономерность, проявляющаяся в третьем примере, покажется вам неожиданной.

2.4.2. Реализуйте в программе `Percolation` метод `print()`, который выводит 1 для заблокированных узлов, 0 для открытых и * для заполненных.

2.4.3. Приведите рекурсивные вызовы для реализации `flow()` из листинга 2.4.5 при следующих входных данных:

```
3 3
1 0 1
0 0 0
1 1 0
```

2.4.4. Напишите клиент `Percolation` (по аналогии с `PercolationVisualizer`), который проводит серию экспериментов для значения n, полученного из командной строки, с вероятностью открытого узла p, возрастающей от 0 до 1 с заданным приращением (также передаваемым в командной строке).

2.4.5. Опишите порядок пометки узлов при использовании `Percolation` в системе без заблокированных узлов. Какой узел помечается последним? Какова глубина рекурсии?

2.4.6. Поэкспериментируйте с использованием `PercolationPlot` для рисования графиков различных математических функций (с заменой вызова `PercolationProbability.estimate()` другим выражением, вычисляющим математическую функцию). Опробуйте функцию $f(x) = \sin x + \cos 10x$ и посмотрите, как график адаптируется к периодической кривой. Найдите интересные графики для трех или четырех функций на ваш выбор.

2.4.7. Измените программу `Percolation` так, чтобы вычисления сопровождались анимацией с последовательным заполнением узлов. Проверьте свой ответ к предыдущему упражнению.

2.4.8. Измените программу `Percolation`, чтобы она вычисляла максимальную глубину рекурсии в ходе вычислений. Постройте график ожидаемого значения этой величины как функции от вероятности свободного узла p. Как изменится результат, если рекурсивные вызовы будут следовать в обратном порядке?

2.4.9. Измените программу `PercolationProbability`, чтобы она выдавала примерно такой же результат, как программа `Bernoulli` (листинг 2.2.6). *Дополнительное задание*: воспользуйтесь своей программой для проверки гипотезы о том, что данные имеют гауссово распределение.

2.4.10. Создайте программу `PercolationDirected`, которая проверяет существование направленного протекания (с исключением последнего рекурсивного вызова в рекурсивном методе `flow()` в листинге 2.4.5, как описано в тексте). Затем используйте `PercolationPlot` для рисования графика вероятности направленного протекания как функции от вероятности свободного узла p.

2.4.11. Напишите тестовый клиент для `Percolation` и `PercolationDirected`, который получает из командной строки вероятность свободного узла p и выводит оценку вероятности того, что в системе существует протекание, но нет вертикального протекания. Проведите достаточное количество экспериментов для получения оценки с точностью до трех знаков.

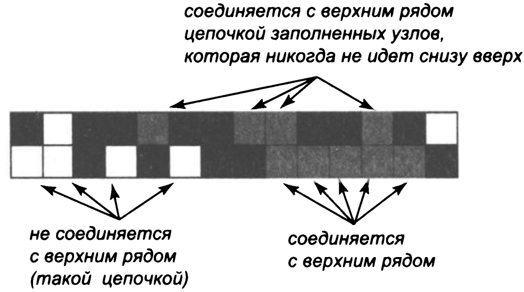
Упражнения повышенной сложности

2.4.12. *Вертикальное протекание.* Покажите, что в системе с вероятностью открытого узла p вертикальное протекание существует с вероятностью $1 - (1 - p^n)^n$. Используйте `PercolationProbability` для проверки своего аналитического результата для разных значений n.

2.4.13. *Протекание в прямоугольных системах.* Измените код из этого раздела так, чтобы он позволял исследовать протекание в прямоугольных системах. Сравните графики вероятности протекания в системах с соотношениями ширины к высоте 2:1 и 1:2.

2.4.14. *Адаптивные графики.* Измените программу `PercolationPlot`, чтобы она получала свои управляющие параметры (горизонтальный и вертикальный допуск, количество испытаний) как аргументы командной строки. Поэкспериментируйте с разными значениями параметров, чтобы проанализировать их влияние на качество кривой и затраты вычислительных ресурсов на ее вычисление. Кратко опишите свои результаты.

2.4.15. *Нерекурсивное направленное протекание.* Напишите нерекурсивную программу, которая проверяет существование направленного протекания с перемещением от верхнего до нижнего ряда, как и в случае с кодом вертикального протекания. Решение должно базироваться на следующей идее: если хотя бы один узел в смежной подстроке открытых узлов текущей строки соединен с заполненным узлом предыдущей строки, то все узлы подстроки становятся заполненными.

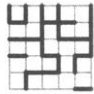


Моделирование направленного протекания

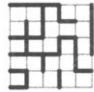
2.4.16. *Быстрая проверка протекания.* Измените рекурсивный метод `flow()` из листинга 2.4.5 так, чтобы он возвращал управление сразу же после обнаружения узла в нижней строке (и не заполнял новые узлы). *Подсказка*: используйте аргумент `done`, который равен `true`, если был достигнут нижний ряд, и `false` в противном случае. Приведите приближенную оценку коэффициента повышения производительности при запуске `PercolationPlot` с этим изменением. Используйте значения n, для которых программа выполняется как минимум несколько секунд, но не более нескольких минут. Обратите внимание: усовершенствование неэффективно, если первый рекурсивный вызов `flow()` не относится к узлу, находящемуся *под* текущим.

2.4.17. *Протекание по стыкам.* Напишите модульную программу для изучения протекания при условии, что связи обеспечиваются «стыками» между ячейками. Другими словами, ребро может быть либо полным, либо пустым, а протекание в системе существует в том случае, если существует путь, состоящий из заполненных ребер и проходящий от верхнего до нижнего ряда. *Примечание*: эта задача была решена аналитически, поэтому ваши модели должны подтвердить гипотезу о том, что порог протекания по стыкам при больших n стремится к 1/2.

протекание есть



протекания нет

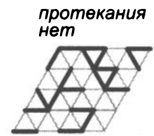


2.4.18. *Трехмерная задача о протекании.* Реализуйте класс `Percolation3D` и класс `BooleanMatrix3D` (для ввода/вывода и генерирования случайных значений) для изучения протекания в трехмерных кубах путем обобщения двумерного случая, рассмотренного в этом разделе. Система протекания выглядит как куб из $n \times n \times n$ узлов; каждый узел представляет собой единичный куб, открытый с вероятностью p и заблокированный с вероятностью $1 - p$. Пути могут соединять открытый код с открытым кубом, имеющим с ним общую грань (по одной для шести соседей, кроме внешних граней). В системе существует протекание, если существует путь, соединяющий хотя бы один открытый узел на нижней плоскости хотя бы с одним открытым узлом на верхней плоскости. Используйте рекурсивную версию `flow()`, как в листинге 2.4.5, но с восемью[1] рекур-

[1] Вероятно, здесь имелось в виду «шесть». Наибольшее число «соседей» у любого узла в 3-мерной матрице — шесть. — *Примеч. науч. ред.*

сивными вызовами вместо четырех. Постройте график зависимости вероятности протекания от вероятности открытого узла p для наибольшего значения n, которое вы можете обеспечить. Обязательно разрабатывайте свое решение с пошаговым наращиванием функциональности, как подчеркивалось в этом разделе.

2.4.19. *Протекание по стыкам на треугольной сетке.* Напишите модульную программу для изучения протекания по стыкам ячеек треугольной сетки: система состоит из $2n^2$ равносторонних треугольников, упакованных в ромбовидную сетку $n \times n$. Каждая внутренняя точка имеет шесть связей; каждая точка на внешней границе имеет четыре связи; каждая угловая точка — всего две.



2.4.20. *Игра «Жизнь».* Реализуйте класс `GameOfLife`, моделирующий игру Конуэя (игра «Жизнь»). Представьте бинарную матрицу, соответствующую системе ячеек, каждая из которых может быть либо «живой», либо «мертвой». Игра состоит из проверки и, возможно, обновления значения каждой ячейки в зависимости от состояния ее соседей (ближайших ячеек по всем направлениям, включая диагонали). «Живая» ячейка остается «живой», а «мертвая» — «мертвой», кроме случаев:

- «мертвая» ячейка, имеющая ровно трех «живых» соседей, становится «живой»;
- «живая ячейка», имеющая ровно одного[1] «живого» соседа, становится «мертвой»;
- «живая» ячейка, имеющая более трех «живых» соседей, становится «мертвой».

Инициализируйте бинарную матрицу случайным содержимым или воспользуйтесь одним из начальных узоров на сайте книги. Игра Конуэя была подробно изучена; она имеет прямое отношение к основам теории вычислений (за дополнительной информацией обращайтесь на сайт книги).

Пять поколений конфигурации типа «планер»

[1] Чаще встречается иная формулировка этого правила: «менее двух соседей», то есть одиночные ячейки, не имеющие соседей, также «умирают». — *Примеч. науч. ред.*

Глава 3

Объектно-ориентированное программирование

Ваш следующий шаг в программировании достаточно прост, если рассматривать его на концептуальном уровне. К настоящему моменту вы уже умеете пользоваться примитивными типами данных; в этой главе вы научитесь *использовать*, *создавать* и *проектировать* типы данных более высокого уровня.

Абстракция представляет собой упрощенное описание, которое отражает важнейшие элементы чего-либо, опуская при этом прочие подробности. В науке, технике и программировании все попытки разобраться в сложной системе всегда идут через абстракции. В языке Java для этого используется объектно-ориентированное программирование: в этой методологии большая (и, скорее всего, сложная) программа разбивается на множество взаимодействующих элементов — *объектов*. Эта идея берет начало от представления в программной логике таких сущностей реального мира, как электроны, люди, здания, планетарные системы и т. д., но она легко расширяется и для абстрактных сущностей: битов, чисел, цветов, изображений или даже самих программ.

Тип данных представляет собой множество значений и множество операций, определенных для этих значений[1]. Значения и операции для примитивных типов (таких, как `int` и `double`) в языке Java определены изначально. А в объектно-ориентированном программировании разработчик сам пишет определения новых типов данных. *Объект* представляет собой сущность, хранящую значение данных соответствующего типа; вы можете работать с этим значением, применяя одну из операций, определенных для этого типа. Возможность определять новые типы данных и выполнять операции с объектами, содержащими значения этих типов, также называется *абстракцией данных*. Она ведет нас к стилю модульного про-

---

[1] Здесь целесообразно немного пояснить используемую автором терминологию, поскольку в другой литературе, посвященной ООП, она может отличаться. В этой главе и далее *тип* данных — совокупность множества возможных значений и возможных операций над ними (реализуемых соответствующими подпрограммами — *методами*). В других источниках такую сущность чаще называют *классом*. Здесь же *класс* — реализация типа в виде модуля (файла) Java, однако иногда термин *класс* употребляется и как синоним типа; в таких случаях сохранена авторская терминология. *Объектом* принято называть конкретный экземпляр некоторого типа данных (или класса). — *Примеч. науч. ред.*

граммирования, который естественным образом расширяет *процедурный стиль программирования* для примитивных типов, описанный в главе 2. Абстракция типа данных позволяет выделить и обособить как данные, так и функции. В этой главе мы будем часто повторять другую мантру: *всегда, когда вы можете четко выделить данные и связанные с ними операции в своей программе, — сделайте это.*

3.1. Использование типов данных

Структурирование данных для обработки является важным шагом процесса разработки компьютерных программ. Программирование на языке Java в значительной степени основано на работе с типами данных, относящихся к категории *ссылочных типов*[1]. Они были спроектированы для поддержки *объектно-ориентированного программирования* — стиля программирования, упрощающего структурирование и обработку данных.

Теперь к восьми примитивным типам данных (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long` и `short`), которые вы уже использовали в своих программах, добавляются обширные библиотеки ссылочных типов, приспособленных для самых разных приложений. Один из примеров такого рода — тип данных `String` — вам уже встречался. В этом разделе вы больше узнаете о типе `String`, а также об использовании других ссылочных типов для работы с графикой и вводом/выводом. Одни из них встроены в Java (`String` и `Color`), другие были разработаны специально для этой книги (`In`, `Out`, `Draw` и `Picture`), но вы можете пользоваться ими как ресурсами общего назначения.

Наверняка вы заметили по первым двум главам книги, что наши программы в основном ограничиваются операциями с числами. Конечно, это объясняется тем, что примитивные типы Java представляют числа. Единственное исключение составляли строки — ссылочный тип, встроенный в Java. Со ссылочными типами можно писать программы, которые работают не только со строками, но и с изображениями, звуками и сотнями других абстракций, доступных в библиотеках Java или на сайте книги.

В этом разделе мы сосредоточимся на программах, которые *используют* существующие типы данных. Это поможет вам понять новые концепции и продемонстрирует широту их применения. Будут рассмотрены программы, работающие со строками, цветами, изображениями, файлами и веб-страницами, — большой шаг вперед по сравнению с примитивными типами из главы 1.

В следующем разделе будет сделан еще один шаг: вы научитесь *определять* собственные типы данных для реализации произвольных абстракций; ваше про-

---

[1] Здесь под ссылочными типами подразумеваются не *ссылки* как таковые (фактически разновидность *указателей*), а сложные *структурированные* типы (противопоставляемые *примитивным*). Для доступа к объектам такого типа используются, как правило, именно ссылки. — *Примеч. науч. ред.*

граммирование выйдет на совершенно новый уровень. Написание программ, работающих с вашими собственными типами данных, — исключительно мощный и практичный подход к программированию, который занимает ведущее положение уже много лет.

Базовые определения

Тип данных представляет собой множество значений и множество операций, определенных для этих значений. Это утверждение также будет довольно часто повторяться ввиду его важности. В главе 1 мы подробно рассматривали примитивные типы данных Java. Например, примитивный тип данных `int` представляет целые числа в диапазоне от -2^{31} до $2^{31} - 1$; к операциям, определенным для типа данных `int`, относятся базовые математические операции и сравнения, такие как +, *, %, < и >.

Также вы использовали в своих программах тип данных, не являющийся примитивным, — `String`. Вы знаете, что значения типа данных `String` представляют собой последовательности символов, и вы можете выполнить операцию конкатенации двух значений `String` для получения результата `String`. В этом разделе вы узнаете, что существуют десятки других операций для работы со строками: получение длины строки, извлечение отдельных символов, сравнение двух строк и т. д.

И хотя тип данных определяется множеством значений и множеством операций, определенных для них, при использовании типа данных нас будут в основном интересовать операции, а не значения. Когда вы пишете программы, использующие значения `int` или `double`, вас обычно не интересует то, как эти данные хранятся в памяти (мы вообще не обсуждали эти подробности); тот же принцип действует и при написании программ, использующих ссылочные типы, такие как `String`, `Color` или `Picture`. Другими словами, *чтобы использовать тип данных, вам не обязательно знать, как он реализован* (еще одна мантра).

Тип данных String

Давайте снова вернемся к типу данных Java `String` в контексте объектно-ориентированного программирования. Основных причин две: во-первых, вы уже использовали тип `String` с момента написания самой первой программы, так что он вам знаком. Во-вторых, работа со строками крайне важна во многих приложениях. Строки занимают центральное место при компиляции и запуске программ Java, а также при выполнении многих базовых вычислений; они лежат в основе систем обработки информации, необходимых для большинства бизнес-систем; пользователи повседневно работают с ними при вводе текста в приложениях электронной почты, блогах или чатах или при подготовке документов для публикации. Наконец, строки оказались ключевым элементом для прогресса в нескольких областях науки, в частности в молекулярной биологии.

Мы будем писать программы, которые объявляют, создают и используют значения типа `String`. Мы начнем с описания API типа `String` и документирования доступ-

ных операций. Затем будет рассмотрен механизм языка Java для *объявления переменных*, *создания объектов* для хранения значений типов данных и *вызова методов экземпляров* для выполнения операций с типом. Эти механизмы отличаются от соответствующих механизмов для примитивных типов, хотя вы заметите также и ряд общих особенностей.

API

Класс Java предоставляет механизм для определения типов данных. В классе задаются значения типа данных и реализуются операции с типом данных. Чтобы клиент мог использовать тип данных, не зная, как этот тип реализован, мы описываем поведение класса, перечисляя его методы экземпляров в *программном интерфейсе*, или API (Application Programming Interface), — по аналогии с тем, как это делалось для библиотек статических методов. API предоставляет информацию, необходимую для написания клиентских программ, использующих тип данных.

В следующей таблице приведена сводка наиболее часто используемых методов экземпляров из API класса Java; полный API содержит более 60 методов! Некоторые методы используют целые числа для обозначения индекса символа в строке; как и в случае с массивами, значения индексов начинаются с 0.

public class String (тип данных Java)		
	`String(String s)`	Создает строку с таким же значением, как у s
`int`	`length()`	Количество символов
`char`	`charAt(int i)`	Символ в позиции с индексом i
`String`	`substring(int i, int j)`	Символы с индексами от i до (j-1)
`boolean`	`contains(String substring)`	Содержит ли строка подстроку substring?
`boolean`	`startsWith(String pre)`	Начинается ли строка с префикса pre?
`boolean`	`endsWith(String post)`	Заканчивается ли строка суффиксом post?
`int`	`indexOf(String pattern)`	Индекс первого вхождения подстроки pattern
`int`	`indexOf(String pattern, int i)`	Индекс первого вхождения подстроки pattern после i
`String`	`concat(String t)`	Результат присоединения t к строке
`int`	`compareTo(String t)`	Сравнение строк
`String`	`toLowerCase()`	Строка, преобразованная к нижнему регистру
`String`	`toUpperCase()`	Строка, преобразованная к верхнему регистру
`String`	`replaceAll(String a, String b)`	Строка, в которой все вхождения a заменяются на b
`String[]`	`split(String delimiter)`	Строка между вхождениями подстрок delimiter
`boolean`	`equals(Object t)`	Совпадает ли значение строки со значением t?
`int`	`hashCode()`	Целочисленный хеш-код

Многие другие методы описаны в электронной документации и на сайте книги.

Выдержка из API типа данных Java String

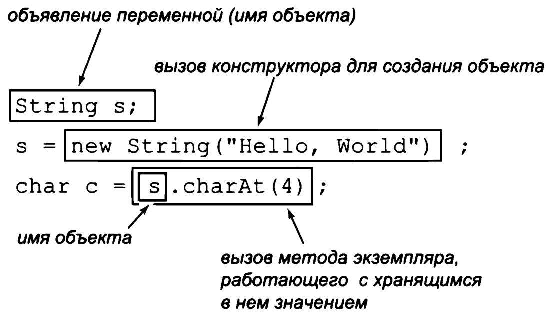
Первым в списке указан специальный метод, называемый *конструктором*, — его имя совпадает с именем класса, и он не имеет возвращаемого значения. Далее идут *методы экземпляров*, которые могут получать аргументы и возвращать значения точно так же, как статические методы, с которыми мы уже работали. Однако эти методы *не являются* статическими: они реализуют операции для конкретного экземпляра данных этого типа. Например, метод экземпляра `length()` возвращает количество символов в заданной строке, а `charAt()` возвращает ее символ с заданным индексом.

Объявление переменных

Переменные ссылочных типов объявляются точно так же, как и переменные примитивных типов: в строке с объявлением, состоящим из имени типа данных, за которым следует имя переменной. Например, в строке

```
String s;
```

объявляется переменная `s` типа `String`. При этом ничего не *создается*, а просто сообщается, что имя переменной `s` будет использоваться для обращения к объекту `String`. По общепринятой схеме имена ссылочных типов начинаются с букв верхнего регистра, а имена примитивных типов — с букв нижнего регистра[1].



Использование ссылочного типа данных

Создание объектов

В языке Java каждое значение какого-либо типа хранится в *объекте*. Когда клиент вызывает *конструктор*, виртуальная машина Java создает новый объект (также называемый *экземпляром*). Для вызова конструктора используется ключевое слово `new`, за которым следует имя класса и список аргументов конструктора, заключенных в круглые скобки и разделенных запятыми, как и в вызове статического метода. Например, команда `new String("Hello, World")` создает новый объект `String`, содержащий последовательность символов *Hello, World*. Как правило, в клиентском коде

---

[1] Тем не менее для переменных типа `String` авторы делают исключение в связи с тем, что это встроенный тип языка Java. — *Примеч. науч. ред.*

объявление переменной, вызов конструктора для создания объекта и присваивание этого объекта переменной выполняются в одной строке кода:

```
String s = new String("Hello, World");
```

Вы можете создать любое количество объектов одного класса; каждый объект существует самостоятельно, а его значение может совпадать или не совпадать со значением другого объекта того же типа. Например, код

```
String s1 = new String("Cat");
String s2 = new String("Dog");
String s3 = new String("Cat");
```

создает три разных объекта `String`. В частности, `s1` и `s3` соответствуют двум разным объектам, несмотря на то что эти объекты содержат одинаковые последовательности символов.

Вызов методов экземпляров

Самое важное различие между переменными ссылочного и примитивного типов заключается в том, что переменные ссылочного типа могут использоваться для вызова методов, реализующих операции именно с этим типом данных (в отличие от встроенного синтаксиса с такими операторами, как + и *, которые использовались с примитивными типами). Такие методы называются *методами экземпляров*. Вызов метода экземпляра напоминает вызов статического метода, если не считать того, что метод экземпляра связывается не только с классом, но и с конкретным объектом. Соответственно, для идентификации метода обычно используется *имя объекта* (переменная заданного типа) вместо имени класса. Например, если `s1` и `s2` — переменные типа String, определявшиеся ранее, `s1.length()` возвращает целое число 3, `s1.charAt(1)` возвращает символ `'a'`, а `s1.concat(s2)` возвращает новую строку `CatDog`.

```
String a = new String("now is");
String b = new String("the time");
String c = new String(" the");
```

вызов метода экземпляра	возвращаемый тип	возвращаемое значение
`a.length()`	`int`	`6`
`a.charAt(4)`	`char`	`'i'`
`a.substring(2, 5)`	`String`	`"w i"`
`b.startsWith("the")`	`boolean`	`true`
`a.indexOf("is")`	`int`	`4`
`a.concat(c)`	`String`	`"now is the"`
`b.replace("t", "T")`	`String`	`"The Time"`
`a.split(" ")`	`String[]`	`{ "now", "is" }`
`b.equals(c)`	`boolean`	`false`

Примеры операций с типом данных String

Сокращенная запись при работе с String

Как вы уже знаете, язык Java предоставляет специальную языковую поддержку для типа данных `String`. Объект `String` можно создать на основе строкового литерала (вместо явного вызова конструктора). Кроме того, конкатенацию двух строк можно выполнить оператором конкатенации (+) вместо явного вызова метода `concat()`. Мы привели длинную версию исключительно для того, чтобы продемонстрировать синтаксис, используемый с другими типами данных; эти две сокращенные формы записи уникальны для типа данных `String`[1].

сокращенная запись	`String s = "abc";`	`String t = r + s;`
полная запись	`String s = new String("abc");`	`String t = r.concat(s);`

Следующие фрагменты кода демонстрируют использование различных методов обработки строк. В этом коде четко проявляется идея разработки абстрактной модели и отделения кода, реализующего абстракцию, от кода, в котором она

извлечение имени и расширения файла из аргумента командной строки

```
String s = args[0];
int dot = s.indexOf(".");
String base = s.substring(0, dot);
String extension = s.substring(dot + 1, s.length());
```

вывод в стандартный ввод всех текстовых строк, содержащих строку, переданную в аргументе командной строки

```
String query = args[0];
while (StdIn.hasNextLine())
{
   String line = StdIn.readLine();
   if (line.contains(query))
      StdOut.println(line);
}
```

строка является палиндромом?

```
public static boolean isPalindrome(String s)
{
   int n = s.length();
   for (int i = 0; i < n/2; i++)
      if (s.charAt(i) != s.charAt(n-1-i))
         return false;
return true;
}
```

преобразование ДНК в мРНК (замена 'T' на 'U')

```
public static String translate(String dna)
{
   dna = dna.toUpperCase();
   String rna = dna.replaceAll("T", "U");
   return rna;
}
```

Типичный код, работающий со строками

[1] В других объектно-ориентированных языках синтаксис может поддерживать аналогичные «сокращенные» формы записи для произвольных классов. Так, в C++ пример с конкатенацией строк реализуется путем использования *перегруженного оператора* +. — *Примеч. науч. ред.*

используется. Эта особенность вообще характерна для объектно-ориентированного программирования, и мы подходим к поворотному моменту книги: практически весь код, который мы будем писать в будущем, будет основан на определении и вызове методов, реализующих операции с типами данных.

Практическое применение обработки строк: геномика

Чтобы вы получили практический опыт работы со строками, мы приведем очень краткий обзор из области *геномики* и рассмотрим программу, которая могла бы использоваться в биоинформатике для выявления потенциальных генов. Биологи используют для представления структурных элементов жизни простую модель, в которой буквы A, C, T и G представляют четыре базовых элемента ДНК живых организмов. В каждом живом организме эти структурные элементы образуют длинные цепочки (по одной для каждой хромосомы), называемые *геномами*. Понимание свойств генома — ключ к пониманию процессов, происходящих в живых организмах. В настоящее время известны геномные последовательности многих живых существ, включая геном человека, который представляет собой последовательность приблизительно из 3 миллиардов базовых элементов. После того как последовательности были идентифицированы, ученые начали писать компьютерные программы для изучения их структуры. Обработка строк в настоящее время является одной из важнейших методологий (экспериментальных или вычислительных) в молекулярной биологии.

Генное прогнозирование

Ген представляет собой подпоследовательность генома, представляющую функциональную единицу, играющую исключительно важную роль в понимании жизненных процессов. Ген состоит из последовательности *кодонов* (*триплетов*), каждый из которых состоит из трех базовых элементов, представляющих одну аминокислоту. Стартовый кодон ATG отмечает начало гена, а любой из завершающих кодонов TAG, TAA или TGA отмечает конец гена (причем никакие вхождения этих стоп-кодонов в других местах гена встречаться не могут). Одним из первых шагов в анализе генома становится идентификация его потенциальных генов, а эта задача относится к области обработки строк, для решения которой хорошо подходит тип данных Java `String`.

Программа `PotentialGene` (листинг 3.1.1) поможет сделать этот первый шаг. Функция `isPotentialGene()` получает в качестве аргумента строку с цепочкой ДНК и определяет, соответствует ли эта строка потенциальному гену по следующим признакам: длина должна быть кратна 3, цепочка начинается со стартового кодона, завершается стоп-кодоном и не содержит промежуточных стоп-кодонов. Для проверки в программе используются различные строковые методы: `length()`, `charAt()`, `startsWith()`, `endsWith()`, `substring()` и `equals()`.

Хотя реальные правила определения генов несколько сложнее описанной схемы, программа `PotentialGene` служит примером того, как навыки программирования

Листинг 3.1.1. Проверка потенциального гена

```
public class PotentialGene
{
   public static boolean isPotentialGene(String dna)
   {
      // Длина кратна 3.
      if (dna.length() % 3 != 0) return false;

      // Начинается со стартового кодона.
      if (!dna.startsWith("ATG")) return false;

      // Нет промежуточных стоп-кодонов.
      for (int i = 3; i < dna.length() - 3; i++)
      {
         if (i % 3 == 0)
         {
            String codon = dna.substring(i, i+3);
            if (codon.equals("TAA")) return false;
            if (codon.equals("TAG")) return false;
            if (codon.equals("TGA")) return false;
         }
      }

      // Завершается стоп-кодоном.
      if (dna.endsWith("TAA")) return true;
      if (dna.endsWith("TAG")) return true;
      if (dna.endsWith("TGA")) return true;

      return false;
   }
}
```

dna	анализируемая строка
codon	3 последовательных базовых элемента

Функция isPotentialGene() получает в качестве аргумента строку с цепочкой ДНК и определяет, соответствует ли эта строка потенциальному гену: длина кратна 3, цепочка начинается со стартового кодона (ATG), завершается стоп-кодоном (TAA, TAG или TGA) и не содержит промежуточных стоп-кодонов. За примером тестового клиента обращайтесь к упражнению 3.1.19.

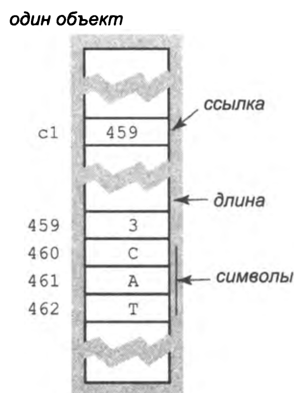
```
% java PotentialGene ATGCGCCTGCGTCTGTACTAG
true

% java PotentialGene ATGCGCTGCGTCTGTACTAG
false
```

повышают эффективность научной работы — в данном случае изучения геномных последовательностей.

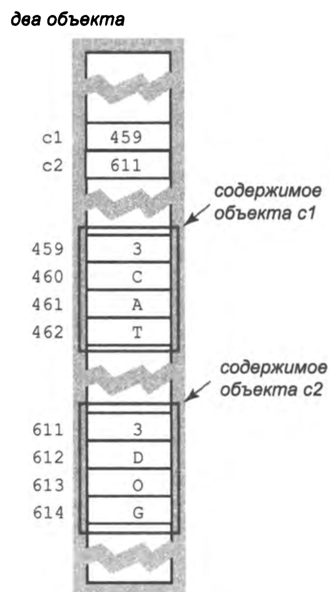
В текущем контексте тип данных `String` для нас интересен прежде всего тем, что он показывает, чем может стать тип данных — хорошо проработанной инкапсуляцией важной абстракции, нужной клиенту. Прежде чем переходить к другим примерам, мы рассмотрим некоторые базовые свойства ссылочных типов и объектов в языке Java.

Ссылки на объекты

Конструктор создает объект и возвращает *ссылку* на этот объект, а не сам объект (отсюда и название — *ссылочный тип*). Что такое «ссылка на объект»? Всего лишь механизм для обращения к объекту. В Java существует несколько разных способов реализации ссылок, но для их использования знать все подробности не обязательно. Тем не менее будет полезно составить мысленную модель одной из типичных реализаций. При выполнении `new` выделяется блок памяти для хранения текущего значения заданного типа и возвращается указатель на этот блок (адрес памяти). Мы будем называть блок памяти, выделенный для объекта, *содержимым* этого объекта.



Почему не манипулировать самим объектом, спросите вы? Для небольших объектов такое решение вполне разумно, но для больших приходится учитывать затраты ресурсов: экземпляр данных может занимать много места в памяти. Было бы неразумно копировать или перемещать все эти данные каждый раз, когда объект передается в метод как аргумент. Если этот довод кажется вам знакомым, то это потому, что ранее точно такие же рассуждения уже имели место, когда мы обсуждали в разделе 2.1 передачу массивов статическим методам. В самом деле, массивы являются объектами, как вы увидите позднее в этом разделе. Значения же переменных примитивных типов, напротив, хранятся в памяти естественным образом, непосредственно, поэтому использовать ссылки для всех обращений к ним не имеет смысла.



Свойства ссылок на объекты будут рассмотрены более подробно, когда мы обсудим примеры кода с использованием ссылочных типов.

Представление объектов в памяти

Использование объектов

Объявление переменной дает имя переменной, которое можно использовать в коде практически так же, как мы используем имя для переменной типа `int` или `double`:

- в аргументе или возвращаемом значении метода;
- в операции присваивания;
- в массиве.

Собственно, именно так мы использовали объекты `String` с самой первой программы `HelloWorld`: в большинстве наших программ вызывается метод `StdOut.println()` с аргументом `String`, и во всех программах имеется метод `main()`, получающий в качестве аргумента массив типа `String`. Как вы уже видели, существует еще один очень важный пункт этого списка, актуальный только для переменных, ссылающихся на объекты:

- для вызова метода экземпляра, определенного для типа.

Для переменных примитивных типов, у которых операции встроены в язык и выполняются только такими операторами, как +, -, * и /, этот способ использования недоступен.

Неинициализированные переменные

Когда вы объявляете переменную ссылочного типа, но не присваиваете ей никакого значения, переменная остается *неинициализированной*, а попытки ее использования приводят к таким же последствиям, как и с примитивными типами. Например, код

```
String bad;
boolean value = bad.startsWith("Hello");
```

приводит к ошибке компиляции, потому что программа пытается использовать неинициализированную переменную.

Преобразование типа

Если вы захотите преобразовать объект из одного типа в другой, вам придется написать код такого преобразования. Впрочем, часто это оказывается ненужным: значения разных типов данных настолько различны, что о преобразовании можно даже не задумываться. Например, как преобразовать объект `String` в объект `Color`? Однако есть один важный частный случай, когда такое преобразование чрезвычайно полезно: все ссылочные типы Java поддерживают специальный метод экземпляра `toString()`, который возвращает объект `String`. Способ преобразования полностью зависит от реализации, но обычно в строке кодируется значение объекта. Программисты часто вызывают метод `toString()` для трассировки в процессе отладки. Кроме того, Java автоматически вызывает метод `toString()` в некоторых ситуациях, включая конкатенацию строк и `StdOut.println()`. Например, для любой ссылки на объект `x` Java автоматически преобразует выражение `"x = " + x` в `"x = " + x.toString()`, а выражение `StdOut.println(x)` — в `StdOut.println(x.toString())`. Механизм языка Java, который делает возможным такое преобразование, будет рассмотрен в разделе 3.3.

Доступ к ссылочному типу данных

Как и в случае с библиотеками статических методов, код реализации каждого класса хранится в файле, имя которого совпадает с именем класса; таким файлам присваивается расширение .java. Чтобы написать клиентскую программу, исполь-

зующую тип данных, необходимо обеспечить для Java доступ к этому классу. Тип данных `String` является частью самого языка Java, поэтому он доступен всегда. Чтобы тип данных, определенный пользователем, стал доступным для вашей программы, следует либо поместить копию файла .java в каталог с кодом клиентского приложения, либо воспользоваться механизмом *пути к классам* (описан на сайте книги). После этого остается понять, как использовать тип данных в клиентском коде.

Различия между методами экземпляров и статическими методами

Наконец, вы готовы понять смысл модификатора `static`, который мы использовали в программах еще с листинга 1.1.1. Статические методы создаются обычно для реализации функций; методы экземпляров (нестатические) создаются для реализации операций с типом данных. Использование этих двух типов методов в клиентском коде также различается, потому что вызовы статических методов обычно начинаются с имени класса (его принято записывать с символа верхнего регистра), а вызовы методов экземпляров обычно начинаются с имени объекта (принято записывать с символа нижнего регистра). Краткая сводка основных различий приведена в следующей таблице, но после написания нескольких клиентских программ вы научитесь быстро различать их.

	метод экземпляра	**статический метод**
Пример вызова	`s.startsWith("Hello")`	`Math.sqrt(2.0)`
Указывается	Имя объекта (или ссылка на объект)	Имя класса
Параметры	Ссылки на объект и аргумент(ы)	Аргумент(ы)
Основное назначение	Работа со значением объекта	Вычисление возвращаемого значения

Сравнение методов экземпляров и статических методов

Основные концепции, которые мы только что рассмотрели, станут отправной точкой для изучения объектно-ориентированного программирования, поэтому будет нелишне их повторить. *Тип данных* представляет собой множество значений и множество операций, определенных для этих значений. Типы данных реализуются в независимых модулях, а разработчик пишет клиентские программы, которые их используют. *Объект* представляет собой *экземпляр* типа данных. Объекты характеризуются тремя основными характеристиками: *состоянием*, *поведением* и *содержимым*. *Состояние* объекта определяется значением его данных. *Поведение* объекта определяется операциями, определенными для этих данных. *Содержимым* объекта называется место в памяти, где хранятся данные объекта. В объектно-ориентированном программировании *объекты* создаются вызовом *конструктора*, после чего *методы экземпляров* вызываются для изменения их состояния. В языке Java операции с объектами осуществляются по *ссылкам на объекты*.

Чтобы продемонстрировать эффективность объектно-ориентированного подхода, мы рассмотрим несколько примеров. Начнем со знакомой области обработки графических изображений с использованием объектов `Color` и `Picture`. Затем мы вернемся к библиотекам ввода/вывода в контексте объектно-ориентированного программирования, для работы с информацией из файлов из Сети.

Цвет

Цвет представляет собой восприятие электромагнитного излучения человеческим глазом. Так как компьютеры часто используются для вывода и обработки цветных изображений, цвет принадлежит к числу распространенных абстракций в компьютерной графике, и в языке Java существует специальный тип данных `Color`. Работа с цветом — в профессиональном издательском деле, в печати и в Интернете — считается достаточно сложным делом. Например, внешний вид цветного изображения в значительной мере зависит от носителя, на котором оно выводится. Тип данных `Color` отделяет задачу графического дизайнера по выбору нужного цвета от проблем системы, связанных с точным воспроизведением этого цвета.

В библиотеках Java существуют сотни типов данных, поэтому вы должны явно указать, какие библиотеки Java будут использоваться в программе, чтобы избежать конфликтов имен. А если говорить более конкретно, в начало любой программы, использующей `Color`, следует включить директиву импорта (подключения) соответствующей библиотеки

```
import java.awt.Color;
```

(До настоящего момента мы ограничивались использованием стандартных библиотек Java или наших собственных библиотек, поэтому в явном импорте не было необходимости.)

Для представления цветовых значений `Color` использует цветовую модель RGB, в которой цвет задается целыми числами (лежащими в диапазоне от 0 до 255), представляющими интенсивность красной, зеленой и синей (соответственно) составляющей цвета. Все возможные цвета образуются смешением красной, зеленой и синей составляющих. Таким образом, значения типа `Color` состоят из трех 8-разрядных целых чисел. От нас не требуется знать, какой именно тип — `int`, `short` или `char` — используется для представления этих целых чисел. При таком представлении Java использует для кодирования цвета 24 бита, что позволяет представить $256^3 = 2^{24} \approx$ 16,7 миллиона возможных цветов. По оценкам ученых, человеческий глаз может различать около 10 миллионов разных цветов.

Конструктор типа данных `Color` получает три целочисленных аргумента. Например, можно использовать запись:

```
Color red      = new Color(255,   0,   0);
Color bookBlue = new Color(  9,  90, 166);
```

красный	зеленый	синий	
255	0	0	красный
0	255	0	зеленый
0	0	255	синий
0	0	0	черный
100	100	100	темно-серый
255	255	255	белый
255	255	0	желтый
255	0	255	малиновый

Некоторые значения цветов

Мы использовали цвета в `StdDraw` еще в разделе 1.5, но тогда наши возможности были ограничены множеством предопределенных цветов, таких как `StdDraw.BLACK`, `StdDraw.RED` и `StdDraw.PINK`. Теперь в вашем распоряжении появились миллионы цветов, которыми можно пользоваться в программах. Программа `AlbersSquares` (листинг 3.1.2) — клиент `StdDraw` для экспериментов с цветами.

Как обычно при знакомстве с новой абстракцией, мы представляем `Color`, описывая важнейшие элементы цветовой модели Java, не углубляясь в подробности. В API `Color` входят несколько конструкторов и более 20 методов; те, которые мы будем использовать, кратко перечислены ниже.

public class java.awt.Color	
Color(int r, int g, int b)	
int getRed()	Интенсивность красной составляющей
int getGreen()	Интенсивность зеленой составляющей
int getBlue()	Интенсивность синей составляющей
Color brighter()	Более светлая версия цвета
Color darker()	Более темная версия цвета
String toString()	Представление цвета в виде строки
String equals(Object c)	Значение цвета совпадает с c?

За информацией о других методах обращайтесь к электронной документации и к сайту книги

Часть API типа данных Color

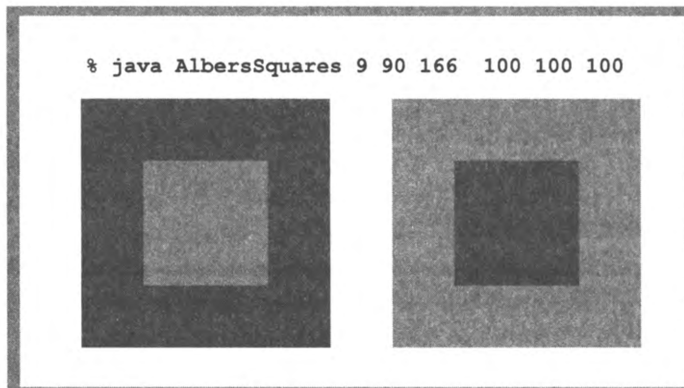
Сейчас мы воспользуемся типом `Color` как примером для демонстрации объектно-ориентированного программирования и одновременно разработаем некоторые полезные инструменты, которые могут пригодиться при написании программ, работающих с цветами. И поэтому мы выберем одно свойство цвета и постараемся показать на примере, что написание объектно-ориентированного кода для обработки абстрактных концепций (таких, как цвет) полезно и удобно.

Листинг 3.1.2. Квадраты Альберса

```
import java.awt.Color;
public class AlbersSquares
{
   public static void main(String[] args)
   {
      int r1 = Integer.parseInt(args[0]);
      int g1 = Integer.parseInt(args[1]);
      int b1 = Integer.parseInt(args[2]);
      Color c1 = new Color(r1, g1, b1);

      int r2 = Integer.parseInt(args[3]);
      int g2 = Integer.parseInt(args[4]);
      int b2 = Integer.parseInt(args[5]);
      Color c2 = new Color(r2, g2, b2);

      StdDraw.setPenColor(c1);
      StdDraw.filledSquare(.25, 0.5, 0.2);
      StdDraw.setPenColor(c2);
      StdDraw.filledSquare(.25, 0.5, 0.1);
      StdDraw.setPenColor(c2);
      StdDraw.filledSquare(.75, 0.5, 0.2);
      StdDraw.setPenColor(c1);
      StdDraw.filledSquare(.75, 0.5, 0.1);
   }
}
```

`r1, g1, b1`	значения RGB
`c1`	первый цвет
`r2, g2, b2`	значения RGB
`c2`	второй цвет

Программа выводит два цвета, заданные в формате RGB в командной строке. Цвета задаются в хорошо известном хроматическом формате, который был предложен в 1960-х годах теоретиком в области цвета Джозефом Альберсом.



Яркость

Качество изображения на современных экранах — ЖК-мониторах, плазменных телевизорах, экранах сотовых телефонов — зависит от понимания характеристики цвета, называемой *монохромной*, или *эффективной*, *яркостью*. Стандартная формула вычисления яркости отражает чувствительность глаза к красному, зеленому и синему цвету. Она вычисляется как линейная комбинация трех интенсивностей: если красная, зеленая и синяя составляющие равны r, g и b соответственно, то монохромная яркость Y вычисляется по следующей формуле:

$$Y = 0{,}299r + 0{,}587g + 0{,}114b.$$

Так как коэффициенты положительные, их сумма равна 1, а все интенсивности представляют собой целые числа в диапазоне от 0 до 255, — яркость является вещественным числом в диапазоне от 0 до 255.

Оттенки серого

Цветовая модель RGB обладает одним важным свойством: если все три интенсивности одинаковы, такой цвет представляет собой оттенок серого в диапазоне от черного (все интенсивности равны 0) до белого (все интенсивности равны 255). Если вы хотите вывести цветную фотографию в газете (или в книге), вам понадобится функция для преобразования цветного изображения в оттенки серого. Простой способ преобразования цвета в оттенки серого основан на замене цвета новым цветом, у которого красная, зеленая и синяя составляющие равны его монохромной яркости.

Совместимость цветов

Монохромная яркость также чрезвычайно важна для определения совместимости двух цветов в том смысле, что текст, выведенный одним цветом на фоне другого цвета, будет нормально читаться. Широко применяемое эмпирическое правило гласит, что разность между яркостью фона и текста должна быть не менее 128. Например, у черного текста на белом фоне разность яркостей равна 255, а у черного текста на синем фоне разность яркостей составляет всего 74. Это правило играет важную роль в графическом дизайне рекламы, дорожных знаков, веб-сайтов и в других случаях. Программа `Luminance` (листинг 3.1.3) содержит библиотеку статических методов, предназначенных для преобразования цвета в оттенки серого и для проверки совместимости двух цветов. Статические методы `Luminance` показывают, какую пользу приносят типы данных для структурирования информации. Использование объектов `Color` в аргументах и возвращаемых значениях существенно упрощает реализацию: вариант с передачей трех цветовых составляющих в аргументах выглядит громоздко, а вернуть сразу несколько значений без применения ссылочных типов не удастся.

Абстракция для представления цвета пригодится не только для непосредственного использования, но и для построения типов данных более высокого уровня со значениями `Color`. Далее для демонстрации этого утверждения мы построим на базе абстракции цвета новый тип данных, который может использоваться в программах для обработки цифровых изображений.

Листинг 3.1.3. Библиотека Luminance

```
import java.awt.Color;

public class Luminance
{
   public static double intensity(Color color)
   {  // Монохромная яркость цвета.
      int r = color.getRed();
      int g = color.getGreen();
      int b = color.getBlue();
      return 0.299*r + 0.587*g + 0.114*b;
   }

   public static Color toGray(Color color)
   {  // Преобразование в оттенки серого, используя монохромную яркость.
      int y = (int) Math.round(intensity(color));
      Color gray = new Color(y, y, y);
      return gray;
   }

   public static boolean areCompatible(Color a, Color b)
   {  // True, если цвета совместимы; иначе false.
      return Math.abs(intensity(a) - intensity(b)) >= 128.0;
   }

   public static void main(String[] args)
   {  // Совместимы ли два заданных цвета RGB?
      int[] a = new int[6];
      for (int i = 0; i < 6; i++)
         a[i] = Integer.parseInt(args[i]);
      Color c1 = new Color(a[0], a[1], a[2]);
      Color c2 = new Color(a[3], a[4], a[5]);
      StdOut.println(areCompatible(c1, c2));
   }
}
```

r, g, b	значения RGB
y	яркость цвета
a[]	значения типа int из args[]
c1	первый цвет
c2	второй цвет

Библиотека содержит три важные функции для работы с цветом: вычисление монохромной яркости, преобразование в оттенки серого и проверка совместимости фонового и основного цвета.

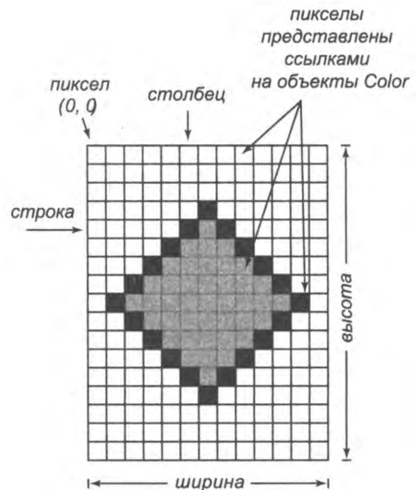
```
% java Luminance 232 232 232    0   0   0
true
% java Luminance   9  90 166  232 232 232
true
% java Luminance   9  90 166    0   0   0
false
```

Цифровая обработка изображения

Конечно, вы знакомы с концепцией *фотографии*. С технической точки зрения фотографию можно определить как двумерное изображение, которое образуется путем приема и фокусирования электромагнитных волн видимой части спектра и запечатлевает сцену в некоторый момент времени. Это техническое определение выходит за рамки книги, хотя стоит заметить, что история фотографии является историей технологического прогресса. В прошлом веке технология фотографии была основана на химических процессах, но ее будущее связано с компьютерами. Ваша камера и ваш мобильный телефон представляют собой компьютеры с линзами и светочувствительными устройствами, способными сохранять изображения в цифровой форме, а на вашем компьютере установлен графический редактор, в котором эти изображения можно редактировать. Фотографии можно обрезать, увеличивать и уменьшать, регулировать контраст, осветлять и затемнять, устранять эффект «красных глаз» и выполнять множество других операций. Как вы вскоре увидите, многие из этих операций реализуются на удивление легко при наличии простого типа данных, представляющего цветное изображение.

Цифровые изображения

Какой набор значений необходим для обработки цифровых изображений и какие операции будут выполняться с этими значениями? Базовая абстракция для экрана монитора не отличается от той, что используется в цифровой фотографии, и она очень проста: *цифровое изображение* представляет собой прямоугольную матрицу *пикселов* (элементов изображения), в которой задается цвет каждого пиксела. Такие цифровые изображения называются *растровыми*. С другой стороны, изображения, которые строятся средствами `StdDraw` (с участием геометрических объектов, таких как линии, точки, круги и квадраты), называются *векторными*.



Строение цифрового изображения

Класс `Picture` — тип данных для цифровых изображений, определение которых непосредственно вытекает из абстракции цифрового растрового изображения. Множество значений представляет собой не что иное, как двумерную матрицу значений `Color`, а в наборе операций тоже нет ничего неожиданного: создание пустого изображения заданной высоты и ширины, загрузка изображения из файла, присваивание пикселу заданного цвета, получение цвета заданного пиксела, возвращение ширины изображения, вывод изображения в окне на экране монитора и сохранение изображения в файле. В этом описании мы намеренно используем термин «*матрица*» вместо термина «*массив*», чтобы подчеркнуть,

что речь идет об абстракции (матрице пикселов), а не о конкретной реализации (двумерном массиве объектов `Color`). *Чтобы использовать тип данных, вам не обязательно знать, как он реализован.* Типичные изображения содержат так много пикселов, что реализация вполне может потребовать более эффективного представления, чем массив объектов `Color`. Как бы то ни было, для написания клиентских программ, работающих с изображениями, достаточно знать следующий API:

`public class Picture`	
`Picture(String filename)`	Создает изображение, используя содержимое файла
`Picture(int w, int h)`	Создает пустое изображение w × h
`int width()`	Возвращает ширину изображения
`int height()`	Возвращает высоту изображения
`Color get(int col, int row)`	Возвращает цвет пиксела (col, row)
`void set(int col, int row, Color c)`	Присваивает пикселу (col, row) цвет c
`void show()`	Выводит изображение в окне
`void save(String filename)`	Сохраняет изображение в файле

API типа данных для цифрового изображения

По общепринятой схеме координаты (0, 0) соответствуют левому верхнему пикселу, так что изображение хранится в порядке, типичном для двумерных массивов (с другой стороны, в схеме `StdDraw` точка (0, 0) находится в левом нижнем углу, так что система координат совпадает с классическими декартовыми координатами.) Многие программы обработки изображений представляют собой фильтры, которые перебирают все пикселы исходного изображения, после чего выполняют некоторые вычисления для определения цвета каждого пиксела результирующего изображения. Первый конструктор и метод `save()` поддерживают распространенные файловые форматы PNG и JPEG, так что вы можете самостоятельно писать программы для обработки ваших фотографий и размещать результаты в альбоме или на сайте. Окно для вывода изображения, создаваемое методом `show()`, имеет также функцию диалога для сохранения в файле. Эти методы в сочетании с типом данных Java `Color` открывают путь к цифровой обработке изображений.

Оттенки серого

На сайте книги представлено множество примеров цветных изображений, и все методы, упоминаемые в тексте, подходят для полноцветных изображений, но на всех печатных иллюстрациях в книге изображения представлены в оттенках серого. Соответственно нашей первой задачей станет написание программы, преобразующей многоцветные изображения в оттенки серого. Это одна из классических операций цифровой обработки графики: для каждого пиксела исходного изображения пикселу результирующего присваивается другой цвет. Программа `Grayscale` (листинг 3.1.4) представляет собой фильтр, который получает имя файла из командной строки и создает версию изображения в оттенках серого. Для этого

Листинг 3.1.4. Преобразование цветного изображения в оттенки серого

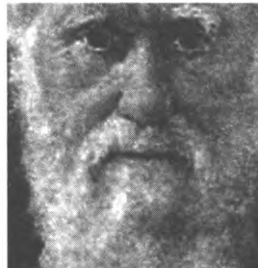
```
import java.awt.Color;

public class Grayscale
{
   public static void main(String[] args)
   {  // Вывод изображения в оттенках серого.
      Picture picture = new Picture(args[0]);
      for (int col = 0; col < picture.width(); col++)
      {
         for (int row = 0; row < picture.height(); row++)
         {
            Color color = picture.get(col, row);
            Color gray  = Luminance.toGray(color);
            picture.set(col, row, gray);
         }
      }
      picture.show();
   }
}
```

`picture`	изображение из файла
`col, row`	координаты пиксела
`color`	цвета пиксела
`gray`	пиксел в оттенках серого

Простое приложение для обработки цифрового изображения. Сначала он создает объект Picture, инициализированный данными из файла, имя которого передается в командной строке. Затем цвет каждого пиксела изображения преобразуется в оттенок серого путем присваивания ему соответствующего вычисленного значения оттенка серого. Наконец, программа выводит изображение. На иллюстрации справа, полученной увеличением изображения в низком разрешении, различимы отдельные пикселы (см. раздел «Масштабирование»).

`% java Grayscale mandrill.jpg`

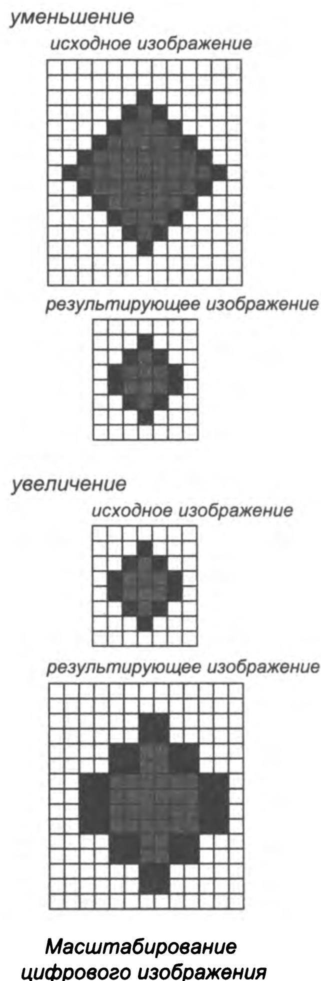
`% java Grayscale darwin.jpg`



она создает новый объект `Picture`, инициализированный данными цветного изображения, а затем присваивает цвету каждого пиксела новый объект `Color`, значение которого в оттенках серого вычисляется применением метода `toGray()` из `Luminance` (листинг 3.1.3) к цвету соответствующего пиксела источника.

Масштабирование

Одна из самых распространенных операций обработки изображений — увеличение или уменьшение изображения. К примерам этой базовой операции, называемой *масштабированием*, можно отнести создание миниатюрных фотографий для чатов или мобильных телефонов; изменение размера фотографии высокого разрешения, чтобы она помещалась в определенном месте на печатной странице или веб-странице; увеличение спутниковых фотографий или изображений, полученных с микроскопа. В оптических системах для достижения желаемого масштаба достаточно подвинуть линзу, но в цифровой графике потребуется более серьезная работа.



Масштабирование цифрового изображения

В некоторых случаях стратегия ясна. Например, если результирующее изображение вдвое меньше исходного (по обоим измерениям), можно просто выбрать половину пикселов — допустим, удалив половину строк и половину столбцов. Если приемник вдвое больше источника (по обоим измерениям), каждый исходный пиксел можно заменить четырьмя пикселами того же цвета. Обратите внимание: при уменьшении происходит потеря информации, поэтому если вы уменьшите изображение вдвое, а потом вдвое увеличите его, результат обычно не будет совпадать с исходным изображением.

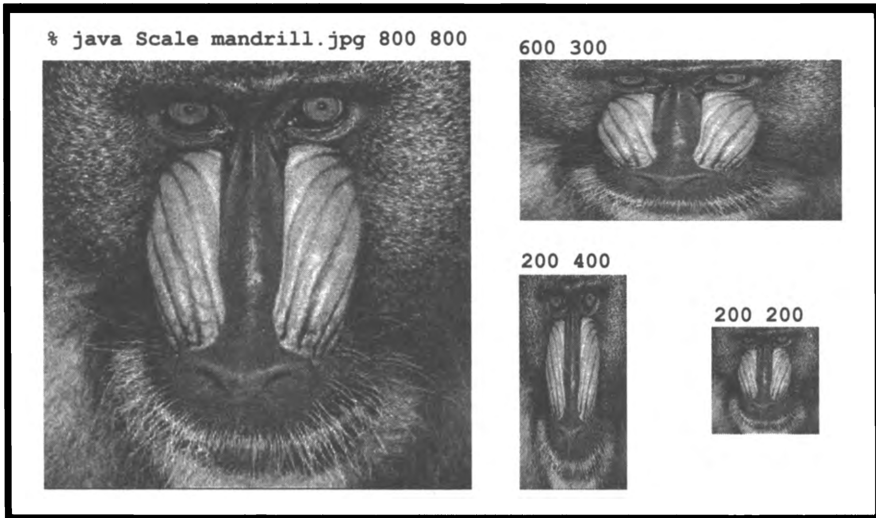
Существует единая стратегия, которая достаточно эффективно работает как для уменьшения, так и для увеличения. Наша цель — получить результирующее изображение, потому мы последовательно перебираем его пикселы и масштабируем координаты каждого пиксела, чтобы определить пиксел исходного изображения, цвет которого может быть присвоен результирующему. Если ширина и высота исходного изображения равны w_s и h_s (соответственно), а ширина и высота результирующего равны w_t и h_t (соответственно), то индекс столбца масштабируется с коэффициентом w_s/w_t, а индекс строки — с коэффициентом h_s/h_t. Иначе говоря, цвет пиксела в столбце c и строке r приемника берется из столбца $c \times w_s/w_t$ и строки $r \times h_s/h_t$ источника. Например, если размер изображения уменьшается вдвое, то масштабные коэффициенты равны 2, так что пикселу в столбце 3 и строке 2 приемника присваивается цвет пиксела в столбце 6 и строке 4 источника; если размер изображения удваивается, то масштабные коэффициенты равны 1/2, и пикселу в столбце 4 и строке 6 приемника присваивается цвет пиксела в столбце 2 и строке 3 источника. Программа `Scale` (листинг 3.1.5) реализует эту стратегию.

Листинг 3.1.5. Масштабирование изображений

```
public class Scale
{
   public static void main(String[] args)
   {
      int w = Integer.parseInt(args[1]);
      int h = Integer.parseInt(args[2]);
      Picture source = new Picture(args[0]);
      Picture target = new Picture(w, h);
      for (int colT = 0; colT < w; colT++)
      {
         for (int rowT = 0; rowT < h; rowT++)
         {
            int colS = colT * source.width() / w;
            int rowS = rowT * source.height() / h;
            target.set(colT, rowT, source.get(colS, rowS));
         }
      }
      source.show();
      target.show();
   }
}
```

`w, h`	размеры приемника
`source`	изображение-источник
`target`	изображение-приемник
`colT, rowT`	координаты пиксела приемника
`colS, rowS`	координаты пиксела источника

Программа получает в аргументах командной строки имя файла, содержащего изображение, и два целых числа (ширина w и высота h). Она масштабирует изображение до размеров w x h и выводит оба изображения.



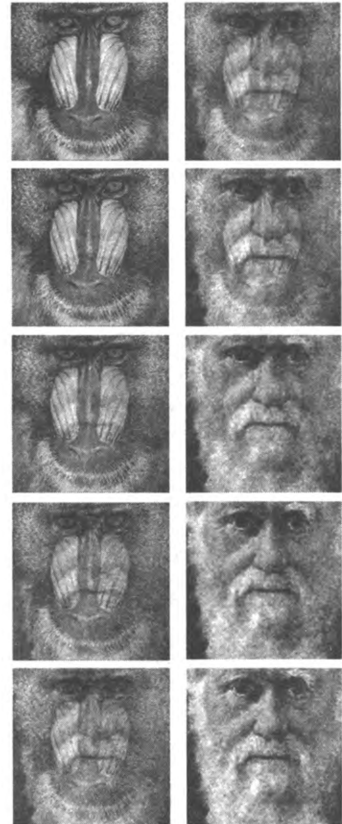
Существуют более сложные методы для изображений низкого разрешения (например, размещенных на старых веб-страницах или полученных со старых камер). Например, при уменьшении изображения вдвое можно усреднять значения четырех пикселов исходного изображения для получения одного результирующего пиксела. Для изображений высокого разрешения, типичных для современных приложений, простой метод, используемый в `Scale`, достаточно эффективен.

Тот же базовый принцип вычисления цвета каждого пиксела приемника как функции цветовых значений некоторых пикселов источника хорошо подходит для всевозможных задач обработки графики. Сейчас мы рассмотрим еще один пример, а в упражнениях и на сайте книги можно найти много других примеров.

Эффект растворения

В нашем последнем примере обработки цифровой графики рассматривается занятное превращение одного изображения в другое серией последовательных шагов. Такие преобразования иногда называют *эффектом растворения*. Программа `Fade` (листинг 3.1.6) является клиентом `Picture` и `Color`, использующим стратегию линейной интерполяции для реализации этого эффекта. Она вычисляет $n - 1$ промежуточных изображений, у которых каждый пиксел изображения i представляет собой взвешенное среднее значений соответствующих пикселов начального и конечного изображений. Статический метод `blend()` реализует интерполяцию: цвету пиксела в начальном изображении назначается весовой коэффициент $1 - i/n$, а в конечном — коэффициент i/n (на шаге $i = 0$ используется цвет начального пиксела, а с $i = n$ — цвет конечного). Это простое вычисление дает впечатляющий результат. Если вы запустите `Fade` на компьютере, изменения словно происходят динамически. Попробуйте запустить программу для некоторых изображений из вашей фотогалереи. Обратите внимание: `Fade` предполагает, что изображения имеют одинаковую ширину и высоту; если для ваших изображений это условие не выполняется, используйте `Scale` для создания масштабированной версии одного (или обоих) из изображений для `Fade`.

```
% java Fade mandrill.jpg darwin.jpg 9
```



Листинг 3.1.6. Эффект растворения

```
import java.awt.Color;
public class Fade
{
   public static Color blend(Color c1, Color c2, double alpha)
   {  // Вычисление взвешенной комбинации цветов c1 и c2.
      double r = (1-alpha)*c1.getRed()   + alpha*c2.getRed();
      double g = (1-alpha)*c1.getGreen() + alpha*c2.getGreen();
      double b = (1-alpha)*c1.getBlue()  + alpha*c2.getBlue();
      return new Color((int) r, (int) g, (int) b);
   }
    public static void main(String[] args)
      { // Отображение последовательности растворения от source
        // к target, состоящей из m промежуточных изображений.
      Picture source = new Picture(args[0]);
      Picture target = new Picture(args[1]);
      int n = Integer.parseInt(args[2]);
      int width  = source.width();
      int height = source.height();
      Picture picture = new Picture(width, height);
      for (int i = 0; i <= n; i++)
      {
         for (int col = 0; col < width; col++)
         {
            for (int row = 0; row < height; row++)
            {
               Color c1 = source.get(col, row);
               Color c2 = target.get(col, row);
               double alpha = (double) i / n;
               Color color = blend(c1, c2, alpha);
               picture.set(col, row, color);
            }
         }
         picture.show();
      }
   }
}
```

`n`	количество изображений
`picture`	текущее изображение
`i`	счетчик изображений
`c1`	цвет источника
`c2`	цвет приемника
`color`	комбинированный цвет

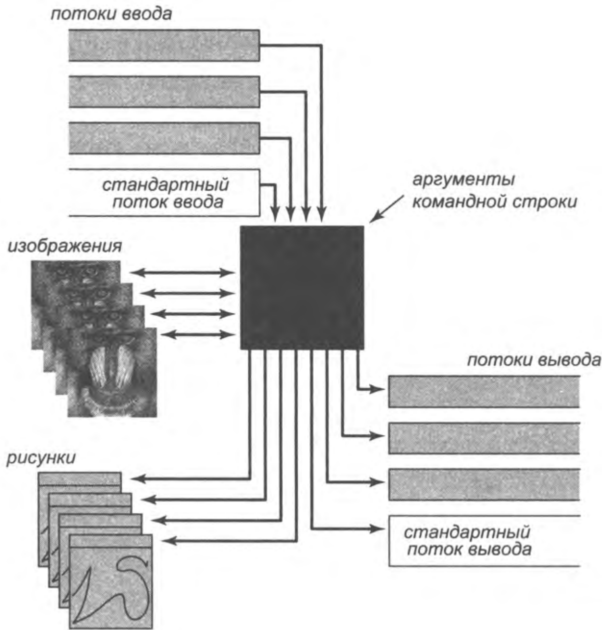
Чтобы перейти от начального изображения к конечному за n шагов, мы присваиваем каждому пикселу промежуточного изображения на шаге i взвешенное среднее соответствующих пикселов начального и конечного изображений; цвет начального пиксела включается с весом 1 – i / n, а цвет конечного — с весом i / n. Пример преобразования представлен на с. 348.

Снова о вводе и выводе

В разделе 1.5 вы научились читать и записывать числа и текст при помощи `StdIn` и `StdOut` и создавать графику средствами `StdDraw`. Разумеется, вы оценили, насколько удобны эти механизмы для получения и выдачи информации в програм-

мах. Одна из причин заключается в том, что «стандартные» соглашения делают их доступными в любом месте программы. Впрочем, есть и недостатки: вы начинаете зависеть от механизмов операционной системы (каналов и перенаправления) при работе с файлами, а в любой конкретной программе вам приходится ограничиваться одним входным файлом, одним выходным файлом и одним векторным рисунком. В объектно-ориентированном программировании можно определить механизмы, которые похожи по своей функциональности на механизмы `StdIn`, `StdOut` и `StdDraw`, но при этом позволяют работать с несколькими потоками ввода, потоками вывода и векторными рисунками в одной программе.

В этом разделе мы определим типы `In`, `Out` и `Draw` для потоков ввода, потоков вывода и для окна векторных изображений соответственно. Как обычно, вы должны обеспечить для Java доступ к этим классам (см. раздел «Вопросы и ответы» в конце раздела 1.5).



Общее представление программы Java (обновленный вариант)

Эти новые типы обеспечивают гибкость, необходимую для решения многих задач обработки данных в программах Java. Вместо того чтобы ограничиваться одним потоком ввода, одним потоком вывода и одним окном векторных изображений, мы можем легко создать несколько объектов каждого типа и связать потоки с разными источниками и приемниками. Также появляются более гибкие возможности для связывания таких объектов с переменными, передачи их в аргументах или

возвращаемых значениях методов, создания массивов и обработки их на тех же принципах, на которых вы работаете с объектами любого другого типа. Некоторые примеры их использования будут представлены после сводки API.

Тип данных — поток ввода

Тип данных `In` представляет собой более общую версию `StdIn` с поддержкой чтения чисел и текста из файлов и с веб-сайтов, а также из стандартного потока ввода. Сводка API приведена ниже. Вместо того чтобы ограничиваться одним абстрактным потоком ввода (стандартный ввод), этот тип дает вам возможность напрямую задать источник потока ввода. Более того, таким источником может быть файл или сайт. Когда вы вызываете конструктор со строковым аргументом, конструктор сначала пытается найти в текущем каталоге локального компьютера файл с заданным именем. Если ему это не удается, конструктор предполагает, что аргумент содержит имя сайта, и пытается связаться с этим сайтом. (Если сайт не существует, генерируется исключение времени выполнения.) В любом случае заданный файл или сайт становится источником данных для созданного объекта потока ввода, а методы `read*()` читают данные из этого потока.

`public class In`

`In()`	Создает объект потока на базе стандартного потока ввода
`In(String name)`	Создает объект потока на базе файла или веб-сайта
Методы экземпляров для чтения лексем из потока ввода	
`boolean isEmpty()`	Поток пуст (или содержит только пропуски)?
`int readInt()`	Читает лексему, преобразует ее в int и возвращает
`double readDouble()`	Читает лексему, преобразует ее в double и возвращает
`...`	
Методы экземпляров для чтения символов из потока ввода	
`boolean hasNextChar()`	В потоке еще остались символы?
`char readChar()`	Читает символ из потока и возвращает его
Методы экземпляров для чтения текстовых строк из потока ввода	
`boolean hasNextLine()`	В потоке еще остались текстовые строки?
`String readLine()`	Читает остаток текстовой строки и возвращает его в формате String
Методы экземпляров для чтения оставшихся данных из потока ввода	
`int[] readAllInts()`	Читает все оставшиеся лексемы, возвращает результат в виде массива int
`double[] readAllDoubles()`	Читает все оставшиеся лексемы; возвращает результат в виде массива double

Примечание: все операции, поддерживаемые StdIn, также поддерживаются и для объектов In.

API нашего типа данных для потоков ввода

Такая схема позволяет работать с несколькими файлами в одной программе. Более того, с возможностью прямого обращения к Сети открывается вся Всемирная паутина, которая становится потенциальным источником ввода для ваших программ. Например, вы можете обрабатывать данные, которые предоставляет и сопровождает кто-то другой. В Интернете доступно очень много таких файлов: ученые регулярно публикуют файлы данных с измерениями или результатами экспериментов, от геномов и белковых цепочек до спутниковых фотографий или астрономических наблюдений; финансовые компании (а также фондовые биржи) регулярно публикуют в Сети подробную информацию о доходности акций и других финансовых инструментов; правительства публикуют результаты выборов и т. д. Теперь вы можете писать на Java программы, которые читают такие файлы напрямую. Тип данных `In` предоставляет большую гибкость в использовании всевозможных доступных источников данных.

Тип данных — поток вывода

Аналогичным образом наш тип данных `Out` представляет собой обобщенную версию `StdOut`, которая поддерживает вывод текста в разные потоки вывода, включая стандартный поток вывода и файлы. Как и в предыдущем случае, в API перечислены те же методы, что и в `StdOut`. Вы задаете файл, который должен использоваться для вывода, при помощи конструктора с одним аргументом, в котором передается имя файла. `Out` интерпретирует эту строку как имя нового файла на локальном компьютере и отправляет вывод в заданный файл. При вызове конструктора без аргументов вы получите стандартный поток вывода.

`public class Out`	
`Out()`	Создает объект потока на базе стандартного потока вывода
`Out(String name)`	Создает объект потока для вывода в файл
`void print(String s)`	Выводит s в поток
`void println(String s)`	Выводит s и символ новой строки в поток
`void println()`	Выводит символ новой строки в поток
`void printf(String format, ...)`	Выводит аргументы в поток в формате, заданном форматной строкой format

API нашего типа данных для потоков вывода

Конкатенация файлов с фильтрацией

В листинге 3.1.7 приведен клиент `In` и `Out`, использующий несколько потоков ввода для конкатенации нескольких входных файлов в один выходной файл. В некоторых операционных системах существует команда `cat`, реализующая эту функцию[1].

[1] `cat` — стандартная команда в Unix-системах. — *Примеч. науч. ред.*

Листинг 3.1.7. Конкатенация файлов

```
public class Cat
{
   public static void main(String[] args)
   {
      Out out = new Out(args[args.length-1]);
      for (int i = 0; i < args.length - 1; i++)
      {
         In in = new In(args[i]);
         String s = in.readAll();
         out.println(s);
      }
   }
}
```

out	поток вывода
i	индекс аргумента
in	текущий поток ввода
s	содержимое in

Программа создает выходной файл, имя которого задается последним аргументом командной строки. Содержимое файла является результатом конкатенации входных файлов, имена которых задаются предыдущими аргументами командной строки.

```
% more in1.txt
This is
% more in2.txt
a tiny
test.
```

```
% java Cat in1.txt in2.txt out.txt
% more out.txt
This is
a tiny
test.
```

Впрочем, программа Java, делающая то же самое, может быть более полезной, потому что ее можно адаптировать для выполнения разнообразной *фильтрации* файлов: исключить ненужную информацию, изменить формат, выбрать только часть данных и т. д. Сейчас мы рассмотрим один из примеров такой обработки; другие примеры можно найти среди упражнений.

Извлечение данных

Сочетанием типов `In` (который позволяет создать поток ввода для любой веб-страницы) и `String` (который предоставляет мощные инструменты для обработки текстовых строк) ваши программы на Java получают прямой доступ ко всей Всемирной паутине без явной зависимости от операционной системы или браузера. Одна из парадигм известна под названием *извлечения данных*: нашей целью является извлечение информации из веб-страниц на программном уровне вместо того, чтобы просматривать их в браузере. Для этого мы воспользуемся тем фактом, что многие веб-страницы являются текстовыми файлами с четко структурированным форматом (потому что они создаются компьютерными программами!). В вашем браузере существуют инструменты, позволяющие просматривать исходный код веб-страницы. Анализ исходного кода часто позволяет получить нужную информацию и понять, как действовать дальше.

Допустим, вы хотите получить биржевое сокращение акций как аргумент командной строки и вывести текущую цену сделки с этими акциями. Такая информация

```
...
(GOOG)</h2> <span class="rtq_
exch"><span class="rtq_dash">-</span>
NMS  </span><span class="wl_sign">
</span></div></div>
<div class="yfi_rt_quote_summary_rt_top
sigfig_promo_1"><div>
<span class="time_rtq_ticker">
<span id="yfs_l84goog">1,100.62</span>
</span> <span class="down_r time_rtq_
content"><span id="yfs_c63_goog">
...
```

Разметка веб-страницы в формате HTML

публикуется в Интернете финансовыми компаниями и поставщиками услуг Интернета. Например, чтобы найти цену акций компании с биржевым сокращением *goog*, нужно обратиться по адресу *http://finance.yahoo.com/q?s=goog*. Как и для многих других веб-страниц, в строке адреса кодируется аргумент (*goog*); вы можете подставить любое другое обозначение для получения веб-страницы с финансовой информацией любой другой компании. Кроме того, как и многие другие данные, публикуемые в Интернете, это текстовый файл, оформленный по правилам языка разметки HTML. С точки зрения программы на Java это просто значение `String`, для работы с которым используется объект `In`. Вы можете загрузить этот файл в браузере или же выполнить команду

```
% java Cat "http://finance.yahoo.com/q?s=goog" goog.html
```

для сохранения данных в файле goog.html на локальном компьютере (хотя реальной необходимости в этом нет). Предположим, цена акций *goog* в настоящее время составляет $1100,62. Если вы проведете поиск строки `"1,100.62"` в тексте страницы, вы найдете цену, зарытую где-то между тегами HTML. Даже не зная всех подробностей HTML, вы сможете кое-что понять о контексте, где находится цена. В данном случае вы видите, что цена заключена между подстроками `<span id="yfs_l84goog">` и `</span>`.

Методы `indexOf()` и `substring()` типа данных `String` позволяют легко извлечь эту информацию, как показано в программе `StockQuote` (листинг 3.1.8). Программа зависит от формата веб-страницы, используемого сервисом *http://finance.yahoo.com*; если этот формат изменится, программа `StockQuote` работать не будет. Собственно, формат может измениться даже к тому моменту, когда вы будете читать эту книгу. Но и в этом случае внести необходимые изменения будет несложно. Поэкспериментируйте с некоторыми интересными доработками `StockQuote`. Например, вы можете получить цену акций за определенные периоды времени и построить ее график, вычислить скользящее среднее или сохранить результаты в файле для последующего анализа. Конечно, этот способ подойдет и для других источников данных в Интернете, как продемонстрируют примеры в упражнениях в конце раздела и на сайте книги.

Листинг 3.1.8. Извлечение данных биржевых котировок

```
public class StockQuote
{
   private static String readHTML(String symbol)
   {  // Получение HTML-данных для определенного вида акций.
      In page = new In("http://finance.yahoo.com/q?s=" + symbol);
      return page.readAll();
   }

   public static double priceOf(String symbol)
   {  // Получение текущей цены.
      String html = readHTML(symbol);
      int p     = html.indexOf("yfs_l84", 0);
      int from  = html.indexOf(">", p);
      int to    = html.indexOf("</span>", from);
      String price = html.substring(from + 1, to);
      return Double.parseDouble(price.replaceAll(",", ""));
   }

   public static void main(String[] args)
   {  // Вывод цены акций.
      String symbol = args[0];
      double price = priceOf(symbol);
      StdOut.println(price);
   }
}
```

symbol	кодовое обозначение акций
page	поток ввода

html	содержимое страницы
p	
from	
to	
price	текущая цена

Программа получает биржевое кодовое обозначение требуемого вида акций как аргумент командной строки и направляет в стандартный поток вывода текущую цену этих акций по данным сайта http://finance.yahoo.com. Для получения данных используются методы indexOf(), substring() и replaceAll() класса String.

```
% java StockQuote goog
1100.62
% java StockQuote adbe
70.51
```

Работа с данными

Возможность использования нескольких потоков ввода и вывода в программе предоставляет более гибкие возможности для решения проблем, связанных с обработкой больших объемов данных из разных источников. Представьте, что ученому или финансовому аналитику приходится поддерживать большой набор данных в электронной таблице. Как правило, электронные таблицы содержат относительно большое количество строк при относительно небольшом количестве столбцов. Может оказаться, что вас интересуют не все данные или лишь некоторые из их столбцов. Электронная таблица позволяет провести с данными некоторые вычисления (в конце концов, для этого она и предназначена), но, конечно, она не обладает такой гибкостью, как программы на Java. Одно из возможных решений заключается в том, чтобы *экспортировать* данные из электронной таблицы в тек-

стовый файл, используя специальный символ для разделения столбцов, а затем написать программу Java, которая читает этот файл из потока ввода. На практике в качестве разделителей часто используются запятые; каждая строка данных выводится в одной строке файла, а данные столбцов разделяются запятыми. Такой формат называется *CSV* (Comma Separated Value)[1]. С помощью метода `String split()` вы можете читать из файла строку за строкой и выделять те данные, которые вам нужны. Примеры такого рода будут приведены позднее в книге. Программа `Split` (листинг 3.1.9) представляет собой клиента `In` и `Out` и идет еще дальше: она создает несколько потоков вывода и отдельный файл для каждого столбца.

Эти примеры убедительно демонстрируют, как удобно работать в Java-программе с текстовыми файлами, несколькими потоками ввода и вывода и прямым доступом к веб-страницам. Веб-страницы имеют формат HTML именно для того, чтобы они были доступны для любой программы, способной читать строковые данные. Текстовые форматы (такие, как .csv) используются вместо закрытых форматов, созданных специально для конкретного приложения, именно для того, чтобы как можно больше людей могло работать с ними с помощью простых программ, таких как `Split`.

Тип данных — векторное изображение

С помощью типа данных `Picture`, представленного ранее в этом разделе, мы можем писать программы, работающие с несколькими изображениями, массивами изображений и т. д., как раз потому, что этот тип данных предоставляет нам средства для работы с объектами `Picture`. Естественно, нам хотелось бы пользоваться такими же возможностями при работе с геометрическими объектами, создаваемыми `StdDraw`. Для этого определим тип данных `Draw` со следующим API:

```
public class Draw
          Draw()
```

Методы для вывода изображения

```
void line(double x0, double y0, double x1, double y1)
void point(double x, double y)
void circle(double x, double y, double radius)
void filledCircle(double x, double y, double radius)
     ...
```

Управляющие методы

```
void setXscale(double x0, double x1)
void setYscale(double y0, double y1)
void setPenRadius(double radius)
     ...
```

Примечание: все операции, поддерживаемые StdDraw, также поддерживаются и для объектов Draw.

[1] На практике так же обычно называют все аналогичные форматы с другими символами-разделителями. — *Примеч. науч. ред.*

Листинг 3.1.9. Разбиение данных

```
public class Split
{
   public static void main(String[] args)
   {  // Разбиение файла по столбцам на n файлов.
      String name = args[0];
      int n = Integer.parseInt(args[1]);
      String delimiter = ",";

      // Создание потоков вывода.
      Out[] out = new Out[n];
      for (int i = 0; i < n; i++)
         out[i] = new Out(name + i + ".txt");

      In in = new In(name + ".csv");
      while (in.hasNextLine())
      {  // Чтение строки и перенаправление
         // содержимого полей в потоки вывода.
         String line = in.readLine();
         String[] fields = line.split(delimiter);
         for (int i = 0; i < n; i++)
            out[i].println(fields[i]);
      }
   }
}
```

`name`	имя исходного файла
`n`	количество полей
`delimiter`	разделитель (запятая)
`in`	поток ввода
`out[]`	потоки вывода
`line`	текущая строка
`fields[]`	значения в текущей строке

Программа использует потоки вывода для разбиения исходного файла в формате .csv на несколько файлов, по одному для каждого столбца тблицы. Имя выходного файла, соответствующего i-му полю, образуется добавлением i и .csv к имени исходного файла.

```
% more DJIA.csv
...
31-Oct-29,264.97,7150000,273.51
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
24-Oct-29,305.85,12900000,299.47
23-Oct-29,326.51,6370000,305.85
22-Oct-29,322.03,4130000,326.51
21-Oct-29,323.87,6090000,320.91
...
```

```
% java Split DJIA 4
% more DJIA2.txt
...
7150000
10730000
16410000
9210000
5920000
12900000
6370000
4130000
6090000
...
```

Новое изображение создается так же, как и объекты других типов: выполните `new` для создания объекта `Draw`, присвойте результат переменной, а затем используйте эту переменную для вызова методов. Например, фрагмент

```
Draw draw = new Draw();
draw.circle(0.5, 0.5, 0.2);
```

выводит окружность в центре окна на экране. Как и в случае с растровыми изображениями Picture, каждое векторное изображение имеет собственное окно, поэтому ваше приложение может выводить на экран сразу несколько разных изображений.

Особенности ссылочных типов

Итак, мы рассмотрели несколько примеров ссылочных типов (Charge, Color, Picture, String, In, Out и Draw) и клиентских программ, в которых эти типы используются. Пора более подробно изучить их важнейшие свойства. Язык Java в значительной степени избавляет новичков от необходимости вникать во все технические подробности. Тем не менее опытные программисты знают, что хорошее понимание основ помогает в написании правильных и эффективных объектно-ориентированных программ.

Концепция ссылок отражает различие между предметом и его именем. Ниже приведены некоторые примеры такого рода.

Тип	Типичный объект	Типичное имя
Веб-сайт	Сайт нашей книги	http://introcs.cs.princeton.edu
Человек	Основоположник информатики	Алан Тьюринг
Планета	Третья планета от Солнца	Земля
Здание	Наш офис	35, Олден-стрит
Корабль	Океанский лайнер, затонувший в 1912 году	Титаник
Число	Отношение длины окружности к ее диаметру	π
Picture	new Picture("mandrill.jpg")	picture

Объект может иметь несколько имен, но содержимое у него всегда одно, это его «идентичность». Вы можете создать новое имя для объекта без изменения его значения (операцией присваивания), но при изменении значения объекта (вызовом метода экземпляра) все имена объекта будут указывать на измененный объект.

Следующая аналогия поможет вам понять суть этого принципиального отличия. Представьте, что вы хотите покрасить свой дом; вы записываете его адрес карандашом на листках бумаги и отдаете нескольким малярам. Если теперь вы наймете одного из маляров, цвет дома изменится. Листок с адресом при этом остается прежним, но дом, на который указывают все эти адреса, уже стал другим. Один из маляров может стереть надпись и записать на листке адрес другого дома, но это никак не повлияет на то, что написано

© 2015 C. Herscovici / Artists Rights Society (ARS), New York

Картина с трубкой

на других листках. Ссылки Java напоминают листки бумаги: они представляют имена объектов. Изменение ссылки не приводит к изменению объекта, но изменение объекта немедленно становится видимым каждому, кто располагает ссылкой на такой объект[1].

Знаменитый бельгийский художник Рене Магритт отразил эту концепцию в своей картине, на которой была изображена трубка с подписью *ceci n'est pas une pipe* («это не трубка»). Надпись можно интерпретировать как утверждение о том, что рисунок является не трубкой, а всего лишь изображением трубки. А может быть, Магритт имел в виду, что подпись — это не трубка и не изображение трубки, а всего лишь подпись! В нашем контексте эта картина иллюстрирует идею о том, что ссылка на объект — это всего лишь ссылка, а не сам объект.

Совмещение имен

Операция присваивания для ссылочного типа создает второй экземпляр ссылки. Она *создает не новый объект*, а всего лишь еще одну ссылку на уже существующий объект. Такая ситуация называется ***совмещением имен***: обе переменные указывают на один объект. Эффект совмещения имен может показаться несколько неожиданным, потому что он отличается от ситуации с хранением в переменных значений примитивных типов. *Обязательно убедитесь в том, что вы понимаете суть различий.* Если `x` и `y` — переменные примитивных типов, то команда присваивания `x = y` копирует значение `y` в `x`. Для ссылочных типов же копируется значение ссылки, но не значение самого объекта.

Совмещение имен часто становится источником ошибок в программах Java, как видно из следующего примера:

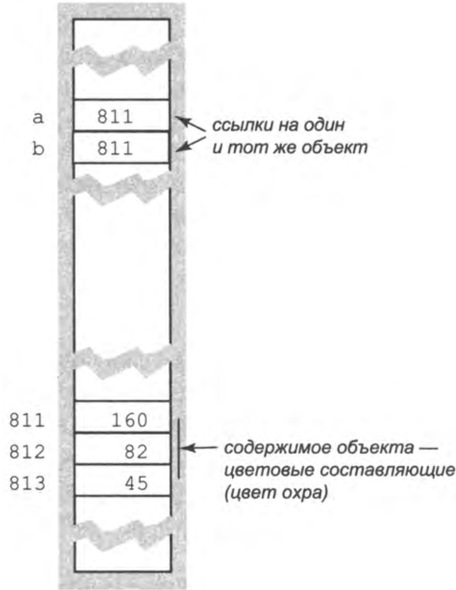
```
Picture a = new Picture("mandrill.jpg");
Picture b = a;
a.set(col, row, color1);  // Обновляется
b.set(col, row, color2);  // Снова обновляется
```

После второй команды присваивания обе переменные `a` и `b` ссылаются на один и тот же объект `Picture`. Изменение состояния объекта распространяется на весь код, в котором используются переменные, ссылающиеся на этот объект. Мы привыкли к тому, что две разные переменные примитивных типов не зависят друг от друга, но это интуитивное представление не переносится на ссылочные объекты. Например, если программист будет считать, что в приведенном коде `a` и `b` ссылаются на разные объекты `Picture`, программа будет работать неверно. Такие ошибки часто встречаются в программах, написанных людьми без опыта работы со ссылочными объектами (и пока вы тоже относитесь к их числу, так что будьте внимательны!).

---

[1] Достаточно близкими аналогиями этой концепции могут служить, например, множественные имена файлов в файловых системах Unix или дескрипторы открытых файлов и других программных объектов (handle в Windows). — *Примеч. науч. ред.*

```
Color a:
a = new Color(160, 82, 45):
Color b = a:
```



Наложение

Неизменяемые типы

Именно по этой причине в программах часто определяются типы данных, значения которых не могут изменяться. Объект называется *неизменяемым*, если его значение не может быть изменено после создания. *Неизменяемым типом данных* называется тип, все объекты которого являются неизменяемыми. Например, тип данных `String` является неизменяемым, потому что клиенту недоступны операции, изменяющие символы строки. Напротив, у *изменяемого* типа данных объекты содержат значения, которые могут изменяться. Например, тип данных `Picture` является изменяемым, потому что мы можем изменять цвета пикселов. Неизменяемость будет более подробно рассмотрена в разделе 3.3.

Сравнение объектов

Оператор `==`, примененный к ссылочным типам, проверяет две *ссылки на объект* (то есть указывают ли они на один и тот же объект). Это не то же самое, что проверка равенства значений объектов. Для примера возьмем следующий код:

```
Color a = new Color(160, 82, 45);
Color b = new Color(160, 82, 45);
Color c = b;
```

Условие `(a == b)` ложно, а условие `(b == c)` истинно. Но если речь заходит о проверке равенства объектов `Color`, то, вероятно, вы подразумеваете проверку равенства значений. В Java не существует автоматического механизма проверки равенства значений объектов, так что программист может (и даже обязан!) определить ее самостоятельно. Для этого в классе определяется метод с именем `equals()`, как описано в разделе 3.3[1]. Например, в классе `Color` такой метод существует, и выражение `a.equals(c)` в нашем примере истинно. Класс `String` также содержит реализацию `equals()`, потому что в программе часто бывает нужно проверить, содержат ли два объекта `String` одинаковые значения (одинаковые последовательности символов).

Передача по значению

При вызове метода с аргументами в Java все происходит так, словно сами аргументы находятся в правой части команды присваивания, а соответствующие имена аргументов — в левой. Иначе говоря, Java передает методу *копию* значения аргумента. Если значение аргумента имеет примитивный тип, то Java передает копию этого значения; если значение аргумента является ссылкой на объект, то Java передает копию ссылки на объект. Такая схема передачи называется *передачей по значению*.

Одно важное последствие такого механизма заключается в том, что метод не может напрямую изменять значение переменной на стороне вызова. Для примитивных типов эта схема полностью соответствует нашим ожиданиям (две переменные существуют независимо); однако каждый раз, когда в аргументе метода передается ссылочный тип, вы тем самым создаете новое «совмещенное» имя (синоним имени), так что будьте осторожны. Например, если передать методу ссылку на тип `Picture`, метод не сможет изменить саму ссылку на объект (например, «перевести» ее на другой объект `Picture`), но сможет изменить *значение* объекта — скажем, вызвать метод `set()` для изменения цвета пиксела[2].

Массивы как объекты

В языке Java любое значение непримитивного типа является объектом. В частности, массивы тоже являются объектами. Как и в случае со строками, для некоторых операций с массивами — объявления, инициализации и индексирования — предусмотрена специальная языковая поддержка. Как и для любого другого объекта, при передаче массива методу или при использовании переменной-массива в правой части операции присваивания создается копия ссылки на этот массив, а не копия

---

[1] Другой подход к решению этой проблемы (например, в C++) — *перегрузка* операторов, в данном случае оператора сравнения `==`. — *Примеч. науч. ред.*

[2] Таким образом, если сама ссылка передается по значению, то объект, на который она ссылается, — фактически *по ссылке*. Поэтому такой подход к передаче аргументов-объектов часто называют *передачей по ссылке*, имея в виду косвенную передачу объекта, а точнее, доступа к нему посредством ссылки. По сути, это разновидность *передачи по указателю* в традиционных языках, например в C. — *Примеч. науч. ред.*

самого массива. Массивы являются изменяемыми объектами — это правило уместно для типичной ситуации, когда метод должен изменить содержимое массива — допустим, переставить его элементы в другом порядке, как методы `exchange()` и `shuffle()` из раздела 2.1.

Массивы объектов

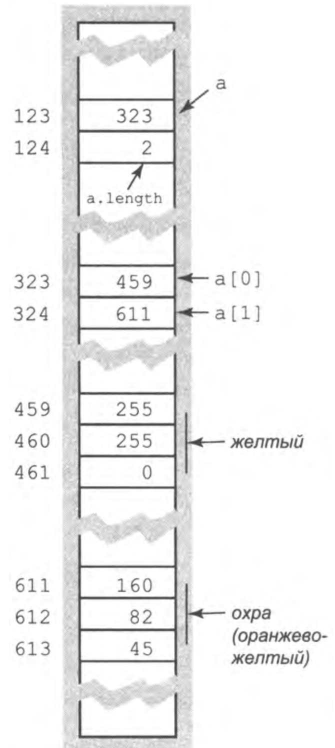
Массив может содержать элементы любого типа. Примеры такого рода вам уже встречались — от `args[]` (массив строк) в наших реализациях `main()` до массива объектов `Out` в программе 3.1.9. Создание массива объектов проходит в два этапа:

- создание массива ключевым словом `new` и квадратными скобками;
- создание каждого элемента массива с использованием `new` для вызова соответствующего конструктора.

Например, следующий фрагмент создает массив с двумя объектами `Color`:

```
Color[] a = new Color[2];
a[0] = new Color(255, 255, 0);
a[1] = new Color(160, 82, 45);
```

Естественно, элементами массива объектов в Java являются ссылки на объекты, а не сами объекты. Для больших объектов это повышает эффективность, так как вам не нужно перемещать в памяти сами объекты, а только ссылки на них. Если же объекты малы, то эффективность снижается, так как каждый раз, когда потребуется получить информацию, программе приходится совершать лишний косвенный переход по ссылке.



Массив объектов

Безопасные указатели

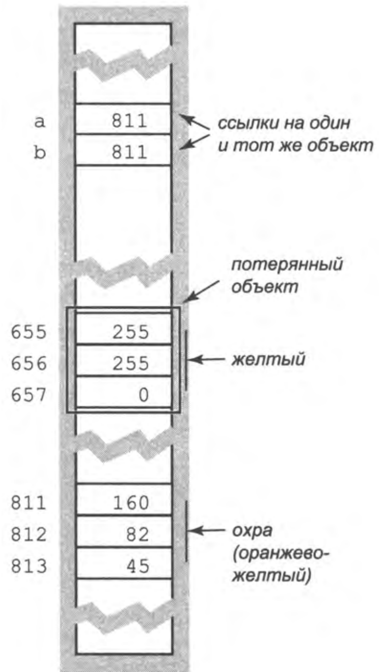
Для работы непосредственно с адресами памяти, по которым хранятся данные, во многих языках программирования присутствуют *указатели* (похожие на ссылки Java) как примитивный тип данных. Программирование с использованием указателей пользуется дурной славой из-за высокого риска ошибок, поэтому любые операции с указателями должны быть тщательно продуманы для предотвращения ошибок. В Java этот подход доведен до логического завершения (как, впрочем, и во многих других современных языках программирования). В Java существует только один способ создания ссылок (ключевым словом `new`) и только один способ работы

с этими ссылками (операция присваивания). Другими словами, единственное, что может делать программист со ссылками, — создавать и копировать их. На жаргоне современного программирования ссылки Java называют *безопасными указателями*, потому что Java гарантирует, что каждая ссылка указывает на объект заданного типа, а не на произвольный адрес памяти. Некоторые программисты, привыкшие к прямым манипуляциям с указателями, считают, что в Java указателей нет вообще, но вопрос о полезности небезопасных указателей до сих пор остается открытым. Короче говоря, в программировании на Java вы не будете напрямую работать с адресами памяти. А если в будущем такая возможность вам встретится в каком-нибудь из языков — будьте осторожны!

Потерянные объекты

Возможность присваивания разных объектов ссылочной переменной создает вероятность того, что в программе будет создан объект, на который не существует ни одной ссылки. Для примера возьмем три команды присваивания на иллюстрации. После выполнения третьей команды присваивания переменные a и b указывают на один и тот же объект Color (с составляющими RGB 160, 82 и 45), а на объект Color, который был создан и использован для инициализации b, не существует ни одной ссылки. Единственная ссылка на этот объект существовала в операции присваивания, и снова обратиться к этому объекту уже невозможно. Такие объекты называются *потерянными*. Объекты также теряются при выходе из области видимости. Программисты Java могут не беспокоиться по поводу «бесхозных» объектов, потому что система автоматически освобождает выделенную для них память.

```
Color a, b;
a = new Color(160, 82, 45);
b = new Color(255, 255, 0);
b = a;
```



Потерянный объект

Управление памятью

В программах часто создаются многие сотни объектов, но в любой момент времени программа работает лишь с небольшой их частью. Соответственно в языках и системах программирования необходимы механизмы выделения памяти для объектов на то время, пока они нужны, и механизмы освобождения памяти, которая больше не используется (в какой-то момент после того, как объект, занимавший эту память, стал потерянным). С примитивными типами управление памятью упрощается, поскольку вся информация, необходимая для ее выделения, известна на стадии компиляции.

В Java (и большинстве других систем) память для переменных резервируется при объявлении и освобождается при выходе их из области видимости. С объектами дело обстоит сложнее: Java знает, что память для объекта должна выделяться при создании (`new`), но не может определить, когда нужно освободить память, связанную с объектом, потому что этот момент зависит от динамики выполняемой программы.

Утечка памяти

Во многих языках (например, в C и C++) программист сам отвечает за выделение и освобождение памяти. Это рутинная работа, часто приводящая к ошибкам. Представьте, что программа освободила память объекта, после чего снова обращается к нему (возможно, намного позже). За прошедшее время система уже могла выделить ту же память для других целей; понятно, что это может привести к непредсказуемым последствиям. Другая коварная ошибка возникает тогда, когда программист забывает освободить память неиспользуемого объекта. Ошибки такого рода приводят к *утечке памяти*: объем памяти, задействованной для потерянных объектов, неуклонно растет (а такая память больше не может использоваться для других целей). В результате производительность снижается, словно память постепенно «вытекает» из компьютера. Вам когда-нибудь приходилось перезагружать компьютер из-за того, что он все медленнее реагировал на ваши действия? Чаще всего это происходит из-за утечки памяти в одном из приложений.

Сбор мусора

Одна из самых важных функций Java — автоматическое управление памятью. Идея заключается в том, чтобы снять с программиста ответственность за управление памятью: система сама выявляет неиспользуемые объекты и возвращает занимаемую ими память в пул свободной памяти. Такой механизм освобождения памяти называется *сбором мусора*, а политика безопасных указателей Java открывает путь к его автоматической и эффективной реализации. Программисты продолжают спорить, что важнее: затраты ресурсов на автоматический сбор мусора или удобство от того, что программисту не нужно беспокоиться об управлении памятью. Мы можем сказать лишь то, что говорилось ранее об указателях: когда вы пишете программу на Java, вам не придется писать код выделения и освобождения памяти[1], но если вам придется заниматься этим в будущем в другом языке программирования — будьте осторожны!

Для удобства мы приводим таблицу с краткой сводкой примеров классов, упоминающихся в этом разделе. Эти примеры помогут вам разобраться в важнейших свойствах типов данных и в принципах объектно-ориентированного программирования в целом.

---

[1] Операция new — явное создание объекта — как раз и сопровождается выделением памяти для него. Здесь имеется в виду, что не приходится *сначала* выделять память и *затем* размещать в ней объект *отдельными* операциями. — *Примеч. науч. ред.*

API	описание
`Color`	Цвета
`Picture`	Цифровые изображения
`String`	Символьные строки
`In`	Потоки ввода
`Out`	Потоки вывода
`Draw`	Векторные рисунки

Сводка примеров классов, упоминающихся в разделе

Тип данных представляет собой множество значений и множество операций, определенных для этих значений. Для примитивных типов данных вы работаете с небольшими и относительно простыми значениями. Строки, цвета, векторные рисунки, потоки ввода/вывода относятся к высокоуровневым типам данных, свидетельствующим о широком спектре применения абстракций данных. *Чтобы использовать тип данных, вам не обязательно знать, как он реализован.* Каждый тип данных (а в библиотеках Java их сотни, к тому же вы вскоре научитесь создавать собственные типы) характеризуется своим программным интерфейсом (API), который предоставляет информацию, необходимую для его использования. Клиентская программа создает объекты, которые содержат значения этого типа, и вызывает методы экземпляров для выполнения операций с этими значениями. Для написания клиентских программ используются основные команды и управляющие конструкции, упоминавшиеся в главах 1 и 2, но теперь вы можете использовать куда более разнообразные типы данных, а не только уже привычные примитивные. С появлением практического опыта вы увидите, что эта возможность открывает новые горизонты в программировании.

С правильно спроектированными типами данных клиентские программы получаются более понятными, простыми в разработке и сопровождении, чем программы, где абстракция данных не используется. Клиентские программы в этом разделе подтверждают правоту этого утверждения. Более того, как вы увидите в следующем разделе, реализация типа данных является логичным продолжением тех базовых навыков программирования, которые вы уже усвоили. В частности, при разработке больших и сложных приложений программист стремится понять данные и выполняемые с ними операции, после чего пишет код, в котором напрямую отражено это понимание. А когда вы освоите этот подход, остается только удивляться: как программисты когда-то разрабатывали большие программы без использования абстракции данных?

Вопросы и ответы

В. Для чего в Java различаются примитивные и ссылочные типы?

О. По соображениям быстродействия. Java предоставляет ссылочные типы-обертки `Integer`, `Double` и т. д., которые могут использоваться программистами, предпо-

читающими забыть о различиях (за подробностями обращайтесь к разделу 3.3). Примитивные типы ближе к типам данных, поддерживаемым компьютерным оборудованием, поэтому с ними программы обычно работают быстрее и занимают меньше памяти, чем при использовании соответствующих ссылочных типов.

В. Что произойдет, если я забуду `new` при создании объекта?

О. Для Java все выглядит так, словно вы пытаетесь вызвать статический метод, который возвращает объект. Так как такой метод не определен, сообщение об ошибке выглядит так же, как при обращении к неопределенному символическому имени. При попытке откомпилировать код

```
Color sienna = Color(160, 82, 45);
```

будет получено следующее сообщение:

```
cannot find symbol
symbol  : method Color(int,int,int)
```

Конструкторы не возвращают значения (их сигнатура не имеет возвращаемого типа) — они могут только следовать за `new`. Такое же сообщение об ошибке будет получено при передаче неправильного числа аргументов конструктору или другому методу.

В. Почему мы можем вывести объект `x` вызовом функции `StdOut.println(x)` вместо `StdOut.println(x.toString())`?

О. Хороший вопрос. Вторая конструкция более правильная, но Java экономит нам немного времени, вызывая метод `toString()` в подобных ситуациях автоматически. В разделе 3.3 будет рассмотрен механизм Java, который обеспечивает эту возможность.

В. Чем отличаются `=`, `==` и `equals()`?

О. Один знак равенства (`=`) обозначает команду присваивания — конечно, вы это уже знаете. Двойной знак равенства (`==`) обозначает логический оператор для проверки равенства двух операндов. Если операнды относятся к примитивному типу, то результат равен `true`, когда значения операндов равны, и `false` в противном случае. Если же операнды являются ссылками на объекты, то результат равен `true`, когда они указывают на один и тот же объект, или `false` в противном случае. Другими словами, оператор `==` проверяет не просто равенство, а совпадение объектов. Метод `equals()` включается в каждый тип Java для проверки двух объектов на равенство значений (в том числе и у двух независимых экземпляров). Обратите внимание: `(a == b)` подразумевает `a.equals(b)`, но не наоборот.

В. Как организовать передачу массива в аргументе функции так, чтобы функция не могла изменять значения элементов массива?

О. Простого решения не существует — массивы изменяемы. В разделе 3.3 вы увидите, как добиться того же эффекта, построив тип-обертку и передавая ссылку на этот тип (см. `Vector` в листинге 3.3.3).

В. Что произойдет, если я забуду использовать new при создании массива объектов?

О. Вы должны использовать new для каждого создаваемого вами объекта, поэтому при создании массива из *n* объектов ключевое слово new должно встречаться *n* + 1 раз: один раз для массива и по одному разу для каждого объекта. Если вы забудете создать массив:

```
Color[] colors;
colors[0] = new Color(255, 0, 0);
```

то получите сообщение об ошибке (попытка присваивания неинициализированной переменной):

```
variable colors might not have been initialized
      colors[0] = new Color(255, 0, 0);
      ^
```

С другой стороны, если вы забудете использовать new при создании объекта в массиве, а потом попробуете использовать его для вызова метода:

```
Color[] colors = new Color[2];
int red = colors[0].getRed();
```

произойдет исключение NullPointerException. Как обычно, лучший способ получить ответ на такой вопрос — написать и откомпилировать код, а затем попытаться интерпретировать сообщение об ошибке Java. Это поможет вам быстрее находить такие ошибки в будущем.

В. Где можно найти более подробную информацию о том, как в Java реализованы ссылки и механизм сбора мусора?

О. Java-системы могут полностью отличаться друг от друга. Например, могут использоваться как указатели (машинные адреса), так и дескрипторы (указатели на указатели). Первая схема характеризуется более быстрым доступом к данным, а вторая упрощает уборку мусора.

В. Почему красный, зеленый и синий, а не красный, желтый и синий?

О. Теоретически подойдут любые три цвета, содержащие некоторую долю всех первичных цветов, но на практике наибольшее распространение получили две цветовые модели: одна (RGB) обеспечивала хорошую цветопередачу на экранах телевизоров, компьютерных мониторах и цифровых камерах, а другая (CMYK) обычно используется для печати (см. упражнение 1.2.32). В модели CMYK желтый цвет присутствует (голубой, малиновый, желтый и черный). Существование двух разных цветовых моделей объясняется тем, что краска на бумаге *поглощает* цвет; то есть при использовании двух разных красок поглощается *больше* света, а отражается *меньше*. И наоборот, на экранах пикселы *светятся*, поэтому с двумя пикселами разного цвета света излучается *больше*.

В. Что именно делает команда import?

О. Не так уж много: просто избавляет вас от необходимости вводить лишние символы. Например, в программе 3.1.2 она позволяет сократить запись `java.awt.Color` до `Color` в любой точке вашего кода.

В. Создание и уничтожение тысяч объектов `Color`, как в программе `Grayscale` (листинг 3.1.4), — это нормально?

О. Любая конструкция языка программирования сопряжена с некоторыми затратами. В данном случае затраты разумны, потому что время создания объектов `Color` мало по сравнению с временем рисования изображения.

В. Почему вызов метода `s.substring(i, j)` класса `String` возвращает подстроку `s`, которая начинается с индекса `i` и завершается в позиции `j-1` (а не `j`)?

О. Почему индексы массива `a[]` лежат в диапазоне от 0 до `a.length-1` вместо диапазона от 1 до `length`? Проектировщики языков программирования принимают решения; мы живем с этими решениями. У такой схемы индексирования есть одна полезная особенность: длина извлекаемой подстроки равна `j-i`.

В. Чем *передача по значению* отличается от *передачи по ссылке*?

О. Когда используется передача по значению, то при вызове метода значения аргументов вычисляются и методу передаются их копии. Это означает, что если метод напрямую изменит переменную-аргумент, на стороне вызова это изменение не видно. При передаче же по ссылке методу передаются адреса памяти, где размещаются все аргументы. Таким образом, если метод изменит переменную-аргумент, внесенное изменение отразится и на стороне вызова. Формально в языке Java используется только передача по значению, но при этом значение может быть как значением примитивного типа, так и ссылкой на объект. В результате при передаче методу значения примитивного типа метод не может изменить соответствующее значение на стороне вызова; при передаче методу ссылки на объект метод не может изменить ссылку (допустим, заставить ее ссылаться на другой объект), но может изменить объект, на который эта ссылка указывает (используя ссылку для вызова одного из методов объекта).

В. Я заметил, что аргумент метода `equals()` в классах `String` и `Color` относится к типу `Object`. Разве аргумент не должен относиться к типу `String` и `Color` соответственно?

О. Нет. В Java метод `equals()` занимает особое место, а типом его аргумента всегда должен быть `Object`. Это пример действия механизма наследования, применяемого в Java при поддержке метода `equals()`; мы рассмотрим этот метод позднее. А пока на этот нюанс можно просто не обращать внимания.

В. Почему мы назвали тип данных для работы с изображениями `Picture`, а не `Image`?

О. В Java уже существует встроенная библиотека с именем `Image`.

Упражнения

3.1.1. Напишите программу, которая получает значение `w` типа `double` в командной строке, создает четыре объекта `Charge` с величиной заряда 1.0, которые располагаются на расстоянии `w` в каждом из четырех основных направлений от точки (0,5, 0,5), и выводит величину потенциала в точке (0,25, 0,5).

3.1.2. Напишите программу, которая получает в командной строке три целых числа от 0 до 255, представляющих красную, зеленую и синюю составляющие цвета, после чего создает и выводит объект `Picture` для изображения 256 × 256 пикселов, в котором каждый пиксел окрашен в этот цвет.

3.1.3. Измените программу `AlbersSquares` (листинг 3.1.2), чтобы она получала в командной строке *девять* аргументов, определяющих *три* цвета, а затем рисовала все шесть возможных квадратов Альберса: большой квадрат закрашивается одним цветом, а маленький внутренний квадрат — другим.

3.1.4. Напишите программу, которая получает имя файла с изображением в оттенках серого как аргумент командной строки и использует `StdDraw` для построения гистограммы частот каждого из 256 оттенков серого.

3.1.5. Напишите программу, которая получает имя графического файла в командной строке и зеркально переворачивает изображение по горизонтали.

3.1.6. Напишите программу, которая получает имя графического файла в командной строке, создает и выводит три объекта `Picture`: один содержит только красные составляющие цветов, второй зеленые, а третий синие.

3.1.7. Напишите программу, которая получает имя графического файла в аргументе командной строки и выводит координаты левого нижнего и правого верхнего угла наименьшего обрамляющего прямоугольника (прямоугольник, стороны которого параллельны осям x и y), содержащего все не белые пикселы.

3.1.8. Напишите программу, которая получает как аргументы командной строки имя графического файла и координаты (в пикселах) прямоугольника в этом изображении; читает из стандартного ввода список значений `Color` (представленных тройками значений `int`) и работает как фильтр, выводя значения цветов, для которых все пикселы в прямоугольнике обладают совместимостью фонового/основного цвета. (Например, такой фильтр может использоваться для выбора цвета текста, которым будет помечаться изображение.)

3.1.9. Напишите статический метод `isValidDNA()`, который получает в качестве аргумента строку и возвращает `true` в том и только в том случае, если строка состоит только из символов `A`, `T`, `C` и `G`.

3.1.10. Напишите функцию `complementWatsonCrick()`, которая получает в аргументе цепочку ДНК и возвращает ее *дополнение Уотсона — Крика*: `A` заменяется на `T`, `C` заменяется на `G` и наоборот.

3.1.11. Напишите функцию `isWatsonCrickPalindrome()`, которая получает строку ДНК и возвращает `true`, если строка является палиндромом в отношении дополнения Уотсона — Крика, то есть если она равна своему дополнению Уотсона — Крика, записанному в обратном порядке.

3.1.12. Напишите программу, которая проверяет код ISBN на правильность (см. упражнение 1.3.35) с учетом того факта, что код ISBN может содержать дефисы в произвольных местах.

3.1.13. Что делает следующий фрагмент кода?

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
StdOut.println(string1);
StdOut.println(string2);
```

3.1.14. Какой результат выведет следующий фрагмент?

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

Ответ: *«Hello World»*. Объекты `String` являются неизменяемыми — все методы `String` возвращают новый объект `String` с нужным значением (но не изменяют значения того объекта, который использовался как аргумент при их вызове). Код игнорирует возвращенные объекты и просто выводит исходную строку. Чтобы фрагмент выводил строку `"WORLD"`, замените вторую и третью строки на `s = s.toUpperCase()` и `s = s.substring(6, 11)`.

3.1.15. Строка `s` называется *циклическим сдвигом* строки `t`, если она может быть получена циклическим сдвигом символов этой строки на некоторое количество позиций. Например, строка `ACTGACG` является циклическим сдвигом строки `TGACGAC`, и наоборот. Проверка этого условия играет важную роль в изучении геномов. На пишите программу, которая проверяет, являются ли две строки циклическими сдвигами друг друга. *Подсказка*: решение записывается в одну строку с `indexOf()` и конкатенацией.

3.1.16. Для заданной строки, представляющей доменное имя, напишите фрагмент кода для выделения домена верхнего уровня. Например, в строке *cs.princeton.edu* доменом верхнего уровня является *edu*.

3.1.17. Напишите статический метод, который получает доменное имя как аргумент и возвращает *обратное доменное имя* (строковые компоненты между точками переставляются в обратном порядке). Например, для *cs.princeton.edu* обратное доменное имя имеет вид *edu.princeton.cs*. Эта операция важна для анализа веб-журналов (см. упражнение 4.2.36).

3.1.18. Что делает следующая рекурсивная функция?

```
public static String mystery(String s)
{
   int n = s.length();
   if (n <= 1) return s;
   String a = s.substring(0, n/2);
   String b = s.substring(n/2, n);
   return mystery(b) + mystery(a);
}
```

3.1.19. Напишите тестовое приложение-клиент для программы `PotentialGene` (листинг 3.1.1). Клиент должен получать строку как аргумент командной строки и сообщать, описывает ли полученная последовательность потенциальный ген.

3.1.20. Напишите версию программы `PotentialGene` (листинг 3.1.1), которая находит все потенциальные гены, содержащиеся в виде подпоследовательностей длинной цепочки ДНК. Добавьте аргумент командной строки, чтобы пользователь мог задать минимальную длину потенциального гена.

3.1.21. Напишите фильтр, который читает текст из потока ввода и направляет его в поток вывода, удаляя все строки, состоящие только из пробельных символов (пробелы, табуляции и т. д.).

3.1.22. Напишите программу, которая получает начальную и конечную последовательности как аргументы командной строки и выводит все подпоследовательности заданной строки, которые начинаются с начальной последовательности, завершаются конечной последовательностью и не содержат их внутренних вхождений. *Примечание*: будьте особенно внимательны с перекрытиями!

3.1.23. Измените программу `StockQuote` (листинг 3.1.8), чтобы она получала несколько биржевых кодовых обозначений в командной строке.

3.1.24. В файле DJIA.csv, который использовался в программе `Split` (листинг 3.1.9), указывается дата, верхнее значение, объем сделок и нижнее значение биржевого индекса Доу-Джонса за каждый день с момента сохранения данных. Загрузите файл с сайта книги и напишите программу, которая создает два объекта `Draw`: для цен и для объемов сделок — и отображает данные с периодичностью, заданной в командной строке.

3.1.25. Напишите программу `Merge`, которая получает как аргументы командной строки разделитель и следующее за ним произвольное количество имен файлов; выполняет конкатенацию соответствующих строк всех файлов, вставляя между ними заданный разделитель, а потом выводит результат в стандартный вывод, фактически выполняя операцию, обратную по отношению к `Split` (листинг 3.19).

3.1.26. Найдите сайт, который публикует метеорологические прогнозы для вашего региона. Напишите программу `Weather`, чтобы команда `java Weather`, за которой следует ваш почтовый индекс, выдавала прогноз погоды.

3.1.27. Предположим, `a[]` и `b[]` — целочисленные массивы, состоящие из миллионов целых чисел. Что делает следующий фрагмент, как долго он выполняется?

```
int[] temp = a; a = b; b = temp;
```

Решение. Фрагмент меняет местами массивы, но для этого он копирует ссылки на объекты, избегая копирования миллионов значений.

3.1.28. Опишите результат выполнения следующей функции.

```
public void swap(Color a, Color b)
{
   Color temp = a;
   a = b;
   b = temp;
}
```

Упражнения повышенной сложности

3.1.29. *Формат файлов для Picture*. Напишите библиотеку статических методов `RawPicture` с методами `read()` и `write()` для сохранения и чтения изображений из файла. Метод `write()` получает в качестве аргументов объект `Picture` и имя файла и записывает изображение в заданный файл в следующем формате: если изображение имеет размеры $w \times h$, то метод записывает w, затем h, а затем $w \times h$ троек целых чисел, представляющих значения цветов пикселов (слева направо, сверху вниз). Метод `read()` получает в аргументе имя файла и возвращает объект `Picture`, созданный чтением изображения из заданного файла в только что описанном формате. *Примечание*: учтите, что такой способ хранения расходует значительно больше дискового пространства, чем реально необходимо, — стандартные форматы предусматривают сжатие данных, чтобы они занимали меньше места.

3.1.30. *Визуализация звука*. Напишите программу, использующую `StdAudio` и `Picture` для создания наглядной двумерной цветной визуализации во время воспроизведения звукового файла. Проявите творческую фантазию!

3.1.31. *Шифр Камасутры*. Напишите фильтр `KamasutraCipher`, который получает две строки (ключи) в аргументах командной строки, затем читает строковые данные (разделенные пропусками) из стандартного потока ввода, выполняет замену каждой буквы в соответствии со строками-ключами и выводит результат в стандартный поток вывода. Эта операция лежит в основе одной из древнейших известных криптографических систем. Ключи должны иметь одинаковую длину, а каждая буква в стандартном вводе должна встречаться ровно в одном из них. Например, если заданы ключи `THEQUICKBROWN` и `FXJMPSVRLZYDG`, строится следующая таблица:

```
T H E Q U I C K B R O W N
F X J M P S V L A Z Y D G
```

Это означает, что программа должна заменять F на T, T на F, H на X, X на H и т. д. в процессе фильтрации данных из стандартного потока ввода в стандартный поток

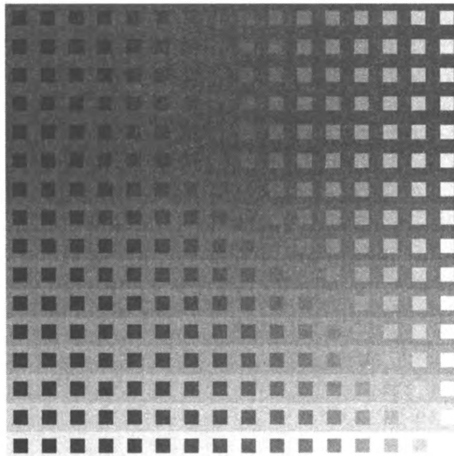
вывода. Сообщение кодируется заменой каждой буквы в паре. Например, сообщение `MEET AT ELEVEN` кодируется в виде `QJJF BF JKJCJG`. Получатель использует те же ключи для того, чтобы восстановить зашифрованное сообщение.

3.1.32. *Проверка надежности пароля.* Напишите статический метод, который получает в качестве аргумента строку и возвращает `true` при выполнении следующих условий или `false` в противном случае:

- длина строки не менее 8 символов;
- строка содержит как минимум одну цифру (0–9);
- строка содержит как минимум одну букву верхнего регистра;
- строка содержит как минимум одну букву нижнего регистра;
- строка содержит как минимум один символ, который не является ни буквой, ни цифрой.

Такие проверки часто используются для паролей в Интернете.

3.1.33. *Образцы цветов.* Напишите программу, которая выводит образцы цветов в виде квадратов Альберса, соответствующих 256 уровням синего (от синего к белому по строкам) и серого (от черного к белому по столбцам).



Образцы цветов

3.1.34. *Энтропия.* Энтропия Шеннона служит показателем информативности входной строки; эта метрика играет важную роль в теории информации и сжатии данных. Для заданной строки из n символов обозначим f_c частоту вхождения символа c. Величина $p_c = f_c/n$ является оценкой вероятности присутствия c в случайной строке; энтропия определяется как сумма величины $-p_c\log_2 p_c$ по всем символам, входящим в строку. Считается, что энтропия оценивает информационное содержание строки: если все символы появляются одинаковое количество раз, то энтропия

для заданной длины строки минимальна. Напишите программу, которая получает имя файла как аргумент командной строки и выводит энтропию текста в этом файле. Запустите программу для веб-страницы, которую вы регулярно читаете, для недавно написанной вами статьи для генома дрозофилы с сайта книги.

3.1.35. *Мозаика.* Напишите программу, которая получает имя файла с изображением и два целых числа m и n в аргументах командной строки и создает мозаику из $m \times n$ копий изображения.

3.1.36. *Фильтр поворота.* Напишите программу, которая получает два аргумента командной строки (имя файла с изображением и вещественное число θ) и поворачивает изображение на θ градусов против часовой стрелки. Чтобы выполнить поворот, скопируйте значение каждого пиксела (s_i, s_j) исходного изображения в пиксел (t_i, t_j) приемника, координаты которого вычисляются по формулам:

$$t_i = (s_i - c_i)\cos\theta - (s_j - c_j)\sin\theta + c_i,$$

$$t_j = (s_i - c_i)\sin\theta - (s_j - c_j)\cos\theta + c_j,$$

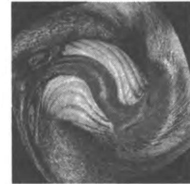
где (c_i, c_j) — центр изображения.

3.1.37. *Фильтр «завихрение».* Фильтр «завихрение» создается почти так же, как и фильтр поворота, но угол изменяется как функция расстояния от центра изображения. Используйте формулы из предыдущего упражнения, но вычислите θ как функцию (s_i, s_j), а конкретнее, как произведение $\pi/256$ и расстояния до центра.

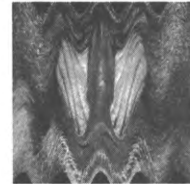
3.1.38. *Фильтр «волна».* Напишите фильтр для создания эффекта волны (по аналогии с двумя предыдущими упражнениями). Значение каждого пиксела (s_i, s_j) источника копируется в пиксел (t_i, t_j) приемника, где $t_i = s_i$, а $t_j = s_j + 20\sin(2\pi s_j/64)$. Добавьте код для передачи амплитуды (20 в приведенном примере) и частоты (64 в приведенном примере) как аргументов командной строки. Поэкспериментируйте с разными значениями этих параметров.

поворот на 30 градусов

фильтр «завихрение»



фильтр «волна»



фильтр «стекло»

Графические фильтры

3.1.39. *Фильтр «стекло».* Напишите программу, которая получает имя файла как аргумент командной строки и применяет фильтр «стекло»: каждому пикселу p присваивается цвет случайно выбранного соседнего пиксела (координаты которого отличаются от координат p не более чем на 5).

3.1.40. *Презентация.* Напишите программу, которая получает как аргументы командной строки имена нескольких графических файлов и отображает их в режиме презентации (один файл каждые две секунды) с использованием эффектов «растворения» и «проявления» при смене изображений.

3.1.41. *Трансформация.* Примеры изображений для программы `Fade` неточно выравниваются по вертикали (у обезьяны рот намного ниже, чем у Дарвина). Измените программу `Fade`, чтобы добавить преобразование по вертикальной оси, с которым переход становится более плавным.

3.1.42. *Цифровое увеличение.* Напишите программу `Zoom`, которая получает как аргументы командной строки имя файла с изображением и три числа s, x и y и выводит масштабированную часть входного изображения. Все числа лежат в диапазоне от 0 до 1; s определяет коэффициент масштабирования, а (x, y) — относительные координаты точки, которая должна находиться в центре выходного изображения. Используйте программу для увеличения части изображения — например, родственника или домашнего питомца на цифровой фотографии. (Если фотография была сделана на старом мобильном телефоне или камере, масштабирование может сопровождаться появлением визуальных артефактов.)

```
% java Zoom boy.jpg 1 .5 .5
```

© 2014 Janine Dietz

```
% java Zoom boy.jpg .5 .5 .5
```

```
% java Zoom boy.jpg .2 .48 .5
```

Цифровое увеличение

3.2. Создание типов данных

Вообще говоря, мы могли бы писать все программы, ограничиваясь восьмью примитивными типами. Но, как было показано в предыдущем разделе, программировать на более высоком уровне абстракции намного удобнее, поэтому в языке Java и его библиотеках можно найти множество типов данных. Тем не менее нельзя ожидать, что в Java будут встроены все возможные типы данных, которые вам когда-либо понадобятся, поэтому программист должен иметь возможность определять собственные типы. В этом разделе вы узнаете, как построить собственный тип данных в виде класса Java.

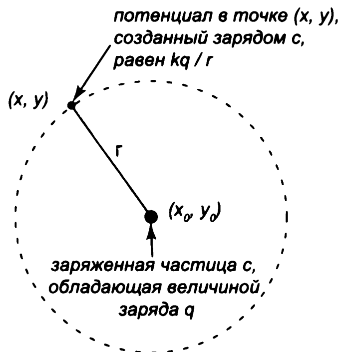
Реализация типа данных в виде класса Java принципиально не отличается от реализации библиотеки статических методов. Основное различие заключается в том, что с реализациями методов связываются *данные*. API определяет конструкторы и методы экземпляров, которые необходимо реализовать, но разработчик может выбрать любое удобное представление. Чтобы закрепить базовые концепции, мы начнем с реализации типа данных для заряженных частиц, представленного в нача-

ле раздела 3.1. Затем процесс создания типов данных будет продемонстрирован на нескольких примерах, от комплексных чисел до учета товарных запасов, включая ряд программных инструментов, которые будут использоваться в дальнейшем. Решающий прикладную задачу клиентский код — доказательство полезности любого типа данных, поэтому мы рассмотрим также ряд клиентов, в том числе и приложение для отображения знаменитого чарующего *множества Мандельброта*.

Построение типов данных называют также *абстрагированием данных*. Разработчик уделяет основное внимание данным и реализует операции над ними. *Всегда, когда вы можете четко выделить данные и связанные с ними операции в своей программе, — сделайте это*. Моделирование физических объектов или математических абстракций — понятный и чрезвычайно полезный процесс, но истинная сила абстракции данных заключается в том, что она позволяет моделировать вообще *все*, что вы можете точно описать. Когда у вас появится опыт использования такого стиля программирования, вы увидите, что он помогает решать задачи любой сложности.

Основные элементы типа данных

Чтобы продемонстрировать процесс реализации типа данных в классе Java, мы рассмотрим тип данных `Charge` для заряженных частиц. В данном случае нас интересует двумерная модель на базе *закона Кулона*, утверждающего, что электрический потенциал в заданной точке, обусловленный воздействием заряженной частицы, равен $V = kq/r$, где q — величина заряда, r — расстояние от точки до заряда, а электростатическая константа k = 8,99 × 10^9 Н × $м^2/Кл^2$. При наличии нескольких заряженных частиц электрический потенциал в любой точке вычисляется как сумма потенциалов, созданных каждым зарядом. В системе СИ в этой формуле используются следующие единицы измерения: ньютоны (сила), метры (расстояние) и кулоны (электрический заряд).



Закон Кулона для заряженной частицы

API

Программный интерфейс приложения (API) определяет контракт со всеми клиентами, а следовательно, является отправной точкой для любой реализации. Ниже приведено описание API для нашего класса заряженных частиц.

public class Charge		
	`Charge(double x0, double y0, double q0)`	
`double`	`potentialAt(double x, double y)`	электрический потенциал в точке (x, y), созданный зарядом
`String`	`toString()`	строковое представление

API класса заряженных частиц (см. листинг 3.2.1).

Чтобы реализовать тип данных `Charge`, необходимо определить значения типа данных и реализовать конструктор, создающий объект «заряженная частица»; метод `potentialAt()`, возвращающий потенциал в точке (x,y), созданный присутствием данного заряда; и метод `toString()`, возвращающий строковое представление заряда.

Класс

В Java тип данных реализуется в классе. Как и в случае с библиотеками статических методов, которые мы использовали, код типа данных хранится в файле с именем, совпадающим с именем класса, и расширением .java. Ранее мы уже реализовывали классы Java, но в этих классах отсутствовали возможности, предоставляемые типами данных: *переменные экземпляров*, *конструкторы* и *методы экземпляров*. Каждый из этих структурных элементов также дополняется *модификатором уровня доступа* (или видимости). Сейчас эти четыре концепции будут рассмотрены с примерами, а итогом этого описания станет тип данных `Charge` (листинг 3.2.1).

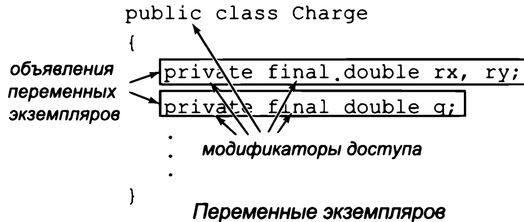
Модификаторы доступа

Ключевые слова `public`, `private` и `final`, иногда располагающиеся перед именем класса, называются *модификаторами доступа*. Модификаторы `public` и `private` управляют доступом со стороны клиентского кода: каждой переменной экземпляра и методу класса назначается либо *открытый* уровень доступа `public` (доступно для клиентов), либо *закрытый* (*«личный»*) уровень доступа `private` (недоступно для клиентов). Модификатор `final` означает, что значение переменной не может изменяться после инициализации — она доступна только для чтения. Мы будем использовать модификатор `public` для конструкторов и методов API (так как они будут предоставляться клиентам) и модификатор `private` для всего остального. Как правило, закрытые методы выполняют вспомогательные функции и используются для упрощения кода других методов класса. Язык Java достаточно либерально относится к использованию этих модификаторов — обсуждение причин для выбора той или иной схемы управления доступом откладывается до раздела 3.3.

Переменные экземпляров

Чтобы писать код методов экземпляров, работающих со значениями этого типа данных, сначала необходимо объявить переменные экземпляров, которые и будут использоваться для обращения к этим значениям в коде. Такие переменные могут иметь любой из доступных типов данных. Типы и имена переменных экземпляров объявляются так же, как при объявлении локальных переменных: так, для `Charge` используются три переменные `double` — две описывают позицию заряда на плоскости, а третья содержат величину заряда. Эти объявления располагаются в первых строках описания класса, а не внутри метода `main()` или любого другого метода. Между переменными экземпляров и уже знакомыми вам *локальными переменными* в методе или блоке существует различие: каждой локальной переменной в заданный момент времени соответствует всего одно значение, а переменная экземпляра для каждого объекта (экземпляра этого типа данных) будет иметь свое собственное значение, не зависящее от других экземпляров этого же типа.

Такая схема не создает неоднозначностей, потому что при каждом вызове метода экземпляра этот вызов связывается с конкретной ссылкой и действия выполняются над значениями того объекта, на который указывает эта ссылка.



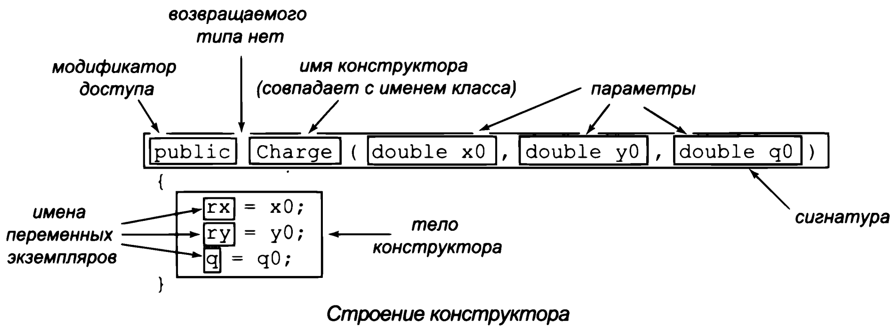
Переменные экземпляров

Конструкторы

Конструктором называется специальный метод, который создает объект и предоставляет ссылку на этот объект. Java автоматически вызывает конструктор, когда в клиентской программе используется ключевое слово `new`. Java выполняет большую часть работы: нашему коду остается только инициализировать переменные экземпляров содержательными значениями. Имя конструктора всегда совпадает с именем класса, но он может быть перегруженным — как и в случае со статическими методами, можно создать несколько конструкторов с разными сигнатурами. Для клиента комбинация из ключевого слова `new`, за которым следует имя конструктора (с аргументами, заключенными в круглые скобки), эквивалентна вызову функции, возвращающему ссылку на объект заданного типа. Сигнатура конструктора не имеет возвращаемого типа, потому что конструкторы всегда возвращают ссылку на объект своего типа данных (имена типа, класса и конструктора совпадают). Каждый раз, когда клиент вызывает конструктор, Java автоматически:

- выделяет память для объекта;
- вызывает код конструктора для инициализации переменных экземпляров;
- возвращает ссылку на созданный объект.

Конструктор `Charge` вполне типичен: он инициализирует переменные экземпляров значениями, переданными клиентом в качестве аргументов.



Строение конструктора

Методы экземпляров

Чтобы реализовать методы экземпляров, мы пишем для них код точно так же, как и в главе 2 для статических методов (функций). Каждый метод обладает сигнатурой (отражающей возвращаемый тип, а также типы и имена параметров) и телом (последовательность строк кода, заканчивающаяся `return` со значением возвращаемого типа, которое передается клиенту). Когда клиент вызывает метод экземпляра, система инициализирует его параметры значениями, переданными клиентом; выполняет код, пока не встретит `return`, и возвращает вычисленное значение клиенту. Фактически это выглядит как если бы вызов метода на стороне клиента был просто заменен возвращаемым значением. Таким образом, все происходит так же, как и при вызове статических методов, но с одним принципиальным различием: *методы экземпляров могут выполнять операции с переменными экземпляров.*

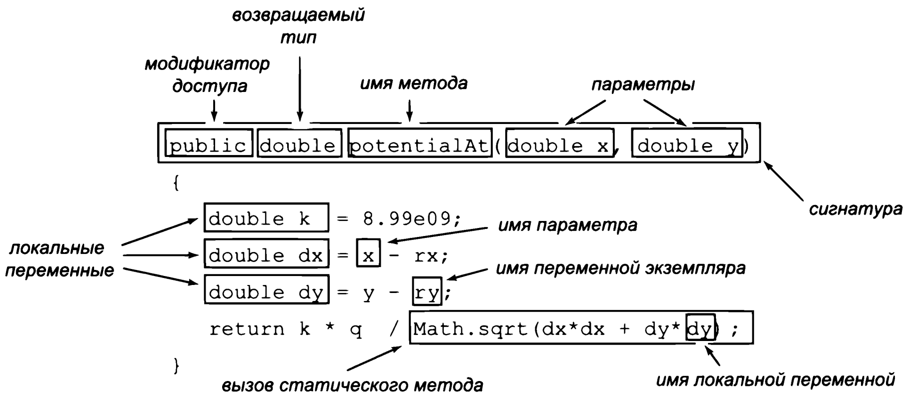
Переменные в методах

В коде Java, который мы пишем для реализации методов экземпляров, используются переменные трех видов:

- переменные-параметры;
- локальные переменные;
- переменные экземпляров.

Первые две категории такие же, как для статических методов: параметры определяются в сигнатуре метода и инициализируются значениями, переданными клиентом при вызове метода, а локальные переменные объявляются и инициализируются

в теле метода. Областью видимости параметров является весь метод; область видимости локальных переменных ограничена следующими за их определением строками в блоке, где они определены. С переменными экземпляров дело обстоит совершенно иначе: они содержат значения, принадлежащие объектам класса, а их областью видимости является весь класс. Как определить, значение какого объекта должно при этом использоваться? Если задуматься над этим вопросом, вы быстро вспомните ответ. Каждый объект класса содержит значение: код метода класса работает со значением *того объекта, который использовался для вызова метода.* Например, в выражении `c1.potentialAt(x, y)` код `potentialAt()` работает с переменными экземпляра `c1`.



Строение метода экземпляра

В реализации `potentialAt()` в `Charge` используются все три вида переменных, как показано на приведенной выше диаграмме и на сводке в следующей таблице.

переменная	назначение	пример	область видимости
переменная экземпляра	для определения значения типа данных	`rx, ry`	класс
параметр	для передачи значения от клиента в метод	`x, y`	метод
локальная переменная	для временного использования в методе	`dx, dy`	блок

Переменные в методах экземпляра

Убедитесь в том, что вы понимаете *суть различий между этими тремя разновидностями переменных, используемыми в реализации методов экземпляров.* В этих различиях заключается суть объектно-ориентированного программирования.

Листинг 3.2.1. Класс «заряженная частица»

```
public class Charge
{
   private final double rx, ry;
   private final double q;

   public Charge(double x0, double y0, double q0)
   {  rx = x0; ry = y0; q = q0;  }

   public double potentialAt(double x, double y)
   {
      double k = 8.99e09;
      double dx = x - rx;
      double dy = y - ry;
      return k * q / Math.sqrt(dx*dx + dy*dy);
   }

   public String toString()
   {
      return q + " at " + "(" + rx + ", " + ry + ")";
   }

   public static void main(String[] args)
   {
      double x = Double.parseDouble(args[0]);
      double y = Double.parseDouble(args[1]);
      Charge c1 = new Charge(0.51, 0.63, 21.3);
      Charge c2 = new Charge(0.13, 0.94, 81.9);
      StdOut.println(c1);
      StdOut.println(c2);
      double v1 = c1.potentialAt(x, y);
      double v2 = c2.potentialAt(x, y);
      StdOut.printf("%.2e\n", (v1 + v2));
   }
}
```

`rx, ry`	проверяемая точка
`q`	заряд

`k`	электростатическая постоянная
`dx, dy`	расстояние до проверяемой точки

`x, y`	проверяемая точка
`c1`	первый заряд
`v1`	потенциал от c1
`c2`	второй заряд
`v2`	потенциал от c2

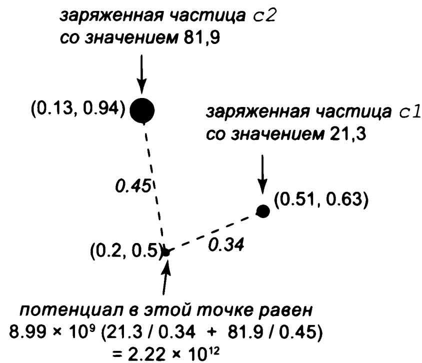
Наша реализация типа данных для моделирования заряженных частиц содержит базовые элементы, присутствующие в любом типе данных: переменные экземпляров rx, ry и q; конструктор Charge(); методы экземпляров potentialAt() и toString(); тестовый клиент main().

```
% java Charge 0.2 0.5
21.3 at (0.51, 0.63)
81.9 at (0.13, 0.94)
2.22e+12
```

```
% java Charge 0.51 0.94
21.3 at (0.51, 0.63)
81.9 at (0.13, 0.94)
2.56e+12
```

Тестовый клиент

Каждый класс может содержать собственный метод `main()`, который обычно резервируется для тестирования типа данных. Как минимум тестовый класс должен вызывать каждый конструктор и каждый метод экземпляра в классе. Например, метод `main()` в программе 3.2.1 получает два аргумента командной строки `x` и `y`, создает два объекта `Charge` и выводит данные обеих заряженных частиц и результирующий электрический потенциал в точке `(x, y)`, создаваемый этими двумя частицами. При наличии нескольких заряженных частиц электрический потенциал в любой точке вычисляется суммированием потенциалов, созданных каждым зарядом.



Таковы базовые элементы, которые необходимо понимать для определения собственных типов данных в Java. Каждая реализация типа данных (класс Java), которую мы разработаем, будет содержать такие же базовые элементы, как в первом примере: переменные экземпляров, конструкторы, методы экземпляров и тестовый клиент. Все типы данных, которые вы будете определять, будут создаваться по одной схеме. Вместо того чтобы думать о действиях, которые следует выполнить для достижения цели (как вы делали тогда, когда учились программировать), вы думаете о потребностях клиента, а затем воплощаете их в типе данных.

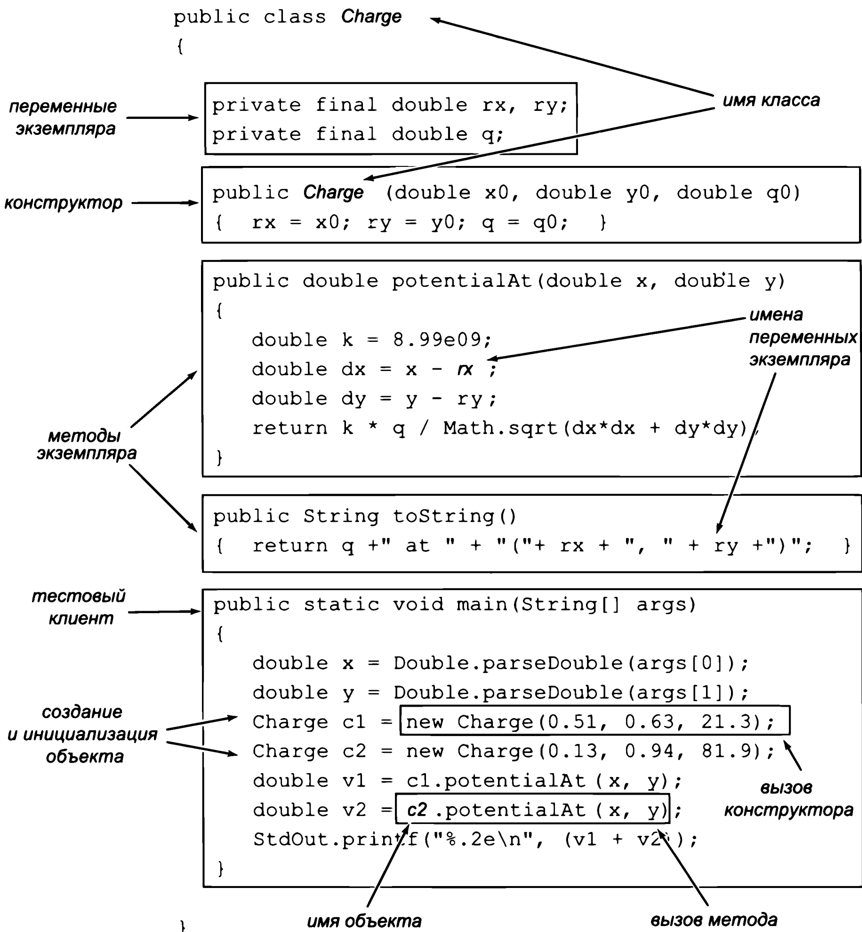
На первом этапе создания типа данных определяется API. Цель API — *отделить клиента от реализации*, чтобы сделать возможным модульное программирование. При определении API следует учитывать два момента. Во-первых, API должен способствовать четкости и правильности клиентского кода. Более того, перед закреплением окончательного вида API можно написать тестовый клиентский код, чтобы удостовериться в том, что включенные в него операции с типом данных действительно нужны клиенту. Во-вторых, вы должны понимать, как эти операции будут реализованы, — в противном случае определять их просто бессмысленно.

На втором этапе создания типа данных реализуется класс Java, удовлетворяющий спецификациям API. Сначала вы выбираете переменные экземпляров, а затем пи-

шете код конструкторов и методов экземпляра, которые работают с переменными экземпляра.

На третьем этапе создания типа данных пишутся тестовые клиенты для проверки проектировочных решений, принятых на первых двух этапах.

Какие значения определяют тип данных и какие операции клиенты должны выполнять с этими значениями? После того как эти базовые решения будут приняты, вы сможете создавать новые типы данных и писать клиентский код, который работает с вашими типами так же, как и со встроенными типами. В конце этого раздела приведены упражнения, которые помогут вам приобрести опыт в создании типов данных.



Строение класса

Stopwatch

Одна из отличительных особенностей объектно-ориентированного программирования — удобство моделирования реальных объектов посредством создания их абстракций — программных объектов. Рассмотрим простой пример — программу `Stopwatch` (листинг 3.3.2), которая реализует следующий API:

`public class Stopwatch`	
`Stopwatch()`	создает новый объект секундомера и запускает его
`double elapsedTime()`	возвращает время, прошедшее с момента создания объекта (в секундах)

API класса «секундомер» (см. листинг 3.2.2)

В общем, `Stopwatch` представляет собой усеченную версию классического секундомера. При создании объекта начинается отсчет времени, а чтобы узнать, сколько времени работал секундомер, вы вызываете метод `elapsedTime()`. Конечно, вы и сами легко предложите всевозможные усовершенствования класса `Stopwatch` — все зависит только от вашего воображения. Возможность сброса секундомера? Остановки и запуска? Интервальный таймер? Все эти функции легко реализуются (см. упражнение 3.2.12).

Реализация `Stopwatch` использует системный метод Java `System.currentTimeMillis()`, который возвращает значение типа `long` с текущим временем в миллисекундах (количество миллисекунд с полуночи 1 января 1970 года UTC). Трудно себе представить более простую реализацию типа данных. `Stopwatch` сохраняет время своего создания в переменной экземпляра, а когда клиент вызывает метод `elapsedTime()`, возвращает разность между текущим временем и сохраненным. Класс `Stopwatch` сам по себе время не отсчитывает (отсчет ведут внутренние системные часы вашего компьютера); он только создает иллюзию отсчета для клиента. Почему бы просто не использовать `System.currentTimeMillis()` в клиентах, спросите вы? Можно, но использование `Stopwatch` делает код более понятным и простым в сопровождении.

Классический секундомер

Тестовый клиент вполне типичен. Он создает два объекта `Stopwatch`, использует их для измерения времени выполнения двух разных вычислений, а затем выводит измеренное время. Вопрос о том, какой из двух вариантов решения вычислительной задачи лучше, играет важную роль в разработке программ. В разделе 4.1 будет представлен научный подход к оценке сложности и трудоемкости вычислений. Класс `Stopwatch` пригодится вам в этом решении.

Листинг 3.2.2. Stopwatch

```
public class Stopwatch
{
   private final long start;

   public Stopwatch()
   {  start = System.currentTimeMillis();  }

   public double elapsedTime()
   {
       long now = System.currentTimeMillis();
       return (now - start) / 1000.0;
   }

   public static void main(String[] args)
   {
      // Вычисления с использованием Math.sqrt() и хронометраж.
      int n = Integer.parseInt(args[0]);
      Stopwatch timer1 = new Stopwatch();
      double sum1 = 0.0;
      for (int i = 1; i <= n; i++)
         sum1 += Math.sqrt(i);
      double time1 = timer1.elapsedTime();
      StdOut.printf(„%e (%.2f seconds)\n", sum1, time1);

      // Вычисления с использованием Math.pow() и хронометраж.
      Stopwatch timer2 = new Stopwatch();
      double sum2 = 0.0;
      for (int i = 1; i <= n; i++)
         sum2 += Math.pow(i, 0.5);
      double time2 = timer2.elapsedTime();
      StdOut.printf("%e (%.2f seconds)\n", sum2, time2);
   }
}
```

`start`	время создания

Класс реализует простой тип данных, который можно использовать для сравнения времени выполнения двух методов, критичных по времени (см. раздел 4.1). Тестовый клиент сравнивает время выполнения двух функций для вычисления квадратного корня из библиотеки Java Math. Для задачи вычисления суммы квадратных корней чисел от 1 до n версия, вызывающая Math.sqrt(), работает в 10 раз быстрее версии, вызывающей Math.pow(). Впрочем, скорее всего, этот результат будет зависеть от конкретной системы.

```
% java Stopwatch 100000000
6.666667e+11 (0.65 seconds)
6.666667e+11 (8.47 seconds)
```

Гистограмма

А теперь рассмотрим тип для визуализации данных с использованием хорошо известной разновидности диаграмм, называемой *гистограммой*. Для простоты будем предполагать, что данные состоят из целых чисел в диапазоне от 0 до $n - 1$. Класс подсчитывает количество вхождений каждого значения и рисует на диаграмме столбец, высота которого пропорциональна относительной частоте. В следующей таблице приведена сводка API типа `Histogram`.

`public class Histogram`

	`Histogram(int n)`	создает гистограмму для целочисленных значений от 0 до n – 1
`double`	`addDataPoint(int i)`	добавляет вхождение значения i
`void`	`draw()`	отображает гистограмму средствами StdDraw

API класса «гистограмма» (см. листинг 3.2.3)

Разработка типа данных начинается с определения используемых переменных экземпляров. В данном случае в качестве переменной экземпляра будет использоваться массив. А именно класс `Histogram` (листинг 3.2.3) содержит переменную экземпляра `freq[]`, в которой `freq[i]` содержит количество вхождений значения `i` в данных, для всех `i` от 0 до `n-1`. Также `Histogram` включает целочисленную переменную экземпляра `max`, в которой хранится максимальная частота по всем значениям (что соответствует высоте самого высокого столбца). Метод экземпляра `draw()` использует переменную `max` для выбора масштаба стандартного графического окна по оси y и вызывает метод `StdStats.plotBars()` для рисования гистограммы. Метод `main()` — простой клиент для проведения испытаний Бернулли. Он существенно проще `Bernoulli` (листинг 2.2.6), потому что в нем используется тип данных `Histogram`.

Создавая такой тип данных, как `Histogram`, мы пользуемся преимуществами модульного программирования (повторное использование кода, независимая разработка небольших программ и т. д.), упоминавшимися в главе 2, с дополнительным преимуществом разделения данных. Без типа `Histogram` нам пришлось бы смешивать код построения гистограммы с кодом управления данными, в результате чего программа стала бы гораздо более сложной для понимания и сопровождения, чем две отдельные программы. *Всегда, когда вы можете четко выделить данные и связанные с ними операции в своей программе, — сделайте это.*

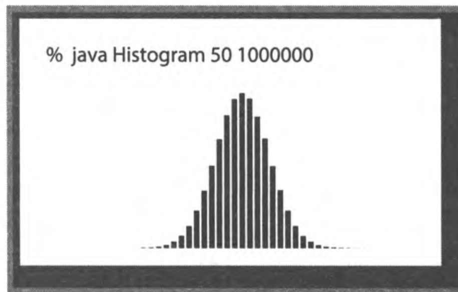
Черепашья графика

Всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это. В объектно-ориентированном программировании эта мантра дополняется *данными задачи*, или ее *состоянием*. Именно состояние — это та небольшая добавка, которая приносит огромную пользу для упрощения вычислений. В этом примере рассматривается так называемая *черепашья графика* — графика с использованием команд относительного перемещения, основанная на типе данных, определяемом в этом API.

Листинг 3.2.3. Класс Histogram

```
public class Histogram
{
   private final double[] freq;
   private double max;
   public Histogram(int n)
   {  // Создание новой гистограммы.
      freq = new double[n];
   }

   public void addDataPoint(int i)
   {  // Добавление одного вхождения значения i.
      freq[i]++;
      if (freq[i] > max) max = freq[i];
   }

   public void draw()
   {  // Выбор масштаба и рисование гистограммы.
      StdDraw.setYscale(0, max);
      StdStats.plotBars(freq);
   }

   public static void main(String[] args)
   {  // См. листинг 2.2.6.
      int n = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);
      Histogram histogram = new Histogram(n+1);
      StdDraw.setCanvasSize(500, 200);
      for (int t = 0; t < trials; t++)
         histogram.addDataPoint(Bernoulli.binomial(n));
      histogram.draw();
   }
}
```

`freq[]`	счетчики частот
`max`	максимальная частота



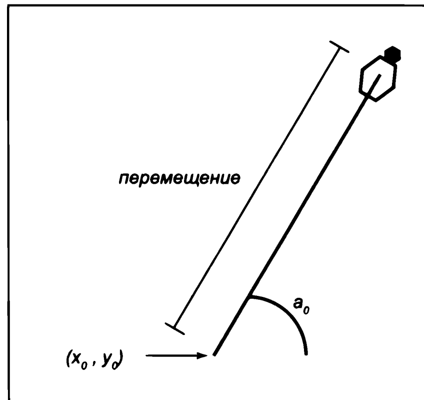
Этот тип данных обеспечивает простое построение гистограмм частот целых значений от 0 до n – 1. Частоты хранятся в переменной экземпляра, которая является массивом. В целочисленной переменной экземпляра max хранится максимальная частота (для масштабирования по оси y при построении гистограммы). Клиент представляет собой модификацию программы Bernoulli (листинг 2.2.6), но он существенно упрощается благодаря использованию типа данных Histogram.

public class Turtle	
Turtle(double x0, double y0, double a0)	создает в точке (x0, y0) новый объект относительного курсора («черепаху»), повернутый на угол a0 градусов против часовой стрелки от оси x
void turnLeft(double delta)	выполняет поворот на delta градусов против часовой стрелки
void goForward(double step)	перемещается вперед на расстояние step с рисованием линии

API черепашьей графики (см. листинг 3.2.4)

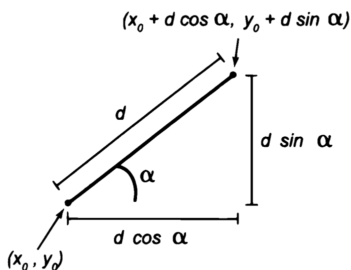
Представьте черепашку, которая живет в единичном квадрате и рисует линии во время перемещения. Она может двигаться на заданное расстояние по прямой линии или поворачивать налево (против часовой стрелки) на заданный угол. В соответствии с API при создании объекта «черепаха» мы размещаем его в заданной точке и ориентируем в заданном направлении. Далее остается построить рисунок, передавая черепахе последовательность команд `goForward()` и `turnLeft()`.

```
double x0 = 0.5;
double y0 = 0.0;
double a0 = 60.0;
double step = Math.sqrt(3)/2;
Turtle turtle  = new Turtle(x0, y0, a0);
turtle.goForward(step);
```



Первое перемещение

Например, чтобы нарисовать равносторонний треугольник, мы создаем объект `Turtle`, расположенный в точке (0,5, 0) и ориентированный под углом 60 градусов против часовой стрелки относительно оси x, приказываем ему двигаться вперед, выполнить поворот на 120 градусов против часовой стрелки, снова передвинуться вперед, повернуться еще на 120 градусов и снова передвинуться вперед для за-



Тригонометрия перемещения

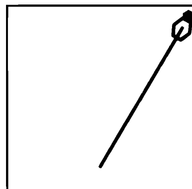
вершения треугольника. Собственно, почти все использующие `Turtle` программы, которые мы будем рассматривать, просто создают объект `Turtle` и передают ему серию чередующихся команд перемещения и поворота с разными расстояниями и углами. Как будет показано на нескольких ближайших страницах, эта простая модель позволяет строить рисунки произвольной сложности и находит много практических применений.

Программа `Turtle` (листинг 3.2.4) реализует этот API с использованием `StdDraw`. Класс содержит три переменные экземпляра: координаты текущей позиции и текущее направление (в градусах против часовой стрелки от оси x). Реализация двух методов предполагает *изменение* значений этих переменных, поэтому они не объявляются с ключевым словом `final`. Эти изменения вполне тривиальны: `turnLeft(delta)` прибавляет `delta` к текущему углу, а `goForward(step)` увеличивает текущую координату x на величину перемещения, умноженную на косинус аргумента, а текущую координату y — на величину перемещения, умноженную на синус аргумента.

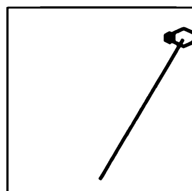
Тестовый клиент `Turtle` получает целочисленный аргумент командной строки `n` и рисует правильный многоугольник с `n` сторонами. Читателям, интересующимся элементарной аналитической геометрией, будет интересно проверить этот факт. Как бы то ни было, подумайте, что нужно было бы выполнить для вычисления координат всех вершин многоугольника. Простота «черепашьего» решения выглядит очень привлекательно. В сущности, графика с использованием команд относительного перемещения является полезной абстракцией для описания всевозможных геометрических фигур. Например, выбор достаточно большого значения `n` позволяет получить неплохую аппроксимацию окружности.

Объекты `Turtle` используются так же, как любые другие объекты. Программы могут создавать массивы объектов `Turtle`, передавать их в функции как аргументы и т. д. Наши примеры продемонстрируют эти возможности и наглядно покажут, что создать тип данных, такой как

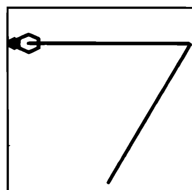
`turtle.goForward(step);`



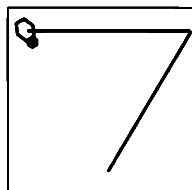
`turtle.turnLeft(120.0);`



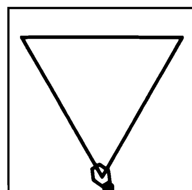
`turtle.goForward(step);`



`turtle.turnLeft(120.0);`



`turtle.goForward(step);`



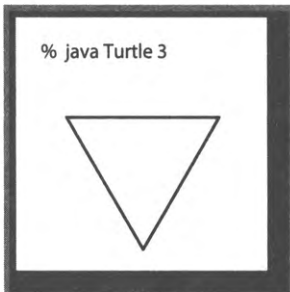
Первый рисунок

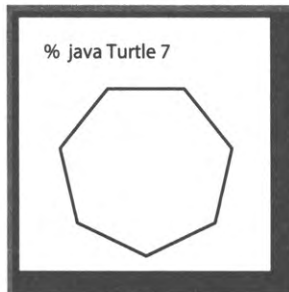
Листинг 3.2.4. Черепашья графика

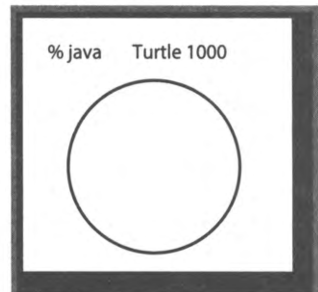
`x, y`	позиция (в единичном квадрате)
`angle`	направление движения (в градусах, отсчитываемых против часовой стрелки от оси x)

```
public class Turtle
{
   private double x, y;
   private double angle;

   public Turtle(double x0, double y0, double a0)
   {  x = x0; y = y0; angle = a0;  }

   public void turnLeft(double delta)
   {  angle += delta;  }

   public void goForward(double step)
   {  // Вычисление новой позиции, перемещение и рисование отрезка.
      double oldx = x, oldy = y;
      x += step * Math.cos(Math.toRadians(angle));
      y += step * Math.sin(Math.toRadians(angle));
      StdDraw.line(oldx, oldy, x, y);
   }

   public static void main(String[] args)
   {  // Рисование правильного многоугольника с n сторонами.
      int n = Integer.parseInt(args[0]);
      double angle = 360.0 / n;
      double step = Math.sin(Math.toRadians(angle/2));
      Turtle turtle = new Turtle(0.5, 0.0, angle/2);
      for (int i = 0; i < n; i++)
      {
         turtle.goForward(step);
         turtle.turnLeft(angle);
      }
   }
}
```

Этот тип данных реализует черепашью графику (с командами относительного перемещения), которая часто упрощает создание векторных рисунков.





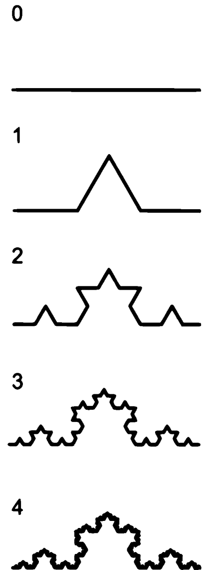


Turtle, несложно, а пользы от него может быть очень много. В любой такой ситуации с рисованием правильного многоугольника *можно* вычислить координаты всех точек и нарисовать отрезки, но *проще* сделать это при помощи Turtle. Таким образом, концепция черепашьей графики служит наглядным примером ценности абстракции данных.

Рекурсивная графика

Кривая Коха порядка 0 представляет собой отрезок прямой. Чтобы построить кривую Коха порядка n, следует нарисовать кривую Коха порядка $n - 1$, повернуть налево на 60°, нарисовать еще одну кривую Коха порядка $n - 1$, повернуть направо на 120°, нарисовать третью кривую Коха порядка $n - 1$, повернуть налево на 60° и нарисовать четвертую кривую Коха порядка $n - 1$. Эти рекурсивные инструкции естественно реализуются в коде графики с относительными командами. После внесения соответствующих изменений подобные рекурсивные схемы могут использоваться для моделирования самоподобных структур, встречающихся в природе, например снежинок.

```
public class Koch
{
   public static void koch(int n, double step, Turtle turtle)
   {
      if (n == 0)
      {
         turtle.goForward(step);
         return;
      }
      koch(n-1, step, turtle);
      turtle.turnLeft(60.0);
      koch(n-1, step, turtle);
      turtle.turnLeft(-120.0);
      koch(n-1, step, turtle);
      turtle.turnLeft(60.0);
      koch(n-1, step, turtle);
   }

   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      double step = 1.0 / Math.pow(3.0, n);
      Turtle turtle = new Turtle(0.0, 0.0, 0.0);
      koch(n, step, turtle);
   }
}
```



Построение кривых Коха с использованием черепашьей графики

Клиентский код получается тривиальным, если не считать величины перемещения. Если вы внимательно проанализируете несколько начальных примеров, то вы увидите (и сможете доказать посредством индукции), что ширина кривой порядка n равна произведению 3^n и величины перемещения, поэтому при выборе перемещения $1/3^n$ будет получена кривая ширины 1. Кроме того, количество шагов для построения кривой порядка n равно 4^n, поэтому для больших n построение кривой может не завершиться успешно. Многочисленные примеры рекурсивных узоров такого рода были созданы математиками, учеными и художниками разных направлений и в разных контекстах. Сейчас для нас важно то, что абстракция черепашьей графики серьезно упрощает код приложений, строящих такие узоры.

Логарифмическая спираль

Вероятно, черепаха устанет после 4^n перемещений во время рисования кривой Коха. Представьте, что при каждом перемещении его величина уменьшается с небольшим постоянным коэффициентом. Что произойдет с рисунком? Изменив тестовый клиент из листинга 3.2.4, который строил многоугольники, для получения ответа на этот вопрос, вы получите изображение, называемое *логарифмической спиралью*; эта кривая часто встречается в природе.

Программа `Spiral` (листинг 3.2.5) создает реализацию этой кривой. Она получает значение `n` и коэффициент ослабления как аргументы командной строки и приказывает черепашке поочередно перемещаться и поворачиваться, пока не будет сделано 10 полных оборотов. Как видно из четырех примеров, приведенных в программе, если коэффициент ослабления больше 1, спираль сходится к центру диаграммы. Аргумент `n` управляет формой спирали. Поэкспериментируйте с программой `Spiral` самостоятельно, чтобы понять, как параметры влияют на внешний вид спирали.

Логарифмическая спираль впервые была описана Рене Декартом в 1638 году. Якоб Бернулли был настолько потрясен математическими свойствами логарифмической спирали, что назвал ее *spira mirabilis* (чудесная спираль) и даже попросил выгравировать ее на своем надгробье. Многие люди тоже считают ее «чудесной», потому что такие формы часто встречаются в природе. Внизу изображены три примера: раковина наутилуса, рукава спиральной галактики и форма облаков в тропическом шторме. Ученые также обнаружили, что именно по такой траектории ястреб преследует свою жертву, а заряженная частица движется в перпендикулярном однородном магнитном поле.

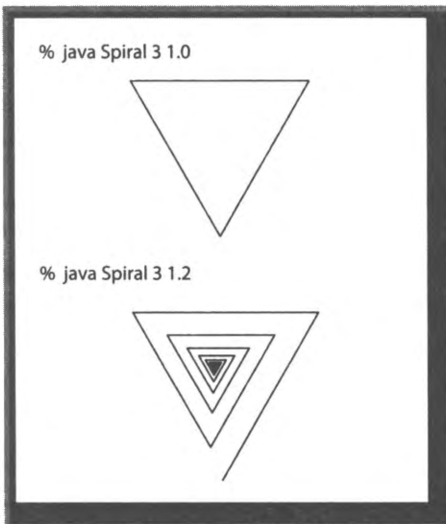
Одна из целей научных исследований — создание простых, но точных моделей сложных естественных явлений. И конечно, наша усталая черепашка проходит эту проверку!

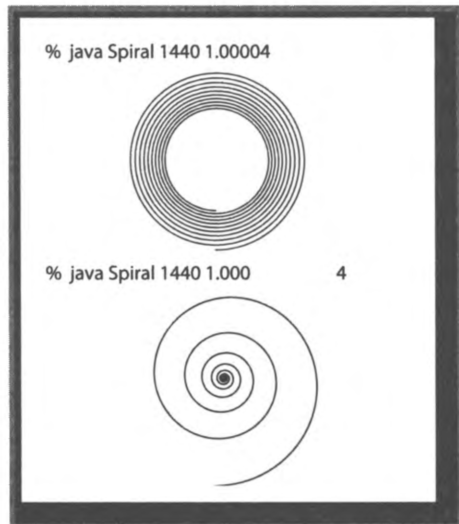
Листинг 3.2.5. Логарифмическая спираль

```
public class Spiral
{
   public static void main(String[] args)
   {
      int n         = Integer.parseInt(args[0]);
      double decay  = Double.parseDouble(args[1]);
      double angle  = 360.0 / n;
      double step   = Math.sin(Math.toRadians(angle/2));
      Turtle turtle = new Turtle(0.5, 0, angle/2);

      for (int i = 0; i < 10 * 360 / angle; i++)
      {
         step /= decay;
         turtle.goForward(step);
         turtle.turnLeft(angle);
      }
   }
}
```

step	**величина перемещения**
decay	**коэффициент ослабления**
angle	**угол поворота**
turtle	**усталая черепашка**

Измененная версия тестового приложения из листинга 3.2.4, в которой величина перемещения уменьшается при каждом шаге, а черепаха делает 10 полных поворотов. Форма спирали зависит от угла и коэффициента ослабления.





раковина наутилуса

Photo: Chris 73 (CC by-SA license)

спиральная галактика

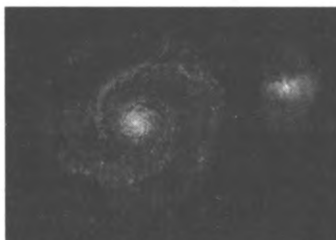


Photo: NASA and ESA

штормовые облака

Photo: NASA

Примеры логарифмических спиралей в природе

Броуновское движение

А может быть, уставшая черепашка решила выпить для бодрости. Представьте, что нетрезвая черепашка (снова действующая по стандартной схеме «поворот-перемещение») после каждого шага поворачивает в *случайном* направлении. Программа снова позволяет легко нарисовать путь, пройденный ею за миллионы шагов, и снова такой путь встречается в природе во многих контекстах. В 1827 году ботаник Роберт Броун увидел в микроскоп, что крошечные частицы пыльцы растений, погруженные в воду, хаотически двигаются во все стороны. Этот процесс, получивший название *броуновского движения*, позднее привел Альберта Эйнштейна к выводам об атомарной природе материи.

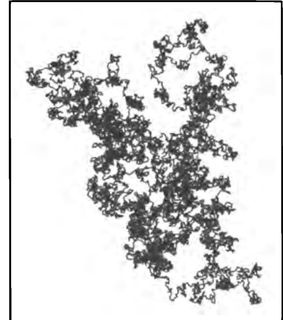
Возможно, у черепашки есть друзья, которые присоединились к попойке. После достаточно долгих блужданий траектории перемещений черепах сливаются и становятся неотличимы от одного пути. Современные астрофизики используют эту модель для анализа наблюдаемых свойств дальних галактик.

Концепция черепашьей графики изначально была разработана Сеймуром Пейпертом в Массачусетском технологическом институте в 1960-е годы как часть учебного языка программирования Logo, который до сегодняшнего дня используется в обучающих и развлекательных целях. Однако черепашья графика, как показывают многие серьезные примеры, — вовсе не игрушка. Она также находит много коммерческих применений. Например, она заложена в основу Postscript — языка программирования для создания печатных страниц в газетах, журналах и книгах[1]. В нашем случае `Turtle` становится типичным примером объектно-ориентированного программирования, который демонстрирует, как небольшой объем хранимых данных о внутреннем состоянии (абстракция данных с использованием объектов, а не функций) может кардинально упростить программу.

[1] Похожим образом организованы также наборы команд реальных устройств, например графопостроителей и даже станков с ЧПУ. — *Примеч. науч. ред.*

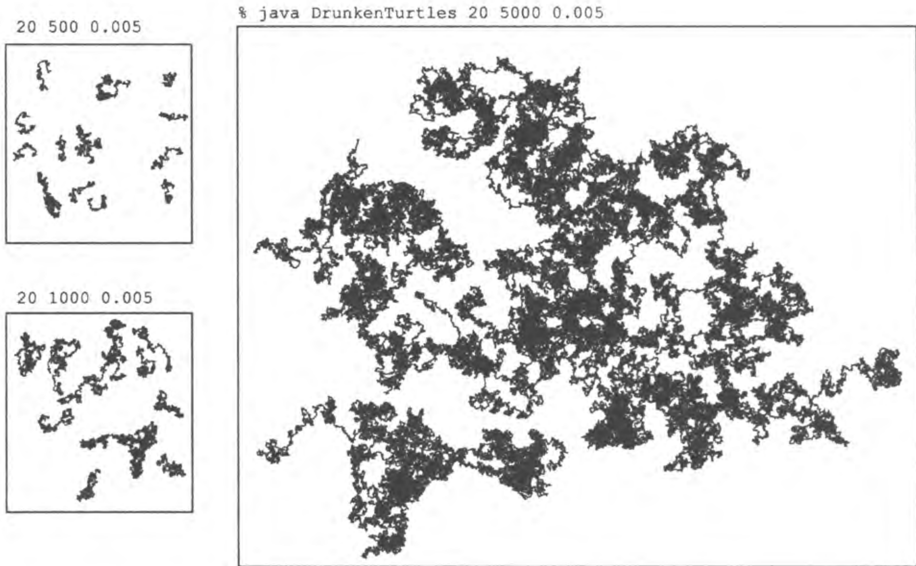
```
public class DrunkenTurtle
{
   public static void main(String[] args)
   {
      int trials = Integer.parseInt(args[0]);
      double step = Double.parseDouble(args[1]);
      Turtle turtle = new Turtle(0.5, 0.5, 0.0);
      for (int t = 0; t < trials; t++)
      {
         turtle.turnLeft(StdRandom.uniform(0.0, 360.0));
         turtle.goForward(step);
      }
   }
}
```

% java DrunkenTurtle 10000 0.01



Броуновское движение пьяной черепашки (перемещение на фиксированное расстояние в случайном направлении)

```
public class DrunkenTurtles
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);          // Количество черепах
      int trials = Integer.parseInt(args[1]);     // Количество перемещений
      double step = Double.parseDouble(args[2]);  // Длина перемещения
      Turtle[] turtles = new Turtle[n];
      for (int i = 0; i < n; i++)
      {
         double x = StdRandom.uniform(0.0, 1.0);
         double y = StdRandom.uniform(0.0, 1.0);
         turtles[i] = new Turtle(x, y, 0.0);
      }
      for (int t = 0; t < trials; t++)
      {  // Все черепахи совершают одно перемещение.
         for (int i = 0; i < n; i++)
         {  // Черепаха i делает один шаг в случайном направлении.
            turtles[i].turnLeft(StdRandom.uniform(0.0, 360.0));
            turtles[i].goForward(step);
         }
      }
   }
}
```



Броуновское движение ватаги пьяных черепах

Комплексные числа

Комплексное число имеет форму $x + iy$, где x и y — вещественные числа, а i — квадратный корень из -1. Число x называется *действительной* частью, а число y — *мнимой* частью (терминология происходит от представления о том, что квадратный корень из -1 является числом «воображаемым», так как среди вещественных чисел такого значения быть не может). Комплексные числа являются одной из важнейших математических абстракций: неважно, можете вы представить, что такое квадратный корень из -1 или нет, — комплексные числа помогают понять реальный мир. Они широко применяются в прикладной математике и играют большую роль во многих отраслях науки и техники. Они используются для моделирования всевозможных физических систем, от электрических цепей и звуковых волн до электромагнитных полей. Обычно такие модели требуют сложных вычислений, в которых к комплексным числам применяются вполне определенные арифметические операции, и нам нужно писать компьютерные программы для таких вычислений. Короче говоря, понадобится новый тип данных для представления комплексных чисел.

Разработка типа данных для комплексных чисел — классический пример объектно-ориентированного программирования. Никакой язык программирования не может предоставить реализации всех математических абстракций, которые могут когда-либо понадобиться, однако реализация типов данных не только упрощает работу

с такими абстракциями, как комплексные числа, многочлены, векторы и матрицы, но и позволяет мыслить понятиями новых абстракций.

Для выполнения базовых вычислений с комплексными числами необходима поддержка следующих операций: сложение и умножение, обладающие свойствами коммутативности, ассоциативности и дистрибутивности (наряду с условием $i^2 = -1$), вычисление модуля и извлечение действительной и мнимой части по следующим формулам:

- сложение: $(x + iy) + (v + iw) = (x + v) + i(y + w)$
- умножение: $(x + iy) \times (v + iw) = (xv \times yw) + i(yv + xw)$
- модуль: $|x + iy| = \sqrt{x^2 + y^2}$
- действительная часть: $\mathrm{Re}(x + iy) = x$
- мнимая часть: $\mathrm{Im}(x + iy) = y$

Например, если $a = 3 + 4i$ и $b = -2 + 3i$, то $a + b = 1 + 7i$, $a \times b = -18 + i$, $\mathrm{Re}(a) = 3$, $\mathrm{Im}(a) = 4$, $|a| = 5$.

При наличии этих базовых определений путь к реализации типа данных для комплексных чисел вполне ясен. Как обычно, все начинается с API, определяющего операции с типом данных:

public class Complex	
Complex(double real, double imag)	
Complex plus(Complex b)	сумма этого числа и b
Complex times(Complex b)	произведение этого числа и b
double abs()	модуль
double re()	действительная часть
double im()	мнимая часть
String toString()	строковое представление

API комплексных чисел (см. листинг 3.2.6)

Для простоты мы сосредоточимся на базовых операциях API, но в упражнении 3.2.19 вам будет предложено реализовать другие полезные операции, которые было бы уместно включить в API.

Программа `Complex` (листинг 3.2.6) — класс, реализующий этот API. Класс содержит такие же компоненты, как и `Charge` (и все остальные реализации типов данных в Java): переменные экземпляров (`re` и `im`), конструктор, методы экземпляров

Листинг 3.2.6. Комплексные числа

```
public class Complex
{
   private final double re;
   private final double im;

   public Complex(double real, double imag)
   {  re = real; im = imag;  }

   public Complex plus(Complex b)
   {  // Возвращает сумму текущего числа и b.
      double real = re + b.re;
      double imag = im + b.im;
      return new Complex(real, imag);
   }

   public Complex times(Complex b)
   {  // Возвращает произведение текущего числа и b.
      double real = re * b.re - im * b.im;
      double imag = re * b.im + im * b.re;
      return new Complex(real, imag);
   }

   public double abs()
   {  return Math.sqrt(re*re + im*im);  }

   public double re()  { return re; }
   public double im()  { return im; }

   public String toString()
   {  return re + " + " + im + "i";  }

   public static void main(String[] args)
   {
      Complex z0 = new Complex(1.0, 1.0);
      Complex z = z0;
      z = z.times(z).plus(z0);
      z = z.times(z).plus(z0);
      StdOut.println(z);
   }
}
```

re	действительная часть
im	мнимая часть

Тип данных предназначен для использования в Java-программах, выполняющих вычисления с комплексными числами.

```
% java Complex
-7.0 + 7.0i
```

(`plus()`, `times()`, `abs()`, `re()`, `im()` и `toString()`) и тестовый клиент. Тестовый клиент сначала присваивает z_0 значение $1 + i$, затем присваивает z значение z_0, после чего вычисляет

$$z = z^2 + z_0 = (1 + i)^2 + (1 + i) = (1 + 2i - 1) + (1 - i) = 1 + 3i$$

$$z = z^2 + z_0 = (1 + 3i)^2 + (1 + i) = (1 + 6i - 9) + (1 + i) = -7 + 7i$$

Код клиента достаточно прямолинеен и похож на код, который вы уже видели ранее в этой главе, с одним исключением: в коде, реализующем арифметические методы, используется новый механизм обращения к значениям объекта.

Обращение к переменным экземпляров объектов того же типа

Каждый из методов экземпляров `plus()` и `times()` должен обращаться к значениям двух объектов: объекта, переданного в аргументе, и объекта, использованного для вызова метода. Если метод вызывается в виде `a.plus(b)`, то вы можете обращаться к переменным экземпляров `a` по именам `re` и `im`, как обычно, но для обращения к переменным экземпляров `b` приходится использовать `b.re` и `b.im`. Объявление переменных экземпляров с ключевым словом `private` означает, что к переменным экземпляров нельзя обращаться напрямую из *другого* класса. Однако внутри класса можно напрямую обращаться к переменным экземпляров *любого* объекта *того же* класса, а не только к переменным экземпляров объекта, использованного для вызова.

Создание и возвращение новых объектов

Обратите внимание на то, как методы `plus()` и `times()` возвращают значения клиентам: они должны возвращать значение типа `Complex` , поэтому они вычисляют действительную и мнимую части, используют их для создания нового объекта, после чего возвращают ссылку на этот объект. Такой механизм позволяет клиентам работать с комплексными числами естественным образом, выполняя операции с локальными переменными типа `Complex`.

Сцепленные вызовы методов

Также обратите внимание на то, как в методе `main()` два вызова методов образуют одно компактное выражение Java `z.times(z).plus(z0)`, которое соответствует математическому выражению $z^2 + z_0$. Такой способ использования удобен тем, что вам не приходится изобретать имена переменных для промежуточных значений. Ведь для вызова метода может использоваться любая ссылка на объект, даже не имеющая собственного имени (как ссылка, полученная в результате вычисления подвыражения). Проанализировав выражение, вы увидите, что никакой неоднозначности в нем нет: если двигаться слева направо, каждый метод возвращает ссылку на объект `Complex`, которая затем используется для вызова следующего метода экземпляра в цепочке. При желании можно воспользоваться круглыми

скобками для измененя порядка действий, принятого по умолчанию (например, выражение Java `z.times(z.plus(z0))` соответствует математическому выражению $z(z + z_0)$).

Переменные экземпляров с ключевым словом final

Две переменные экземпляров `Complex` объявлены с ключевым словом `final`; это означает, что их значения задаются в объекте `Complex` при его создании и не изменяются на протяжении жизненного цикла объекта. Причины для такого решения обсуждаются в разделе 3.3.

Комплексные числа лежат в основе сложных вычислений из области прикладной математики. Используя класс `Complex`, мы можем сосредоточиться на разработке приложений, использующих комплексные числа, не отвлекаясь на повторную реализацию таких методов, как `times()`, `abs()` и т. д. Такие методы реализуются один раз и используются повторно — в отличие от альтернативного решения с копированием этого кода во все прикладные программы, использующие комплексные числа. Такой подход не только экономит время на отладке, но и позволяет изменять или совершенствовать реализацию в случае необходимости, поскольку реализация отделена от ее клиентов. *Всегда, когда вы можете четко выделить данные и связанные с ними операции в своей программе, — сделайте это.*

Чтобы дать представление о природе вычислений с комплексными числами и полезности абстракции комплексного числа, мы рассмотрим знаменитый пример клиента `Complex`.

Множество Мандельброта

Множество Мандельброта представляет собой множество целых чисел, открытое Бенуа Мандельбротом. Оно обладает многими удивительными свойствами. В частности, множество Мандельброта образует фрактальный узор, напоминающий папоротник Барнсли, треугольник Серпинского, броуновский мост, кривую Коха, путь пьяной черепахи и другие рекурсивные (самоподобные) узоры и программы, встречавшиеся в книге. Узоры такого рода часто встречаются в природе, и такие модели и программы чрезвычайно важны для современной науки.

Множество точек, входящих в множество Мандельброта, не описывается единственной математической формулой. Вместо этого оно определяется алгоритмом и, следовательно, идеально подходит для клиента `Complex`: чтобы изучить множество, мы напишем программу для его вывода.

Принадлежность комплексного числа z_0 множеству Мандельброта проверяется по простым правилам. Возьмем последовательность комплексных чисел z_0, z_1, z_2, ..., z_t, ..., где $z_{t+1} = (z_t)^2 + z_0$. Например, в следующей таблице приведены начальные элементы последовательности, соответствующей $z_0 = 1 + i$:

t	z_t	$(z_t)^2$	$(z_t)^2+z_0 = z_{t+1}$
0	$1 + i$	$1 + 2i + i^2 = 2i$	$2i + (1 + i) = 1 + 3i$
1	$1 + 3i$	$1 + 6i + 9i^2 = -8 + 6i$	$-8 + 6i + (1 + i) = -7 + 7i$
2	$-7 + 7i$	$49 - 98i + 49i^2 = -98i$	$-98i + (1 + i) = 1 - 97i$

Вычисление последовательности Мандельброта

Если ряд $|z_t|$ стремится к бесконечности, то z_0 *не* входит в множество Мандельброта; если ряд ограничен, то z_0 входит в него. Для некоторых точек проверка выполняется достаточно просто. Для многих других она требует большего объема вычислений, как показывают примеры в следующей таблице.

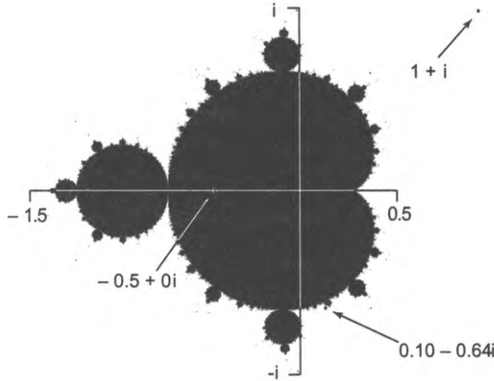
z_0	$0 + 0i$	$2 + 0i$	$1 + i$	$0 + i$	$-0,5 + 0i$	$-0,10 - 0,64i$
z_1	0	6	$1 + 3i$	$-1 + i$	$-0,25$	$-0,30 - 0,77i$
z_2	0	38	$-7 + 7i$	$-i$	$-0,44$	$-0,40 - 0,18i$
z_3	1446	$1 - 97i$	$1 + 3i$	$-1 + i$	$-0,31$	$0,23 - 0,50i$
z_4	0	2090918	$-9407 - 193i$	$-i$	$-0,40$	$-0,09 - 0,87i$
...	...	...	...	...	...	...
входит в множество?	да	нет	нет	да	да	да

Последовательность Мандельброта для нескольких начальных точек

Для краткости числа в двух правых столбцах таблицы приводятся с точностью до двух знаков. В некоторых случаях можно легко доказать, что числа входят в множество. Например, $0 + 0i$ безусловно принадлежит множеству (так как модуль всех чисел ряда равен 0), а $2 + 0i$ явно не входит в множество (так как в этом ряде доминируют степени 2, из-за которых ряд уходит в бесконечность). В других случаях рост не столь очевиден: например, $1 + i$ не входит в множество. В других рядах проявляется периодическое поведение. Например, для i значение $-1 + i$ переходит в $-i$, потом в $-1 + i$, потом снова $-i$ и т. д. У других последовательностей модуль чисел начинает возрастать только по прошествии очень большого времени.

Чтобы получить визуальное представление множества Мандельброта, мы берем выборку точек на комплексной плоскости — по аналогии с выборкой вещественных точек для построения графика функции с действительными значениями. Каждое комплексное число $x + iy$ соответствует точке (x,y) на плоскости, поэтому результаты могут отображаться следующим образом: для заданного разрешения n в заданном квадрате определяется матрица пикселов $n \times n$ с постоянным шагом. Если соответствующая точка входит в множество Мандельброта, то пиксел окрашивается в черный цвет, а если нет — в белый. Так образуется странный и завораживающий узор, в котором все черные точки образуют связную группу в квадрате приблизительно 2×2 с центром в точке $-1/2 + 0i$. Большие значения n позволяют

получить изображение с более высоким разрешением за счет увеличения объема вычислений. При более внимательном рассмотрении на графике проявляются закономерности. Например, один и тот же округлый узор с самоповторяющимися «отростками» проходит по всему контуру основной кардиоиды, а размеры этих «отростков» напоминают простую функцию делений линейки из листинга 1.2.1. А если увеличить изображение у края кардиоиды, видны крошечные самоповторяющиеся кардиоиды!



Множество Мандельброта

Но как именно построить такой график? Однозначного ответа на этот вопрос не существует, потому что нет простого критерия, который бы позволял сделать вывод о том, что точка гарантированно входит в множество. Для каждого комплексного числа можно провести вычисления от начала ряда, но и такое решение не позволяет с полной уверенностью определить, что ряд остается ограниченным. Существует критерий, который точно определяет, что комплексное число не входит в множество: если модуль любого числа последовательности превышает 2 (например, $1 + 3i$), то ряд заведомо расходится.

Программа `Mandelbrot` (листинг 3.2.7) использует этот критерий для отображения визуального представления множества Мандельброта. Так как наше представление о множестве уже не просто «черно-белое», в визуальном представлении используются оттенки серого. В основе программы лежит функция `mand()`, которая получает аргумент `z0` типа `Complex` и аргумент `max` типа `int` и вычисляет последовательность Мандельброта, начиная с `z0`. Функция возвращает количество итераций, для которых модуль остается меньшим (или равным) 2, не превышающее порога `max`.

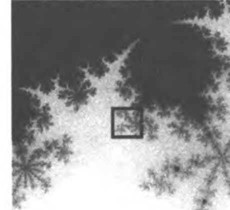
Для каждого пиксела метод `main()` класса `Mandelbrot` вычисляет комплексное число `z0`, соответствующее пикселу, после чего вычисляет выражение `255-mand(z0, 255)` для создания оттенка серого для пиксела. Любой пиксел, который окрашен не в черный цвет, соответствует комплексному числу, которое заведомо не входит в множество Мандельброта, потому что модуль хотя бы одного из чисел его ряда

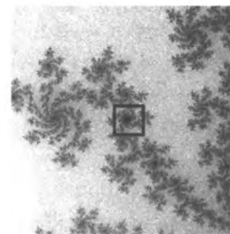
превышает 2 (а следовательно, ряд расходится в бесконечность). Черные пикселы (значение 0) соответствуют точкам, которые, как мы считаем, входят в множество, потому что модуль числа в первых 255 итерациях Мандельброта не превышает 2.

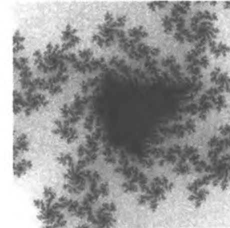
Даже эта простая программа строит на удивление сложные изображения, даже если увеличить крошечную часть плоскости. Для получения более эффектных картин можно воспользоваться цветом (см. упражнение 3.2.35). А множество Мандельброта выводится на основании итеративного выполнения всего одной функции $f(z) = (z^2 + z_0)$; а ведь изучение других функций также может принести много полезной информации.

За простотой кода скрывается основательный объем вычислений. Изображение 512 × 512 содержит 0,25 миллиона пикселов, и все черные пикселы требуют 255 итераций Мандельброта, поэтому построение изображения требует сотен миллионов операций со значениями `Complex`.

При всем внешнем очаровании множества Мандельброта нас оно интересует прежде всего как клиент `Complex` — для демонстрации того, что вычисления с типом данных, не встроенным в Java (например, с комплексными числами), могут быть абсолютно естественными и удобными. Программа `Mandelbrot` — пример того, как процесс вычислений стал более простым и естественным благодаря спроектированному и реализованному типу `Complex`. Программу `Mandelbrot` можно написать и без `Complex`, но тогда в ней фактически придется объединить содержимое листингов 3.2.6 и 3.2.7, и программа станет намного менее понятной. *Всегда, когда вы можете четко выделить данные и связанные с ними операции в своей программе, — сделайте это.*







Множество Мандельброта в увеличении

Обработка финансовой информации

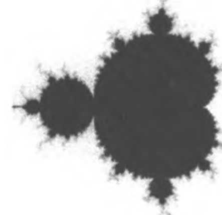
Одной из движущих сил разработки объектно-ориентированного программирования была растущая потребность во все большем количестве надежных программных продуктов для обработки финансовой информации. В качестве примера мы рассмотрим тип данных, который может использоваться в финансовой организации для хранения информации о клиенте.

Листинг 3.2.7. Множество Мандельброта

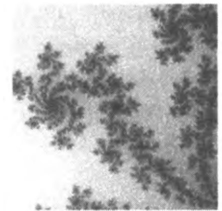
```
import java.awt.Color;

public class Mandelbrot
{
   private static int mand(Complex z0, int max)
   {
      Complex z = z0;
      for (int t = 0; t < max; t++)
      {
         if (z.abs() > 2.0) return t;
         z = z.times(z).plus(z0);
      }
      return max;
   }

   public static void main(String[] args)
   {
      double xc   = Double.parseDouble(args[0]);
      double yc   = Double.parseDouble(args[1]);
      double size = Double.parseDouble(args[2]);
      int n = 512;
      Picture picture = new Picture(n, n);
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
         {
            double x0 = xc - size/2 + size*i/n;
            double y0 = yc - size/2 + size*j/n;
            Complex z0 = new Complex(x0, y0);
            int gray = 255 - mand(z0, 255);
            Color c = new Color(gray, gray, gray);
            picture.set(i, n-1-j, c);
         }
      picture.show();
   }
}
```

x0, y0	точка на квадратном поле
z0	$x_0 + iy_0$
max	ограничение числа итераций
xc, yc	центр квадрата
size	квадрат имеет размеры size× size
n	матрица из n × n пикселов
pic	выводимое изображение
c	цвет выводимого пиксела

-.5 0 2



.1015 -.633 .01



Программа получает три аргумента командной строки, задающие центр и размер квадратной области, и строит цифровое изображение, отображающее результат проверки принадлежности точек, равномерно распределенных в виде матрицы size × size в заданной области, к множеству Мандельброта. Каждый пиксел окрашивается в оттенок серого, определяемый подсчетом итераций до того, как модуль очередного числа из ряда Мандельброта превысит 2,0 (но не более 255 итераций).

Предположим, биржевому брокеру нужно вести несколько портфелей клиентов с различными акциями. Таким образом, в множество значений, которые брокеру приходится обрабатывать, входит имя клиента, количество видов акций, их биржевые кодовые обозначния, количество акций каждого вида и количество имеющихся денежных средств. Для операций с портфелем акций клиента в API должны быть определены как минимум следующие методы.

public class StockAccount	
`StockAccount(String filename)`	создает новый портфель акций по данным файла
`double valueOf()`	сумма на счете в долларах
`void buy(int amount, String symbol)`	добавляет акции в портфель акций
`void sell(int amount, String symbol)`	удаляет акции из портфеля акций
`void save(String filename)`	сохраняет данные портфеля акций в файле
`void printReport()`	выводит подробный отчет об акциях и суммах

API для операций с портфелем акций (см. листинг 3.2.8)

Разумеется, брокер должен иметь возможность покупать и продавать акции, а также предоставлять отчеты клиенту, но ключом к пониманию обработки клиентских данных является конструктор `StockAccount()` и метод `save()` из этого API. Информация о клиенте предназначена для долгосрочного использования, поэтому она сохраняется в *файле* или в *базе данных*. Чтобы обработать данные портфеля акций, клиентская программа должна прочитать информацию из соответствующего файла, обработать ее требуемым образом и, если информация изменится, — записать ее обратно в файл для будущего использования. Чтобы реализовать это для данных портфеля, потребуется *формат файла* и *внутреннее представление* (структура данных в памяти).

В качестве (вымышленного) примера представим, что брокер ведет небольшой портфель акций ведущих компаний-разработчиков для Алана Тьюринга, основоположника современной информатики. Кстати говоря, история жизни Тьюринга весьма интересна и заслуживает более подробного знакомства. Среди прочего, Тьюринг работал над вычислительной криптографией, которая помогла приблизить конец Второй мировой войны, разработал основу теории вычислений, спроектировал и построил один из первых компьютеров и стал первооткрывателем в области искусственного интеллекта. Пожалуй, можно смело предположить, что Тьюринг, каким бы ни было его финансовое положение как научного работника в середине прошлого века, относился бы к перспективам программных продуктов в современном мире с оптимизмом и сделал бы небольшие инвестиции в них.

Формат файла

Современные системы часто используют текстовые файлы (даже для хранения данных), чтобы свести к минимуму зависимость от какой-либо конкретной программы. Для простоты мы будем использовать примитивное представление, в котором указывается имя держателя акций (символьная строка), баланс (вещественное число) и количество акций в портфеле. За ними следуют строки — по одной для каждого вида акций; в каждой строке указывается количество акций и биржевое кодовое обозначение, как в примере справа. Вообще говоря, информацию желательно помечать тегами вида `<Name>`, `<Number of shares>` и т. д., чтобы дополнительно снизить зависимость от любой конкретной программы, но здесь теги опущены для краткости[1].

```
% more Turing.txt
Turing, Alan
10.24
4
100 ADBE
 25 GOOG
 97 IBM
250 MSFT
```

Формат файла

Структура данных

Для представления информации, предназначенной для обработки, в программах Java используются переменные экземпляров. Они описывают характер информации и структурируют ее для простого и удобного обращения в коде программы. В нашем примере необходимы как минимум следующие элементы данных.

- Значение `String` для имени владельца.
- Значение `double` для текущего остатка денежных средств.
- Значение `int` для количества видов акций.
- Массив значений `String` для биржевых кодовых обозначений.
- Массив значений `int` для количества акций каждого вида.

Эти решения непосредственно отражены в переменных экземпляров в программе `StockAccount` (листинг 3.2.8). Массивы `stocks[]` и `shares[]` называются *параллельными массивами*. Для заданного индекса `i` конструкция `stocks[i]` возвращает биржевое сокращение, а `shares[i]` — количество их в портфеле. Также возможно другое решение: определить отдельный тип данных для акций, чтобы работать с этой информацией для каждого вида акций и хранить массив объектов этого типа в `StockAccount`.

```
public class StockAccount
{
   private final String name;
   private double cash;
   private int n;
   private int[] shares;
   private String[] stocks;
   ...
}
```

Эскиз структуры данных

[1] Такие «размеченные» (тегированные) форматы эффективно сочетают удобство для чтения человеком и переносимость текстовых данных с возможностью унифицированного представления структурированной информации. Характерный пример — форматы на основе языка разметки XML, ставшего в настоящее время общепринятым. — *Примеч. науч. ред.*

Листинг 3.2.8. Портфель акций

```
public class StockAccount
{
   private final String name;
   private double cash;
   private int n;
   private int[] shares;
   private String[] stocks;

   public StockAccount(String filename)
   {  // Построение структуры данных и загрузка из заданного файла.
      In in = new In(filename);
      name = in.readLine();
      cash = in.readDouble();
      n = in.readInt();
      shares = new int[n];
      stocks = new String[n];
      for (int i = 0; i < n; i++)
      {  // Обработка одного вида акций.
         shares[i] = in.readInt();
         stocks[i] = in.readString();
      }
   }

   public static void main(String[] args)
   {
      StockAccount account = new StockAccount(args[0]);
      account.printReport();
   }
}
```

`name`	имя клиента
`cash`	текущий остаток денежных средств
`n`	количество видов акций
`shares[]`	количество акций
`stocks[]`	биржевые кодовые обозначения

Класс для работы с портфелями акций демонстрирует типичное применение объектно-ориентированного программирования для обработки финансовой информации. Реализация printReport() приведена в тексте, а реализации priceOf(), save(), buy() и sell() — в упражнениях 3.2.22 и 4.4.65.

```
% more Turing.txt
Turing, Alan
10.24
4
100 ADBE
 25 GOOG
 97 IBM
250 MSFT
```

```
% java StockAccount Turing.txt
Turing, Alan
 100  ADBE      70.56     7056.00
  25  GOOG     502.30    12557.50
  97   IBM     156.54    15184.38
 250  MSFT      45.68    11420.00
                Cash:       10.24
                Total:   46228.12
```

Класс `StockAccount` включает конструктор, который читает файл в указанном формате и создает портфель акций на основе этой информации. Кроме того, брокер должен периодически предоставлять отчеты своим клиентам — например, используя следующий код `printReport()` в `StockAccount`, который использует `StockQuote` (листинг 3.2.8) для загрузки цены каждого вида акций из Интернета.

```
public void printReport()
{
   StdOut.println(name);
   double total = cash;
   for (int i = 0; i < n; i++)
   {
      int amount = shares[i];
      double price = StockQuote.priceOf(stocks[i]);
      total += amount * price;
      StdOut.printf("%4d %5s ", amount, stocks[i]);
      StdOut.printf("%9.2f %11.2f\n", price, amount*price);
   }
   StdOut.printf("%21s %10.2f\n", "Cash: ", cash);
   StdOut.printf("%21s %10.2f\n", "Total:", total);
}
```

Реализации `valueOf()` и `save()` тривиальны (см. упражнение 3.2.22). Для реализаций `buy()` и `sell()` потребуются механизмы, описанные в разделе 4.4, поэтому они откладываются до упражнения 4.4.65.

С одной стороны, этот пример демонстрирует вычисления, которые были одним из главных движущих факторов эволюции компьютерных технологий в 1950-е годы. Банки и другие компании покупали ранние модели компьютеров именно из-за потребности в составлении финансовой отчетности такого рода. Например, механизмы форматирования при выводе разрабатывались именно для таких приложений. С другой стороны, приложение воплощает современные веб-ориентированные вычисления, так как информация загружается прямо из Интернета без использования браузера.

Это приложение можно развивать далее, в том числе обслуживая сразу несколько клиентов брокера. Например, брокер может создать массив всех портфелей акций, а затем обработать список транзакций, изменяющих информацию портфелей и проводящих реальные транзакции по Сети. Конечно, такой код должен разрабатываться весьма аккуратно!

Когда в главе 2 вы научились определять функции, которые могут использоваться в разных местах программы (или других программ), вы тем самым перешли из мира, где программа состоит из простой последовательности команд в одном файле, в мир модульного программирования. Суть этой методологии выражена мантрой, которую вы уже хорошо знаете: *всегда, когда вы можете четко разделить задачи в своей программе, — сделайте это.* Аналогичная возможность для данных, представленная в этой главе, переводит вас из мира с несколькими элементарными типами данных в мир, где вы можете сами определить свои типы данных. Эта принципиально новая возможность сильно расширяет горизонты вашего программирования. Как и в случае с концепцией функции, научившись реализовывать и использовать новые типы данных, вы будете удивляться примитивности программ, в которых эта возможность не используется.

Но объектно-ориентированное программирование вовсе не сводится к структурированию данных. Оно позволяет связать данные, актуальные для конкретной

подзадачи, с операциями, которые работают с этими данными, и выделить их в независимый модуль. В объектно-ориентированном программировании эта мантра выглядит так: *всегда, когда вы можете четко выделить данные и связанные с ними операции в своей программе, — сделайте это.*

Рассмотренные примеры наглядно доказывают, что объектно-ориентированное программирование может быть полезно в самых разных ситуациях. Пытаетесь ли вы спроектировать и построить физическое устройство, разработать программную систему, разобраться в явлении природы или обработать информацию, все начинается с определения подходящей абстракции: геометрического описания физического устройства, модульного плана программной системы, математической модели природного явления или структуры данных для информации. А если вам потребуется написать программы для работы с экземплярами четко определенной абстракции, вы можете с таким же успехом реализовать их в виде типа данных Java и написать код Java для создания и обработки объектов этого типа.

Каждый раз, когда вы разрабатываете класс, который использует другие классы, вы тем самым программируете на более высоком уровне абстракции. В следующем разделе рассматриваются некоторые проблемы, присущие этому стилю программирования.

Вопросы и ответы

В. Переменным экземпляров присваиваются начальные значения по умолчанию, на которые мы можем рассчитывать?

О. Да. Им автоматически присваивается 0 для числовых типов, `false` для типа `boolean` и специальное значение `null` для всех ссылочных типов. Эти значения соответствуют тому, как Java автоматически инициализирует элементы массивов. Автоматическая инициализация гарантирует, что в каждой переменной экземпляра всегда хранится валидное (но не всегда осмысленное) значение. Стоит ли писать код, зависящий от этих значений? На этот счет есть разные мнения: некоторым опытным программистам это нравится, потому что полученный код может быть очень компактным; другие предпочитают не использовать значения по умолчанию, потому что такой код будет непонятен тому, кто не знает эти правила.

В. Что такое `null`?

О. Это специальное предопределенное значение, которое не ссылается ни на какой объект. Использование ссылки `null` для вызова метода бессмысленно и приводит к исключению `NullPointerException`. Часто это происходит из-за того, что вы не инициализировали переменные экземпляров объекта или элементы массива.

В. Могу ли я инициализировать переменную экземпляра при объявлении значением, отличным от значения по умолчанию?

О. Обычно переменные экземпляров инициализируются другими значениями в конструкторе. Тем не менее вы можете указать исходные значения переменных экземпляров при объявлении, по той же схеме, что и при встроенной инициализации локальных переменных. Такая инициализация выполняется до вызова конструктора.

В. Каждый класс должен содержать конструктор?

О. Да, но если конструктор не указан, Java автоматически генерирует конструктор по умолчанию (без аргументов). Когда клиент вызывает этот конструктор ключевым словом `new`, переменные экземпляра автоматически инициализируются обычным способом. Если же конструктор определен, то конструктор по умолчанию (без аргументов) исчезает.

В. Предположим, я не определяю метод `toString()`. Что произойдет, если я попытаюсь вывести объект этого типа вызовом `StdOut.println()`?

О. Будет выведено целое число, которое вряд ли принесет вам хоть какую-нибудь пользу.

В. Можно ли включить статический метод в класс, реализующий тип данных?

О. Конечно. Например, все наши классы содержат метод `main()`. Однако если в классе статические методы смешиваются с методами экземпляров, в коде легко запутаться. Например, статические методы естественно использовать для операций с несколькими объектами, ни один из которых не кажется естественным кандидатом для вызова метода. Например, мы используем запись `z.abs()` для вычисления `|z|`, но запись `a.plus(b)` для вычисления суммы уже не столь естественна. А почему не `b.plus(a)`?

Вместо этого можно определить в `Complex` статический метод следующего вида:

```
public static Complex plus(Complex a, Complex b)
{
   return new Complex(a.re + b.re, a.im + b.im);
}
```

Обычно мы стараемся избегать таких конструкций и предпочитаем обходиться выражениями, не смешивающими статические методы и методы экземпляров, чтобы нам не приходилось писать код следующего вида:

```
z = Complex.plus(Complex.times(z, z), z0)
```

Вместо этого используется запись вида:

```
z = z.times(z).plus(z0)
```

В. Все эти вызовы `plus()` и `times()` выглядят довольно неуклюже. Нельзя ли использовать привычные знаки `+` и `*` в выражениях с объектами, для которых они имеют смысл (например, `Complex` и `Vector`), чтобы вместо вызовов можно было использовать более компактные выражения вида `z = z*z + z0`?

О. Некоторые языки (прежде всего C++ и Python) поддерживают такую возможность, называемую *перегрузкой операторов*, но в Java она отсутствует. Как обычно, это решение создателей языка, с которым нам приходится смириться, но многие программисты Java не считают это большой потерей. Перегрузка операторов оправданна только для типов, представляющих числовые или алгебраические абстракции, а таких типов не так уж много; при этом многие программы проще понять, когда операции имеют содержательные имена вида `plus()` и `times()`. В языке программирования APL в 1970-е годы был выбран обратный подход, доведенный до крайности: язык настаивал, чтобы *каждая* операция была представлена одним знаком (включая буквы греческого алфавита).

В. Существуют ли другие виды переменных, кроме аргументов, локальных переменных и переменных экземпляров в классах?

О. Если указать ключевое слово `static` в объявлении переменной (вне какого-либо метода), будет создана так называемая *статическая переменная* (или *переменная класса*). Статические переменные, как и переменные экземпляров, доступны для всех методов в классе; при этом они не связываются ни с каким объектом — на весь класс существует всего одна такая переменная. В старых языках программирования такие переменные назывались *глобальными* из-за своей глобальной области видимости. В современном программировании область видимости переменных обычно ограничивается, поэтому такие переменные используются редко.

В. Программа `Mandelbrot` создает сотни миллионов объектов `Complex`. Не замедляют ли операции создания объектов ее работу?

О. Замедляют, но не настолько, чтобы помешать построению диаграммы. Наша цель — сделать код удобочитаемым и простым в сопровождении; ограничение области видимости за счет применения абстракции комплексного числа помогает в достижении этой цели. Конечно, работу `Mandelbrot` можно ускорить, отказавшись от абстракции комплексного числа или используя другую реализацию `Complex`.

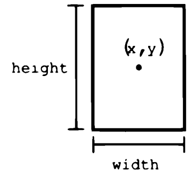
Упражнения

3.2.1. Возьмем следующую реализацию типа данных для прямоугольников, стороны которых параллельны осям; каждый прямоугольник представлен координатами центра, шириной и высотой:

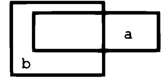
```
public class Rectangle
{
   private final double x, y;    // Центр прямоугольника
   private final double width;   // Ширина прямоугольника
   private final double height;  // Высота прямоугольника
```

```
   public Rectangle(double x0, double y0, double w, double h)
   {
      x = x0;
      y = y0;
      width = w;
      height = h;
   }
   public double area()
   {  return width * height;  }

   public double perimeter()
   {  /* Вычисление периметра. */  }
   public boolean intersects(Rectangle b)
   {  /* Прямоугольник пересекается с b? */  }

   public boolean contains(Rectangle b)
   {  /* Прямоугольник содержит b? */  }

   public void draw(Rectangle b)
   {  /* Рисование прямоугольника средствами стандартной графики. */  }
}
```

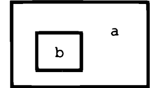
представление



intersects



contains



Напишите API для этого класса и заполните код `perimeter()`, `intersects()` и `contains()`. *Примечание*: два прямоугольника считаются пересекающимися, если они имеют одну или несколько общих точек (нестрогое пересечение). Например, оба выражения `a.intersects(a)` и `a.contains(a)` истинны.

3.2.2. Напишите для `Rectangle` тестовое приложение-клиент, которое получает три аргумента командной строки `n`, `min` и `max`; генерирует в единичном квадрате `n` случайных прямоугольников с шириной и высотой, равномерно распределенной между `min` и `max`; отображает их средствами стандартной графики и выводит среднюю площадь и периметр в стандартный поток вывода.

3.2.3. Добавьте в тестовое приложение-клиент код из предыдущего упражнения для вычисления среднего количества прямоугольников, пересекающих заданный.

3.2.4. Разработайте реализацию API `Rectangle` из упражнения 3.2.1, в которой прямоугольники представляются координатами x и y левого нижнего и правого верхнего углов. API при этом не изменяется.

3.2.5. Что не так в следующем коде?

```
public class Charge
{
   private double rx, ry;   // Позиция
   private double q;        // Заряд
   public Charge(double x0, double y0, double q0)
   {
      double rx = x0;
      double ry = y0;
      double q = q0;
   }
...
}
```

Ответ: команды присваивания в конструкторе также представляют собой объявления, которые создают новые локальные переменные `rx`, `ry` и `q`, выходящие из области видимости при завершении конструктора. Переменные экземпляра `rx`, `r` и `q` сохраняют значение по умолчанию 0.

Примечание: локальная переменная, имя которой совпадает с именем переменной экземпляра, *замещает* переменную экземпляра. В следующем разделе мы рассмотрим возможность обращения к замещенным переменным экземпляров, хотя начинающим программистам лучше избегать этого.

3.2.6. Создайте тип данных `Location` для представления точки земной поверхности, определяемой широтой и долготой. Включите метод `distanceTo()` для вычисления расстояний по дуге большого круга (см. упражнение 1.2.33).

3.2.7. Реализуйте тип данных `Rational` для рациональных чисел. Тип должен поддерживать операции сложения, вычитания, умножения и деления.

`public class Rational`

`Rational(int numerator, int denominator)`	
`Rational plus(Rational b)`	сумма этого числа и b
`Rational minus(Rational b)`	разность этого числа и b
`Rational times(Rational b)`	произведение этого числа и b
`Rational divides(Rational b)`	частное этого числа и b
`String toString()`	строковое представление

Используйте метод `Euclid.gcd()` (листинг 2.3.1) для проверки отсутствия общих делителей у числителя и знаменателя. Напишите тестовое приложение-клиент для проверки всех методов. Не беспокойтесь о проверке целочисленного переполнения (см. упражнение 3.3.17).

3.2.8. Напишите тип данных `Interval`, реализующий следующий API:

`public class Interval`

`Interval(double left, double right)`	
`boolean contains(double x)`	значение x входит в интервал?
`boolean intersects(Interval b)`	текущий интервал пересекается с b?
`String toString()`	строковое представление

Интервал определяется как множество всех точек прямой, больших или равных `left` и меньших или равных `right`. Интервал, у которого `right` меньше `left`, считается пустым. Напишите клиентское приложение-фильтр, которое получает вещественное значение `x` в аргументе командной строки и выводит все интервалы из стандартного потока ввода, содержащие `x` (каждый интервал определяется парой значений типа `double`).

3.2.9. Напишите приложение-клиент для класса `Interval` из предыдущего примера, которое получает аргумент командной строки `n`, читает из стандартного ввода `n` интервалов (каждый интервал определяется парой значений типа `double`) и выводит все пары пересекающихся интервалов.

3.2.10. Разработайте реализацию API `Rectangle` из упражнения 3.2.1, использующую тип данных `Interval` для упрощения кода.

3.2.11. Напишите тип данных `Point`, который реализует следующий API:

public class Point	
`Point(double x, double y)`	
`double distanceTo(Point q)`	евклидово расстояние от текущей точки до q
`String toString()`	строковое представление

3.2.12. Добавьте в `Stopwatch` методы для остановки и перезапуска секундомера.

3.2.13. Используйте `Stopwatch` для сравнения стоимости вычисления гармонических чисел в цикле `for` (см. листинг 1.3.5) и решения с использованием рекурсивного метода из раздела 2.3.

3.2.14. Разработайте версию `Histogram`, использующую `Draw`, чтобы клиент мог создать несколько гистограмм. Добавьте на диаграмму красную вертикальную линию, обозначающую математическое ожидание, и синие вертикальные линии на расстоянии двух среднеквадратичных отклонений от математического ожидания. Используйте тестовое приложение-клиент, которое создает гистограммы для результатов бросков монеты (испытания Бернулли) с несимметричной монетой, у которой «орел» выпадает с вероятностью p, где p = 0,2, 0,4, 0,6 и 0,8. Программа должна получать в командной строке количество бросков и количество испытаний, как в листинге 3.2.3.

3.2.15. Измените тестовое приложение-клиент `Turtle`, чтобы оно получало как аргумент командной строки нечетное целое число n и рисовало n-конечную звезду.

3.2.16. Измените метод `toString()` из класса `Complex` (листинг 3.2.6), чтобы он выводил комплексные числа в традиционном формате. Например, программа должна выводить значение $3 - i$ в виде `3-i` вместо `3.0 + -1.0i`, значение 3 в виде `3` вместо `3.0 + 0.0i`, а значение $3i$ в виде `3i` вместо `0.0 + 3.0i`.

3.2.17. Напишите клиент `Complex`, который получает три вещественных числа a, b и c в аргументах командной строки и выводит два (комплексных) корня $ax^2 + bx + c$.

3.2.18. Напишите программу `Roots` — клиент для `Complex`, который получает из командной строки две значения a и b типа `double` и целое значение n и выводит корни n-й степени из $a + bi$. *Примечание*: если операция извлечения корня из комплексного числа вам незнакома, пропустите это упражнение.

3.2.19. Реализуйте следующие дополнения API `Complex`:

`double theta()`	Фаза (угол) числа
`Complex minus(Complex b)`	Разность этого числа и b
`Complex conjugate()`	Комплексное сопряжение числа
`Complex divides(Complex b)`	Результат деления числа на b
`Complex power(int b)`	Результат возведения числа в b-ю степень

Напишите приложение-клиент, тестирующее все методы.

3.2.20. Предположим, вы хотите добавить в `Complex` конструктор, который получает в аргументе значение `double` и создает число `Complex`. Полученное число определяет действительную часть комплексного числа (мнимая часть отсутствует). Вы пишете следующий код:

```
public void Complex(double real)
{
   re = real;
   im = 0.0;
}
```

но команда `Complex c = new Complex(1.0)` не компилируется. Почему?

Ответ: конструкторы не имеют возвращаемого типа, даже `void`. Этот код определяет метод с именем `Complex`, а не конструктор. Удалите ключевое слово `void`.

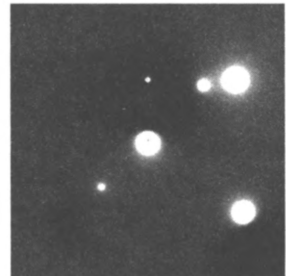
3.2.21. Найдите значение `Complex`, для которого `mand()` возвращает число, большее 100. Увеличьте изображение на этом значении, как в примере в тексте.

3.2.22. Реализуйте методы `valueOf()` и `save()` класса `StockAccount` (листинг 3.2.8).

Упражнения повышенной сложности

3.2.23. *Визуализация потенциалов.* Напишите программу `Potential`, которая создает массив заряженных частиц по значениям, полученным из стандартного потока ввода (каждая заряженная частица определяется координатой x, координатой y и величиной заряда), и строит визуализацию электрического потенциала в единичном квадрате. Для этого выполните дискретизацию квадрата (представьте его в виде матрицы точек) и для каждой точки вычислите электрический потенциал (суммированием электрических потенциалов,

```
% more charges.txt
9
.51 .63 -100
.50 .50   40
.50 .72   10
.33 .33    5
.20 .20  -10
.70 .70   10
.82 .72   20
.85 .23   30
.90 .12  -50
% java Potential < charges.txt
```



Потенциальная визуализация группы зарядов

созданных всеми заряженными частицами) и отобразите точку в оттенке серого, пропорциональном ее потенциалу.

3.2.24. *Изменяющиеся заряды.* Измените программу `Charge` (листинг 3.2.1), чтобы величина заряда `q` не была неизменяемой величиной. Добавьте метод `increaseCharge()`, который получает аргумент `double` и добавляет к заряду заданное значение. Затем напишите программу-клиент, которая инициализирует массив следующими значениями:

```
Charge[] a = new Charge[3];
a[0] = new Charge(0.4, 0.6, 50);
a[1] = new Charge(0.5, 0.5, -5);
a[2] = new Charge(0.6, 0.6, 50);
```

а затем выводит в виде сменяющихся изображений результат постепенного уменьшения заряда `a[i]`, для чего код расчета изображения заключается в цикл следующего вида:

```
for (int t = 0; t < 100; t++)
{
   // Вычисление изображения.
   picture.show();
   a[1].increaseCharge(-2.0);
}
```

Изменение заряда

3.2.25. *Хронометраж для типа* `Complex`. Напишите клиент `Stopwatch`, который сравнивает затраты времени для вычислений в программе `Mandelbrot` при использования `Complex` и при работе непосредственно с двумя значениями `double`. А точнее, создайте версию программы `Mandelbrot`, которая просто выполняет вычисления (удалите код с обращениями к Picture), затем создайте программу без использования `Complex` и сравните отношение времени выполнения двух программ.

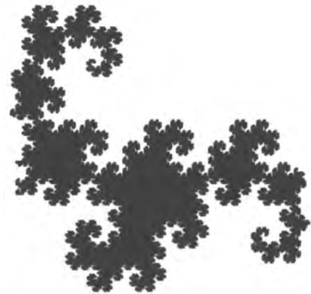
3.2.26. *Кватернионы.* В 1843 году сэр Уильям Гамильтон ввел расширение комплексных чисел — так называемые *кватернионы.* Кватернион представляет собой вектор $a = (a_0, a_1, a_2, a_3)$ со следующими операциями:

- модуль: $|a| = \sqrt{a_0^2 + a_1^2 + a_2^2 + a_3^2}$
- сопряжение: сопряжением a является вектор $(a_0, -a_1, -a_2, -a_3)$

- обратная величина: $a^{-1} = (a_0/|a|^2, -a_1/|a|^2, -a^2/|a|^2, -a_3/|a|^2)$
- сумма: $a + b = (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- произведение: $a \times b = (a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3, a_0b_1 - a_1b_0 + a_2b_3 - a_3b_2, a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1, a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0)$
- частное: $a/b = ab^{-1}$

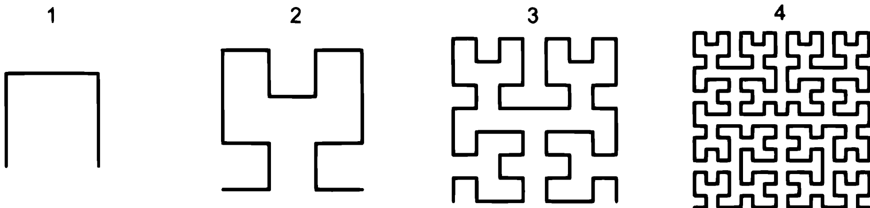
Создайте тип данных `Quaternion` для кватернионов и тестовое приложение-клиент, тестирующее весь код. Кватернионы расширяют концепцию трехмерного поворота в четвертое измерение. Они используются в компьютерной графике, теории управления, обработке сигналов и орбитальной механике.

3.2.27. *Кривая дракона.* Напишите рекурсивную программу `Dragon` — клиент для `Turtle`, который стоит кривые дракона (см. упражнение 1.2.35 и 1.5.9).



Ответ: эти кривые, обнаруженные тремя физиками из NASA, были популяризированы в 1960-е годы Мартином Гарднером, а позднее использовались Майклом Крайтоном в книге и фильме «Парк юрского периода». Это задача может быть решена на удивление компактно, с использованием двух взаимно рекурсивных методов, следующих непосредственно из определения в упражнении 1.2.35. Один метод, `dragon()`, рисует нормальную кривую; другой, `nogard()`, рисует кривую в обратном порядке. За подробностями обращайтесь на сайт книги.

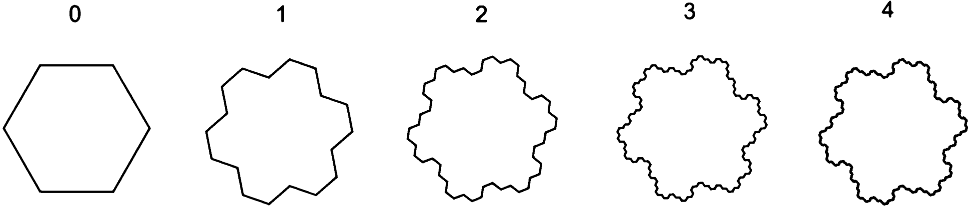
3.2.28. *Кривые Гильберта. Заполняющей кривой* называется непрерывная кривая в единичном квадрате, проходящая через каждую его точку[1]. Напишите рекурсивную программу-клиент `Turtle`, которая строит рекурсивные узоры, представляющие собой заполняющие кривые, определенные математиком Дэвидом Гильбертом в конце XIX века.



[1] Подразумевается дискретизация квадрата — представление его матрицей точек (двумерной «выборкой») с определенным шагом. — *Примеч. науч. ред.*

Подсказка: напишите пару взаимно рекурсивных методов: `hilbert()` и `treblih()` для обхода кривой Гильберта в прямом и обратном порядке. За подробностями обращайтесь на сайт книги.

3.2.29. *Остров Госпера*. Напишите рекурсивную программу-клиент `Turtle` для построения следующих рекурсивных узоров.



3.2.30. *Химические элементы*. Создайте тип данных `ChemicalElement` для элементов Периодической таблицы элементов. Включите в значения типа данных поля для наименования элемента, его атомного номера, условного обозначения и атомного веса, а также методы доступа для каждого из этих значений. Затем создайте тип данных `PeriodicTable`, который читает значения из файла для создания массива объектов `ChemicalElement` (вы найдете файл и описание процесса его построения на сайте книги) и отвечает на запросы, получаемые из стандартного потока ввода: пользователь вводит молекулярную формулу типа H_2O, а программа выдает молекулярный вес. Разработайте API и реализации обоих типов данных.

3.2.31. *Анализ данных*. Напишите тип данных для серии экспериментов: управляющая переменная — целое число в диапазоне `[0, n)`, зависимая переменная — значение типа `double`. (Например, в ходе экспериментов может анализироваться время выполнения программы.) Реализуйте следующий API:

`public class Data`	
`Data(int n, int max)`	создает новый объект анализа данных для n целых значений в диапазоне [0, n)
`double addDataPoint(int i, double x)`	добавляет точку данных (i, x)
`void plotPoints()`	строит график всех точек данных

Используйте статические методы `StdStats` для проведения статического анализа и построения графика. Напишите тестовый клиент, который выводит результаты (вероятность протекания) при проведении экспериментов с `Percolation` с увеличением размера сетки *n*.

3.2.32. *Цены акций*. Файл DJIA.csv на сайте книги содержит данные цен при закрытии торгов в истории индекса Доу-Джонса в формате CSV (текстовые поля,

разделенные запятыми). Создайте тип данных DowJonesEntry для хранения одной записи таблицы, со значениями для даты, цены при открытии, дневного максимума, дневного минимума, цены при закрытии и т. д. Затем создайте тип данных DowJones, который читает данные из файла, создает объекты DowJonesEntry и поддерживает методы для вычисления средних значений за различные периоды времени. Наконец, создайте интересные программы-клиенты DowJones для построения графиков данных. Проявите фантазию: по этой дороге многие ходили до вас.

3.2.33. *Максимальная и минимальная стоимость.* Напишите программу-клиент для StockAccount, который строит массив объектов StockAccount, вычисляет общую стоимость каждого портфеля акций и выводит отчет по портфелям с наибольшей и наименьшей стоимостью. Предполагается, что информация хранится в одном файле, в котором описания портфелей следуют одно за другим — в формате, приведенном в тексте.

3.2.34. *Неопределенность в методе Ньютона.* Многочлен $f(z) = z^4 - 1$ имеет четыре корня: 1, –1, i и $-i$. Для поиска корней можно воспользоваться методом Ньютона на комплексной плоскости: $z_{k+1} = z_k - f(z_k) / f'(z_k)$. Здесь $f(z) = z^4 - 1$, а $f'(z) = 4z^3$. Метод сходится к одному из четырех корней в зависимости от выбора начальной точки z_0. Напишите программу NewtonChaos — клиент Complex и Picture, которая получает как аргумент командной строки число n и создает изображение $n \times n$ пикселов, соответствующее квадрату со стороной 2 и центром в начале координат. Окрасьте каждый пиксел в белый, красный, зеленый или синий цвет в зависимости от того, к какому из четырех корней сходится соответствующее комплексное число (или в черный, если после 100 итераций схождение не обнаружено).

3.2.35. *Цветное отображение множества Мандельброта.* Создайте файл из 256 троек целых чисел, представляющих желаемые значения Color, а затем используйте эти цвета вместо оттенков серого для рисования всех пикселов Mandelbrot. Прочитайте значения и создайте массив из 256 значений Color, индексируйте элемент массива возвращаемым значением mand(). Экспериментируя с разным расположением цветов в множестве, можно получить потрясающие изображения. Пример представлен в файле mandel.txt на сайте книги.

3.2.36. *Множества Жюлиа.* Множество Жюлиа для заданного комплексного числа c представляет собой множество точек, связанных с функцией Мандельброта. Вместо того чтобы фиксировать z и изменять c, мы фиксируем c и изменяем z. Эти точки z, для которых измененная функция Мандельброта остается ограниченной, входят в множество Жюлиа; точки, для которых последовательность уходит в бесконечность, в множество не входят. Все точки z, представляющие интерес, лежат в квадрате 4×4 с центром в начале координат. Множество Жюлиа для c является связным в том и только в том случае, если c входит в множество Мандельброта! Напишите программу ColorJulia, которая получает два аргумента командной строки a и b и строит цветное отображение множества Жюлиа для $c = a + bi$, используя метод таблицы цветов из предыдущего упражнения.

3.3. Проектирование типов данных

Возможность создания типов данных превращает каждого программиста в разработчика языка. Вам не приходится ограничиваться типами данных и операциями, встроенными в язык, потому что вы можете создавать собственные типы данных и писать клиентские программы, которые их используют. Например, в Java нет заранее определенного типа данных для комплексных чисел, но вы можете определить тип `Complex` и писать клиентские программы, такие как `Mandelbrot`. Аналогичным образом в Java нет встроенной поддержки графики с командами относительного перемещения, но вы можете определить класс `Turtle` и написать программу, использующую эту абстракцию. Даже когда в Java уже предусмотрена какая-либо конкретная возможность, может оказаться, что вы предпочтете создать собственный тип данных, лучше приспособленный для ваших потребностей, как произошло с `Picture`, `In`, `Out` и `Draw`.

Первое, к чему необходимо стремиться при создании программы, — это понимание того, какие типы данных вам нужны. Развитие этого понимания — одна из составляющих процесса *проектирования*. В этом разделе мы сосредоточимся на разработке API как на критически важном шаге разработки любой программы. Вы должны рассмотреть разные альтернативы, понять, как они отражаются на клиентских программах и реализациях, и уточнить решение для того, чтобы выдержать баланс между потребностями клиентов и возможными стратегиями реализации.

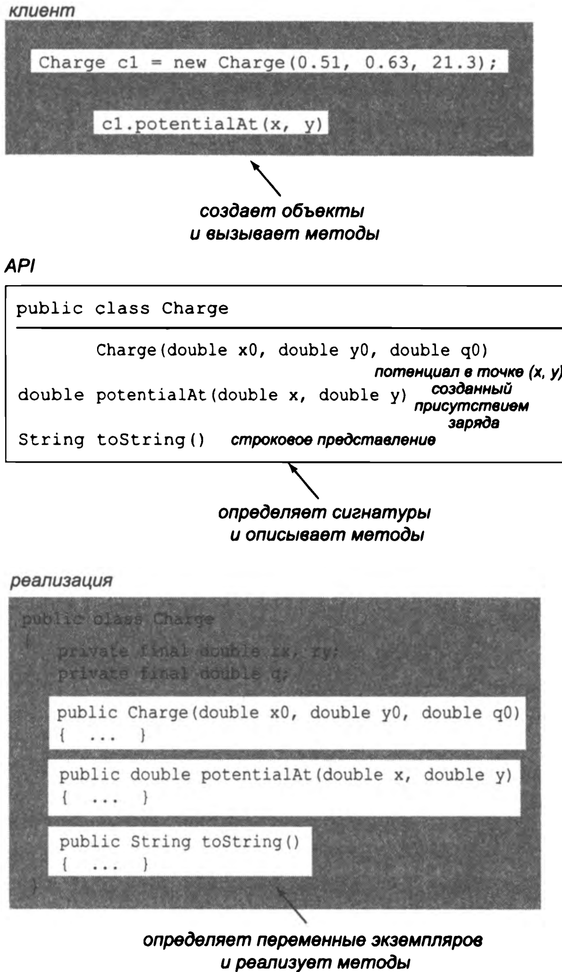
Если вы займетесь изучением программирования сложных систем, вы узнаете, что этот аспект проектирования очень важен при построении сложных систем, а в Java и других языках существуют мощные высокоуровневые механизмы, обеспечивающие повторное использование кода при написании больших программ. Многие из этих механизмов предназначены для специалистов по созданию больших систем, но общий подход полезно знать каждому программисту, а некоторые из этих механизмов могут пригодиться и при написании небольших программ.

В этом разделе рассматриваются концепции *инкапсуляции*, *неизменяемости* и *наследования*; особое внимание будет уделено применению этих механизмов в проектировании типов данных для применения модульного программирования, упрощения отладки и написания четкого, правильно работающего кода.

В конце этого раздела будут описаны механизмы Java, применяемые для проверки соответствия предположений разработчика с фактическими условиями во время выполнения. Такие инструменты оказывают неоценимую помощь в разработке надежных программных продуктов.

Проектирование API

В разделе 3.1 мы писали клиентские программы, которые использовали API; в разделе 3.2 мы занялись реализацией API. Теперь пора перейти к задаче проектирования API. Рассматривать эти темы именно в таком порядке и под таким



Объектно-ориентированная абстракция библиотеки

углом правильно, потому что большую часть времени, которое вы проводите за программированием, вы будете писать клиентские программы.

Часто самым важным и сложным шагом при построении программного продукта становится проектирование API. Эта работа требует опыта, тщательного обдумывания и многократных доработок. Тем не менее время, затраченное на проектирование качественного API, наверняка окупится временем, сэкономленным в процессе отладки или при повторном использовании кода.

Может показаться, что при написании небольшой программы формулировать API — это перебор, но вы должны подходить к написанию любой программы так, словно вы собираетесь когда-нибудь повторно использовать этот код, — и даже

не потому, что вы действительно будете повторно использовать *именно этот* код. Просто необходимость в повторном использовании возникнет почти наверняка, и вы еще не знаете, *какой именно* код вам понадобится.

Стандарты

Чтобы понять, почему так важно сформулировать API, достаточно обратиться к другим областям. От железнодорожных рельсов, болтов и гаек с резьбой до MP3, радиочастот и интернет-стандартов, наличие общего стандартного интерфейса обеспечивает для технологии максимально широкое применение. Примером служит даже сам язык Java: ваши программы являются клиентами виртуальной машины Java, которая предоставляет стандартный интерфейс, реализованный на самых разных аппаратных и программных платформах. Применяя API для отделения клиентов от реализации, мы пользуемся преимуществами стандартных интерфейсов в каждой написанной программе.

Проблема спецификации

API представляет собой список методов, снабженных коротким описанием того, что эти методы должны делать. В идеале API должен четко описывать поведение для разных вариантов входных данных, включая побочные эффекты, после чего некий программный инструмент проверял бы, что реализация соответствует спецификации. К сожалению, фундаментальный вывод из теории обработки данных, называемый *проблемой спецификации*, утверждает, что эта цель недостижима. В двух словах, такая спецификация должна быть записана на формальном языке, сходном с языком программирования, а задача определения того, выполняют ли две программы одинаковые вычисления, нерешаема. (Если вас интересует этот момент, вы можете узнать намного больше о природе нерешаемых задач и их роли в природе вычислений в курсе теоретической информатики.) По этой причине в своих примерах мы используем неформальные описания вроде тех, что приведены в тексте, сопровождающем API.

Широкие интерфейсы

Широким называют интерфейс, который содержит слишком большое количество методов. Важный принцип, который следует соблюдать при проектировании API, — нежелательность широких интерфейсов. Со временем API обычно естественным образом разрастается, потому что добавить методы в существующий API легко, а исключить их без нарушения работоспособности существующих клиентов трудно. В некоторых ситуациях широкие интерфейсы оправданны — например, в часто используемых системных библиотеках (таких, как `String`). Существуют различные приемы для сокращения фактической ширины интерфейсов. Например, можно включать в интерфейс только методы, ортогональные по функциональности. Так, библиотека Java `Math` включает тригонометрические функции для синуса, косинуса и тангенса, но не для секанса и косеканса.

Начните с клиентского кода

Одна из основных целей разработки типа данных — упрощение клиентского кода. А это означает, что о клиентском коде стоит задумываться с самого начала. Часто бывает разумно написать клиентский код до того, как начинать работу над реализацией. Когда вы сталкиваетесь с клиентским кодом, который становится громоздким и неудобным, попробуйте написать упрощенную версию, реализующую вычисления так, как вы их себе представляете. Или, если вам удалось написать содержательные и компактные комментарии для описания ваших вычислений, можно поискать возможности для преобразования комментариев в код.

Избегайте зависимости от представления

Обычно при проектировании API мы ориентируемся на представление. В конце концов, тип данных — это множество значений и множество операций с этими значениями, и говорить об операциях, не зная значений, довольно-таки бессмысленно. Однако это не означает, что вы должны знать *представление* этих значений. Среди прочего, типы данных призваны упрощать клиентский код, избавляя его от подробностей и зависимости от конкретного представления. Например, наши клиентские программы для `Picture` и `StdAudio` работают с простыми абстрактными представлениями изображения и звука соответственно. Основная ценность API таких абстракций заключается в том, что они ограждают клиента от многих технических подробностей, присутствующих в стандартных представлениях этих абстракций.

Типичные ошибки при проектировании API

API может быть слишком сложным в реализации; это бывает, когда разработка реализаций предельно сложна или если реализациями слишком трудно пользоваться, и клиентский код получается более сложным, чем без этого API. API может оказаться слишком узким (в него не входят методы, нужные клиенту) или слишком широким (содержащим множество методов, не нужных клиентам). API может быть *слишком общим*, не содержащим полезных абстракций, или *слишком конкретным*, предоставляющим абстракции настолько подробные или настолько мелкие, что они становятся практически бесполезными. Иногда эти проблемы обобщаются еще в одном девизе: *предоставляйте клиентам только те методы, которые им необходимы, и ничего более.*

Когда вы только начинали программировать, вы ввели программу HelloWorld.java, мало что понимая в ней, кроме конечного результата. С этого момента вы учились программировать, сначала только воспроизводя код, приведенный в книге, а затем и создавая собственный для решения различных задач. Сейчас вы находитесь приблизительно в той же стадии освоения проектирования API. В книге, на сайте книги и в электронной документации Java приведены многочисленные примеры API, которые вы можете изучать и использовать. С ними вы сможете приобрести опыт проектирования и разработки собственных API.

Инкапсуляция

Процесс отделения клиентов от реализаций за счет сокрытия информации называется *инкапсуляцией*. Подробности реализации скрываются от клиентов, а реализации ничего не знают о подробностях клиентского кода, который может быть написан когда-нибудь в будущем.

Как вы, возможно, предположили, мы уже применяли инкапсуляцию в своих реализациях типов данных. Раздел 3.1 начинался с мантры: *чтобы использовать тип данных, вам не обязательно знать, как он реализован.* Это утверждение описывает одно из главных преимуществ инкапсуляции. Мы считаем его настолько важным, что не стали описывать другие методы проектирования типа данных. А теперь мы подробно рассмотрим еще три причины для применения инкапсуляции. Инкапсуляция применяется для следующих целей.

- Чтобы сделать возможным модульное программирование.
- Чтобы упростить отладку.
- Чтобы сделать код программы более понятным.

Эти три причины связаны между собой: хорошо спроектированный модульный код проще понять и отладить, чем код, основанный исключительно на примитивных типах в длинных программах.

Модульное программирование

Стиль программирования, который мы развивали с главы 2, был основан на идее разбиения больших программ на меньшие модули, которые можно разрабатывать и отлаживать независимо друг от друга. Этот подход повышает гибкость кода за счет ограничения и локализации последствий вносимых изменений, а также способствует повторному использованию кода, делая возможной замену реализаций типа данных для улучшения быстродействия, точности или затрат памяти. Этот принцип работает во многих ситуациях. Мы часто пользуемся преимуществами инкапсуляции при использовании системных библиотек. Новые версии системы Java часто включают новые реализации различных типов данных, но API при этом не изменяется. У разработчика появляется сильный и постоянный мотив для совершенствования реализаций типа данных, потому что улучшение реализации может принести пользу всем клиентам. Ключом к успеху модульного программирования является обеспечение *независимости* модулей. Для этого требуется, чтобы API был единственной точкой зависимости между клиентом и реализации. *Чтобы использовать тип данных, вам не обязательно знать, как он реализован.* Однако у этого утверждения есть и другая сторона: реализация типа данных может предполагать, что клиент не знает о типе данных ничего, кроме API.

Пример

Для примера возьмем программу `Complex` (листинг 3.3.1). Она использует то же имя и API, что и листинг 3.2.6, но с другим представлением комплексных чисел.

В листинге 3.2.6 используется алгебраическое (декартово) представление, в котором переменные экземпляров x и y представляют комплексное число $x + iy$. В листинге 3.3.1 используется *тригонометрическое представление* (в *полярных координатах*), в котором переменные экземпляров `r` и `theta` представляют комплексное число в форме $r(\cos\theta + i \sin\theta)$. Тригонометрическое представление интересно тем, что некоторые операции с комплексными числами (такие, как умножение и деление) в такой форме выполняются более эффективно[1]. Инкапсуляция позволяет заменить любую из этих реализаций другой (по любой причине) без изменения клиентского кода. Выбор между двумя реализациями зависит от клиента. В самом деле, теоретически единственным внешним проявлением замены для клиентов может стать изменение производительности. Инкапсуляция чрезвычайно важна по многим причинам, и одна из наиболее важных среди них заключается в том, что инкапсуляция позволяет постоянно совершенствовать программный код: если вы разработаете более качественный способ реализации типа данных, от этого выиграют все клиенты. Вы пользуетесь этим свойством каждый раз, когда устанавливаете новую версию какой-либо программной системы, включая ту же систему Java.

Ключевое слово private

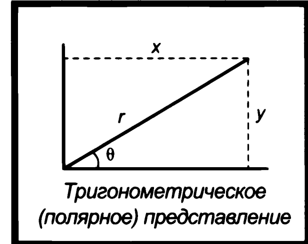
В языке Java поддержка инкапсуляции воплощена в модификаторе доступа `private`. Объявляя закрытую переменную (или метод) экземпляра, вы тем самым запрещаете любому клиенту (коду другого класса) напрямую обращаться к этой переменной (или методу). Клиенты могут обращаться к типу данных только через открытые методы и конструкторы, то есть через API. Соответственно вы можете изменить реализацию так, чтобы она использовала другие переменные экземпляров (или изменить структуру закрытого метода экземпляра), зная, что эти изменения не отразятся напрямую на клиентах. Java не требует, чтобы все переменные экземпляров были закрытыми, но в программах, приведенных в книге, мы будем следовать этому правилу. Например, если переменные экземпляров `re` и `im` класса `Complex` (листинг 3.2.6) будут открытыми, то клиент сможет написать код, который будет обращаться к ним напрямую. Если переменная `z` содержит ссылку на объект `Complex`, то выражения `z.re` и `z.im` обращаются к этим значениям. Но любой клиентский код, в котором используется такая возможность, становится зависимым от этой реализации, что полностью противоречит основной идее инкапсуляции. При переходе на другую реализацию — например, на описанную в программе 3.3.1 — такой код перестанет работать. Чтобы защититься от подобных неприятностей, мы всегда объявляем переменные экземпляров с ключевым словом `private`. Проиллюстрируем это правило.

[1] Существует также *экспоненциальное* представление комплексного числа, тоже эффективное для операций умножения и деления. — *Примеч. науч. ред.*

Листинг 3.3.1. Комплексные числа (альтернативная реализация)

```
public class Complex
{
   private final double r;
   private final double theta;

   public Complex(double re, double im)
   {
      r = Math.sqrt(re*re + im*im);
      theta = Math.atan2(im, re);
   }

   public Complex plus(Complex b)
   {  // Возвращает сумму текущего числа и b.
      double real = re() + b.re();
      double imag = im() + b.im();
      return new Complex(real, imag);
   }

   public Complex times(Complex b)
   {  // Возвращает произведение текущего числа и b.
      double radius = r * b.r;
      double angle  = theta + b.theta;
      // См. "Вопросы и ответы"
   }

   public double abs()
   {  return r;  }

   public double re()  { return r * Math.cos(theta); }
   public double im()  { return r * Math.sin(theta); }

   public String toString()
   {  return re() + " + " + im() + "i";  }

   public static void main(String[] args)
   {
      Complex z0 = new Complex(1.0, 1.0);
      Complex z = z0;
      z = z.times(z).plus(z0);
      z = z.times(z).plus(z0);
      StdOut.println(z);
   }
}
```

`r`	радиус
`theta`	угол



Тригонометрическое (полярное) представление

Тип данных реализует тот же API, что и программа из листинга 3.2.6. Он использует те же методы экземпляров, но с другими переменными экземпляров. Так как переменные экземпляров объявлены закрытыми, эта программа может использоваться вместо листинга 3.2.6 без модификации клиентского кода.

```
% java Complex
-7.000000000000002 + 7.000000000000003i
```

Планирование на будущее

Можно вспомнить немало примеров того, как недостаточная инкапсуляция типов данных приводила к значительным затратам.

- *Проблема 2000 года.* В предыдущем тысячелетии во многих программах год представлялся только двумя цифрами для экономии памяти. Такие программы не отличали 1900 год от 2000. С приближением 1 января 2000 года программистам пришлось спешно исправлять такие ошибки для предотвращения катастрофических последствий, предсказанных многими технологами.
- *Почтовые индексы.* В 1963 году почтовая служба США стала использовать почтовые индексы из 5 цифр для упрощения сортировки и доставки почты. Программисты писали программы, которые предполагали, что индексы всегда будут состоять из 5 цифр, и представляли их в своих программах одним 32-разрядным целым числом. В 1983 году был введен расширенный формат почтовых индексов, который состоял из исходного значения из 5 цифр и 4 дополнительных цифр.
- *IPv4 и IPv6.* Протокол IP (Internet Protocol) является стандартом для передачи данных между устройствами в сети Интернет. Каждому устройству присваивается уникальный адрес. Версия IPv4 использует 32-разрядные адреса и поддерживает до 4,3 миллиарда возможных адресов. Из-за бурного роста Интернета появилась новая версия IPv6, которая использует 128-разрядные адреса и поддерживает 2^{128} адресов.

В каждом из этих случаев необходимое изменение внутреннего представления приводило к тому, что большой объем клиентского кода, зависевшего от текущего стандарта (потому что тип данных не был инкапсулирован), попросту не работал так, как предполагалось[1]. Приблизительные затраты на внесение изменений в каждом из этих случаев исчислялись сотнями миллионов долларов! Отсутствие инкапсуляции всего одного числа обошлось очень дорого. А если вы решите, что все эти поучительные истории далеки от вас, учтите: каждый программист (и вы тоже), не применяющий инкапсуляцию для защиты данных, рискует значительными потерями времени и усилий на исправление неработоспособного кода в случае изменения стандартов.

Мы определяем все переменные экземпляров с модификатором доступа `private` — это ограничение предоставляет некоторую защиту от таких проблем. Применяя это правило в реализации типа данных для года, почтового индекса, IP-адреса и т. д., вы сможете изменять представление без ущерба для клиента. *Реализация типа данных* знает используемое представление, а *объект* хранит данные; *клиент* получает только ссылку на объект и не знает никаких подробностей.

[1] В случае с протоколами IP инкапсуляция реализована на уровне API системы и библиотек сетевых функций, но она может выглядеть не так, как свойственно языкам ООП (в том числе Java). Вообще сетевые протоколы — хороший пример модульного подхода к проектированию сложной программной системы. — *Примеч. науч. ред.*

Снижение риска ошибок

Инкапсуляция также помогает программистам обеспечить правильность работы их кода. В качестве примера можно привести еще одну известную историю: на президентских выборах 2000 года Альберт Гор получил –16 022 голоса на электронной машине для голосования в округе Волузия (штат Флорида). Переменная-счетчик неправильно инициализировалась в программном обеспечении машины! Чтобы понять суть проблемы, рассмотрим класс `Counter` (листинг 3.3.2), реализующий простой счетчик в соответствии со следующим API:

public class Counter		
	`Counter(String id, int max)`	создает счетчик и инициализирует его 0
`void`	`increment()`	увеличивает счетчик, пока его значение не достигнет max
`int`	`value()`	возвращает значение счетчика
`String`	`toString()`	строковое представление

API типа данных счетчика (листинг 3.3.2)

Эта абстракция может использоваться в разных контекстах, в том числе и на электронных машинах для голосования. Она инкапсулирует одно целое число и гарантирует, что с этим числом может выполняться только одна операция — *увеличение на 1*. Следовательно, счетчик ни при каких условиях не станет отрицательным. Целью абстракции данных является ограничение операций с данными. Кроме того, она обеспечивает *изоляцию* операций с данными. Например, можно добавить новую реализацию с поддержкой ведения журнала, чтобы метод `increment()` сохранял временную метку каждого голоса, или другую информацию, которая могла бы использоваться для проверки целостности данных. Но без модификатора `private` где-то в коде машины для голосования может присутствовать команда вида:

```
Counter c = new Counter("Volusia", VOTERS_IN_VOLUSIA_COUNTY);
c.count = -16022;
```

С модификатором `private` этот код не будет компилироваться; без него счетчик голосов может быть инициализирован отрицательным значением. Возможно, инкапсуляция не обеспечивает полного решения всех проблем безопасности голосования, но она станет хорошей отправной точкой.

Ясность кода

Точное описание типа данных также является признаком качественного проектирования, потому что клиентский код более ясно выражает выполняемые вычисления. Многочисленные примеры такого кода встречались в разделах 3.1 и 3.2; кроме того, проблема упоминалась в обсуждении `Histogram` (листинг 3.2.3). Клиенты этой программы более понятны, чем код, не использующий ее, потому что вызов метода экземпляра `addDataPoint()` четко говорит, что требуется клиенту. Код, написанный на основе правильно выбранных абстракций, практически является

Листинг 3.3.2. Счетчик

```
public class Counter
{
   private final String name;
   private final int maxCount;
   private int count;

   public Counter(String id, int max)
   {  name = id; maxCount = max;  }

   public void increment()
   {  if (count < maxCount) count++;  }

   public int value()
   {  return count;  }

   public String toString()
   {  return name + ": " + count;  }

   public static void main(String[] args)
   {
     int n = Integer.parseInt(args[0]);
     int trials = Integer.parseInt(args[1]);
     Counter[] hits = new Counter[n];
     for (int i = 0; i < n; i++)
        hits[i] = new Counter(i + „", trials);
     for (int t = 0; t < trials; t++)
        hits[StdRandom.uniform(n)].increment();
     for (int i = 0; i < n; i++)
        StdOut.println(hits[i]);
   }
}
```

name	имя счетчика
maxCount	максимальное значение
count	текущее значение

Класс инкапсулирует простой целочисленный счетчик: он присваивает ему строковое имя и инициализирует 0 (инициализация по умолчанию в языке Java). Счетчик увеличивается каждый раз, когда клиент вызывает increment(), выдает текущее значение при вызове value() и создает строку с именем и текущим значением при вызове toString().

```
% java Counter 6 600000
0: 100684
1: 99258
2: 100119
3: 100054
4: 99844
5: 100037
```

самодокументируемым. Некоторые сторонники объектно-ориентированного программирования могут возразить, что сам код `Histogram` станет более понятным, если в нем будет использоваться класс `Counter` (см. упражнение 3.3.3), но, пожалуй, это утверждение спорно.

Преимущества инкапсуляции неоднократно подчеркивались в книге. Сейчас мы снова кратко повторим их в контексте проектирования типов данных. Инкапсуляция делает возможным модульное программирование, позволяя нам:

- независимо разрабатывать код клиента и код реализации;
- совершенствовать реализации без вреда для клиента;
- поддерживать еще не написанные программы (клиент пишется для API).

Инкапсуляция также изолирует операции с типами данных, благодаря чему:

- вы можете добавлять в реализации проверки целостности и другие средства отладки;
- клиентский код становится более ясным.

Правильно реализованный тип данных (с инкапсуляцией) расширяет язык Java и может использоваться любой клиентской программой.

Неизменность

Как упоминалось в конце раздела 3.1, объект некоторого типа данных называется *неизменяемым*, если его значение не может быть изменено после создания. Неизменяемым типом данных называется тип, все объекты которого являются неизменяемыми. Напротив, изменяемым типом данных называется тип, значения объектов которого могут изменяться. Из типов данных, представленных в этой главе, `String`, `Charge`, `Color` и `Complex` являются неизменяемыми, а `Turtle`, `Picture`, `Histogram`, `StockAccount` и `Counter` — изменяемыми. Стоит ли делать тип данных неизменяемым? Это важное решение из области проектирования, которое зависит от конкретной ситуации.

Неизменяемые типы

Многие типы данных предназначены для инкапсуляции значений, которые не изменяются, и своим поведением напоминают примитивные типы. Например, для программиста, реализующего клиента `Complex`, разумно ожидать, что команда `z = z0` для двух переменных `Complex` будет работать так же, как и для переменных типа `double` или `int`. Но если бы объекты `Complex` были изменяемыми, а *после* присваивания `z = z0` значение `z` вдруг изменилось, то оказалось бы, что значение `z0` *тоже* изменилось (обе переменные являются ссылками на один и тот же объект!). Этот неожиданный результат, известный как *совмещение имен*, нередко преподносит неприятные сюрпризы новичкам в объектно-ориентированном программировании. Очень важная причина для реализации неизменяемых типов заключается в том, что неизменяемые типы можно использовать в командах присваивания (или в аргументах и возвращаемых значениях методов), не беспокоясь о непредвиденном изменении их значений.

неизменяемые	изменяемые
String	Turtle
Charge	Picture
Color	Histogram
Complex	StockAccount
Vector	Counter
	Массивы Java

Изменяемые типы

Для многих типов данных главной целью абстракции является инкапсуляция значений при их изменении. Типичным примером служит программа Turtle (листинг 3.2.4). Мы используем Turtle для того, чтобы избавить клиента от обязанности следить за изменениями значений. Аналогичным образом мы ожидаем, что у типов данных Picture, Histogram, StockAccount и Counter, а также у массивов Java значения будут изменяться. Передавая Turtle методу в качестве аргумента (как в программе Koch), мы ожидаем, что значение объекта Turtle будет изменяться.

Массивы и строки

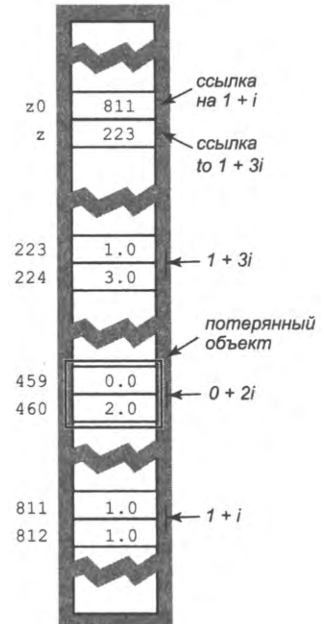
Различие между изменяемыми и неизменяемыми объектами уже встречалось вам при написании клиентского кода, когда вы использовали массивы Java (изменяемые) и тип данных Java String (неизменяемый). Когда вы передаете String методу, вам не нужно беспокоиться о том, что метод изменит исходную цепочку символов String; с другой стороны, при передаче методу массива метод может изменять значения элементов в массиве. Тип данных String является неизменяемым, потому что обычно изменение строковых значений *нежелательно*, а массивы Java являются изменяемыми, потому что значения элементов массива обычно *должны* изменяться. При этом в некоторых ситуациях желательно иметь изменяемые аналоги строк (для этого предназначен тип данных Java StringBuilder) и неизменяемые аналоги массивов (для этого предназначен тип данных Vector, который будет рассмотрен далее в этом разделе).

Преимущества неизменяемости

Как правило, неизменяемые типы проще в использовании и создают меньше риска для ошибок, потому что область видимости кода, способного влиять на их значения, намного меньше, чем для изменяемых типов. Код, использующий неизменяемые типы, проще отлаживается — разработчику проще следить за тем, что переменные клиентского кода, использующие эти типы, сохраняют корректные значения. При использовании изменяемых типов всегда приходится думать о том, где и как могут изменяться их значения.

Цена неизменяемости

Оборотная сторона неизменяемости — необходимость создания нового объекта для каждого значения. Например, выражение `z = z.times(z).plus(z0)` требует создания нового объекта (возвращаемого значения `z.times(z)`), который используется для вызова `plus()`, но ссылка на этот объект нигде не сохраняется. В такой программе, как `Mandelbrot` (листинг 3.2.7), количество промежуточных потерянных объектов может быть достаточно большим. Однако эти затраты обычно не создают проблем, потому что уборщики мусора Java оптимизированы для таких ситуаций. К тому же если программа предназначена для создания большого количества значений (как в случае с `Mandelbrot`), разработчик уже должен быть готов к затратам на их представление. Кроме объектов `Complex`, программа `Mandelbrot` создает также множество (неизменяемых) объектов `Color`.



Промежуточный потерянный объект

Ключевое слово final

Для обеспечения неизменяемости типа данных используется ключевое слово `final`. Объявляя переменную экземпляра с ключевым словом `final`, вы гарантируете, что значение ей будет присвоено только один раз — либо при инициализации по умолчанию, либо в конструкторе. Любые другие попытки изменения значения `final`-переменной приведут к ошибке компиляции. В своем коде мы используем модификатор `final` для переменных экземпляров, значения которых никогда не изменяются. Это правило дополнительно документирует неизменность значения, предотвращает случайные изменения и упрощает отладку программ. Так, переменные с ключевым словом `final` не нужно включать в трассировку, поскольку вы знаете, что их значение никогда не изменяется.

Ссылочные типы

К сожалению, ключевое слово `final` гарантирует неизменяемость только в том случае, если переменные экземпляров относятся к примитивным, а не к ссылочным типам. Если переменная экземпляра ссылочного типа имеет модификатор `final`, то неизменность значения этой переменной (ссылки на объект) будет означать лишь то, что переменная всегда указывает на один и тот же объект. Однако при этом может изменяться значение самого объекта. Например, если создать переменную экземпляра для массива с ключевым словом `final`, изменение самого массива

(скажем, его длины или типа) невозможно, но при этом вы можете изменять значения отдельных элементов. Таким образом, в программах проявляются ошибки совмещения имен. Например, в следующем коде неизменяемость типа данных *не* реализуется:

```
public class Vector
{
   private final double[] coords;
   public Vector(double[] a)
   {
      coords = a;
   }
   ...
}
```

Клиентская программа может создать объект `Vector`, задав значения элементов массива, а затем (в обход API) изменить значения `Vector` после создания:

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 17.0;   // coords[0] теперь содержит 17.0
```

Переменная экземпляра `coords[]` объявлена с ключевыми словами `private` и `final`, но сам объект `Vector` может изменяться, потому что клиент хранит ссылку на *тот же* массив. Когда клиент изменяет значение элемента в своем массиве, это изменение также отражается на соответствующем массиве `coords[]` из-за совмещения `coords[]` и `a[]`. Чтобы гарантировать неизменяемость типа данных, включающего переменную экземпляра изменяемого типа, необходимо создать его локальную копию, которая называется *«защитной»* копией. Пример такой реализации приводится ниже.

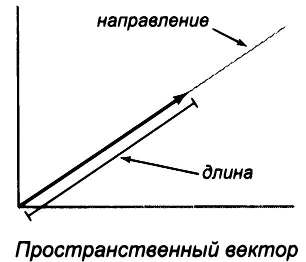
Проблема неизменяемости должна учитываться при проектировании любых типов данных. В идеале неизменяемость типа данных должна быть объявлена в API, чтобы клиенты знали, что значения объекта не будут изменяться. Реализация неизменяемого типа данных усложняется присутствием ссылочных типов. Для сложных типов данных приходится, во-первых, создавать «защитную» копию, а во-вторых, следить за тем, чтобы значения не изменялись никакими методами экземпляров.

Пример: пространственные векторы

Чтобы продемонстрировать применение этих идей в контексте полезной математической абстракции, рассмотрим тип данных для представления *вектора*. Как и в случае с комплексными числами, базовое определение абстракции вектора выглядит достаточно знакомо, поскольку оно занимает центральное место в прикладной математике уже более 100 лет. Область математики, называемая *линейной алгеброй*, занимается изучением свойств векторов. Линейная алгебра — плодородная и успешная теоретическая дисциплина, которая имеет многочисленные прак-

тические применения и играет важную роль во всех областях общественных и естественных наук. Разумеется, полный курс линейной алгебры выходит за рамки книги, но ряд важных практических применений базируется на элементарных и хорошо известных вычислениях, поэтому в книге затрагиваются некоторые аспекты векторов и линейной алгебры (скажем, пример со случайными блужданиями из раздела 1.6 основан на линейной алгебре). А это означает, что такую абстракцию будет полезно инкапсулировать в типе данных.

Пространственным вектором называется абстрактная сущность, обладающая *длиной* и *направлением*. Пространственные векторы предоставляют естественный способ для описания таких понятий физического мира, как сила, скорость, импульс и ускорение. Одним из стандартных способов определения вектора является направленный отрезок от начала координат до точки в декартовой системе координат: направление определяется углом отрезка к оси, а длина — длиной отрезка (расстоянием от начала координат до точки). Таким образом, для определения вектора достаточно определить одну точку (а вектор определяет точку в пространстве).



Пространственный вектор

Эта концепция расширяется до любого количества измерений: вектор в n-мерном пространстве задается серией из n вещественных чисел (координат n-мерной точки). Мы будем обозначать векторы буквой, выделенной жирным шрифтом, а его значение — индексированными именами переменных (та же буква, записанная курсивом), разделенными запятыми и заключенными в круглые скобки. Например, $\mathbf{x}$ обозначает вектор $(x_0, x_1, ..., x_{n-1})$, а $\mathbf{y}$ — вектор $(y_0, y_1, ..., y_{n-1})$.

API

Основные операции с векторами — сложение, масштабирование, вычисление скалярного произведения, вычисление длины и направления:

- *сложение*: $\mathbf{x} + \mathbf{y} = (x_0 + y_0, x_1 + y_1, ..., x_{n-1} + y_{n-1})$
- *масштабирование*: $\alpha\mathbf{x} = (\alpha x_0, \alpha x_1, ..., \alpha x_{n-1})$
- *скалярное произведение*: $\mathbf{x} \bullet \mathbf{y} = x_0 y_0 + x_1 y_1 + ... + x_{n-1} y_{n-1}$
- *длина*: $|\mathbf{x}| = (x_0{}^2, x_1{}^2, ..., x_{n-1}{}^2)$
- *направление*: $\mathbf{x}/|\mathbf{x}| = (x_0/|\mathbf{x}|, x_1/|\mathbf{x}|, ..., x_{n-1}/|\mathbf{x}|)$

Результатами операций сложения, масштабирования и направления являются векторы, а длина и скалярное производительность представляют собой скалярные величины (вещественные числа). Например, если $\mathbf{x} = (0, 3, 4, 0)$, а $\mathbf{y} = (0, -3, 1, -4)$, то $\mathbf{x} + \mathbf{y} = (0, 0, 5, -4)$, $3\mathbf{x} = (0, 9, 12, 0)$, $\mathbf{x} \bullet \mathbf{y} = -5$, $|\mathbf{x}| = 5$, а $\mathbf{x}/|\mathbf{x}| = (0, 3/5, 4/5, 0)$. Вектор направления является единичным вектором: его длина равна 1. Эти определения непосредственно приводят к следующему API.

public class Vector	
`Vector(double[] a)`	создает вектор для заданных декартовых координат
`Vector plus(Vector that)`	сумма текущего вектора и that
`Vector minus(Vector that)`	разность текущего вектора и that
`Vector scale(double alpha)`	текущий вектор, масштабированный с коэффициентом alpha
`double dot(Vector b)`	скалярное произведение этого вектора и that
`double magnitude()`	длина
`Vector direction()`	единичный вектор с таким же направлением, как у текущего вектора
`double cartesian(int i)`	i-я координата в декартовой системе
`String toString()`	строковое представление

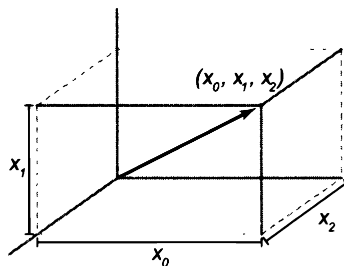
API пространственных векторов (см. листинг 3.3.3)

Как и в случае с API `Complex`, этот API не указывает явно, что данный тип является неизменяемым, но мы знаем, что разработчики клиентского кода (которые с большой вероятностью будут мыслить понятиями математической абстракции) наверняка будут ожидать этого.

Представление

Как обычно, первым решением в разработке реализации является выбор представления данных. Использование массива для хранения декартовых координат, переданных в конструкторе, — понятный, но не единственный разумный вариант. В самом деле, один из базовых канонов линейной алгебры гласит, что в качестве основы для системы координат могут использоваться и другие множества из n векторов: любой вектор может быть выражен в виде линейной комбинации множества n векторов, удовлетворяющей некоторому условию, известному как *линейная независимость*. Возможность изменения системы координат хорошо сочетается с инкапсуляцией. Многим клиентам вообще не нужно знать о внутреннем представлении; они могут работать с объектами `Vector` и операциями над ними. Если потребуется, реализация может изменять систему координат без изменения клиентского кода.

Реализация



Проецирование вектора (3D)

При таком представлении код, реализующий все эти операции (`Vector` в программе 3.3.3), достаточно очевиден. Конструктор создает «защитную» копию клиентского массива, и ни один из методов не присваивает значения этой копии, так что тип данных `Vector` является неизменным. Метод `cartesian()` легко реализуется в представлении декартовой системы координат: достаточно вернуть i-ю координату из массива. Фактически он

Листинг 3.3.3. Пространственные векторы

coords[]	декартовы координаты

```
public class Vector
{
   private final double[] coords;
   public Vector(double[] a)
   {  // Создание "защитной" копии для обеспечения неизменяемости.
      coords = new double[a.length];
      for (int i = 0; i < a.length; i++)
         coords[i] = a[i];
   }

   public Vector plus(Vector that)
   {  // Сумма текущего вектора и that.
      double[] result = new double[coords.length];
      for (int i = 0; i < coords.length; i++)
         result[i] = this.coords[i] + that.coords[i];
      return new Vector(result);
   }

   public Vector scale(double alpha)
   {  // Масштабирование текущего вектора с коэффициентом alpha.
      double[] result = new double[coords.length];
      for (int i = 0; i < coords.length; i++)
         result[i] = alpha * coords[i];
      return new Vector(result);
   }

   public double dot(Vector that)
   {  // Скалярная производительность текущего вектора и that.
      double sum = 0.0;
      for (int i = 0; i < coords.length; i++)
         sum += this.coords[i] * that.coords[i];
      return sum;
   }

   public double magnitude()
   {  return Math.sqrt(this.dot(this));  }

   public Vector direction()
   {  return this.scale(1/this.magnitude());  }

   public double cartesian(int i)
   {  return coords[i];  }
}
```

Данная реализация инкапсулирует математическую абстракцию пространственного вектора в виде неизменяемого типа данных Java. Типичные клиенты — Sketch (листинг 3.3.4) и Body (листинг 3.4.1). Реализация методов экземпляров minus() и toString() откладывается для упражнений (упражнение 3.3.4 и упражнение 3.3.14), как и тестовый клиент (упражнение 3.3.5).

реализует математическую функцию, определенную для любого представления `Vector`: геометрическую проекцию на i-ю ось в декартовых координатах.

Ссылка this

В методе экземпляра (или конструкторе) ключевое слово `this` обозначает ссылку на объект, для которого был вызван метод экземпляра (или конструктор). Ключевое слово `this` используется точно так же, как любая другая ссылка на объект (например, для `this` можно вызвать метод, передать в аргументе метода или обратиться к переменной экземпляра). Например, метод `magnitude()` типа данных `Vector` использует ключевое слово `this` в двух местах: для вызова метода `dot()` и в аргументе метода `dot()`. Таким образом, выражение `vector.magnitude()` эквивалентно `Math.sqrt(vector.dot(vector))`. Некоторые программисты на Java всегда используют `this` для обращения к переменным экземпляров. В пользу этого подхода можно сказать то, что он четко показывает, что вы обращаетесь к переменной экземпляра (вместо локальной переменной или параметра). Однако при этом в коде появляется множество избыточных ключевых слов `this`, поэтому мы будем экономно использовать `this` в своем коде.

Стоит ли возиться с типом данных `Vector`, если все его операции так легко реализуются с массивами? Вероятно, к настоящему моменту ответ на этот вопрос вам уже очевиден: стоит, чтобы пользоваться преимуществами модульного программирования, упростить отладку и сделать код ясным. Массив `double` — низкоуровневый механизм Java, и с его элементами могут выполняться самые разнообразные операции. Ограничиваясь операциями API `Vector` (многим клиентам нужны только эти операции), мы упрощаем процесс проектирования, реализации и сопровождения своих программ. Так как тип данных `Vector` является неизменяемым, его можно использовать точно так же, как примитивные типы. Например, при передаче `Vector` методу мы знаем, что его значение не изменится (а при передаче массива такой гарантии нет). Использование типа данных `Vector` и связанных с ним операций в программе — простой и естественный путь использования огромного объема математических знаний, накопленных на базе этой абстрактной концепции.

Язык Java предоставляет поддержку для определения отношений между объектами, называемых *наследованием*. Этот механизм широко применяется разработчиками, поэтому вы сможете подробно изучить его на учебных курсах по программированию. Вообще говоря, эффективное использование таких механизмов выходит за рамки книги, но мы кратко опишем две основные формы наследования в Java — *наследование интерфейса* и *наследование реализации*, потому что вы, скорее всего, столкнетесь с ними.

Наследование интерфейса

В языке Java существует конструкция `interface` для объявления интерфейсов — отношений между классами, в остальном не связанных; при этом определяется общий набор методов, которые должны входить в каждый класс, *реализующий* этот

интерфейс. Таким образом, интерфейс определяет контракт (обязательство) на реализацию классом некоторого набора методов. Мы называем это обязательство наследованием интерфейса, потому что реализующий класс наследует часть API из интерфейса. Интерфейсы позволяют писать клиентские программы, которые работают с разнотипными объектами, используя общие методы из интерфейса. Как и большинство новых концепций программирования, наследование интерфейса на первых порах может вызвать замешательство, но после нескольких примеров все встает на свои места.

Определение интерфейса

Рассмотрим вводный пример: допустим, вы хотите написать код для построения графика произвольной функции с вещественными значениями. Ранее вам уже встречались программы, которые строят график одной конкретной функции; для этого программа вычисляет значения нужной функции в точках, разделенных некоторым интервалом. Чтобы обобщить эти программы для произвольных функций, мы определим интерфейс Java для вещественных функций одной переменной:

```
public interface Function
{
   public abstract double evaluate(double x);
}
```

Первая строка объявления интерфейса напоминает объявление класса, но вместо ключевого слова `class` в ней используется ключевое слово `interface`. Тело интерфейса состоит из списка абстрактных методов. *Абстрактным методом* называется метод объявленный, но не содержащий никакого кода реализации; объявление состоит только из сигнатуры метода, за которой следует точка с запятой (`;`). Модификатор `abstract` помечает метод как абстрактный. Как и в случае с классом Java, интерфейс Java должен храниться в файле с именем, совпадающим с именем интерфейса, и расширением .java.

Реализация интерфейса

Интерфейс определяет контракт класса для реализации некоторого набора методов. Чтобы написать класс, реализующий интерфейс, необходимо выполнить два условия. Во-первых, в объявление класса должно быть включено ключевое слово `implements` с именем интерфейса. Считайте это своего рода «подписью под контрактом» — обязательством реализовать все абстрактные методы, объявленные в интерфейсе. Во-вторых, вы должны реализовать каждый абстрактный метод. Например, класс для вычисления квадрата вещественного числа, реализующий интерфейс `Function`, определяется следующим образом:

```
public class Square implements Function
{
   public double evaluate(double x)
   {  return x*x;  }
}
```

Аналогичным образом определяется класс для вычисления гауссовой функции плотности вероятности (см. листинг 2.1.2):

```
public class GaussianPDF implements Function
{
   public double evaluate(double x)
   {  return Math.exp(-x*x/2) / Math.sqrt(2 * Math.PI);  }
}
```

Если класс не реализует хотя бы один абстрактный метод, указанный в интерфейсе, вы получите ошибку компиляции. Однако класс, реализующий интерфейс, может содержать также и методы, не входящие в интерфейс.

Использование интерфейса

Интерфейс относится к ссылочным типам. Имя интерфейса может использоваться так же, как и имя любого другого типа данных. Например, в программе можно объявить переменную с типом, который является именем интерфейса. В этом случае любой объект, который присваивается этой переменной, должен быть экземпляром класса, реализующего интерфейс. Например, в переменной типа `Function` может храниться объект типа `Square` или `GaussianPDF`, но не типа `Complex`.

```
Function f1 = new Square();
Function f2 = new GaussianPDF();
Function f3 = new Complex(1.0, 2.0);  // Ошибка компиляции
```

Переменная интерфейсного типа может использоваться для вызова только тех методов, которые объявлены в интерфейсе, даже если реализующий класс определяет дополнительные методы.

Когда переменная интерфейсного типа используется для вызова метода, объявленного в интерфейсе, Java знает, какой метод нужно вызвать, потому что знает тип вызывающего объекта. Например, выражение `f1.evaluate()` вызовет метод `evaluate()`, определенный в классе `Square`, а выражение `f2.evaluate()` вызовет метод `evaluate()`, определенный в классе `GaussianPDF`. Этот мощный программный механизм называется *полиморфизмом*, или *динамической диспетчеризацией*[1].

Чтобы понять преимущества использования интерфейсов и полиморфизма, вернемся к приложению, строящему график функции `f` в интервале `[a, b]`. Если функция `f` является достаточно гладкой, можно получить значения функции в $n + 1$ точках, находящихся на одинаковом расстоянии в интервале `[a,b]`, и вывести результаты вызовами `StdStats.plotPoints()` или `StdStats.plotLines()`.

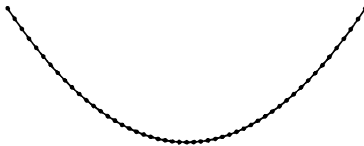
---

[1] Часто используется также термин *позднее связывание*. Таким образом подчеркивается, что «связывание» имени метода с его конкретной реализацией происходит «поздно» — только на этапе выполнения программы, через конкретный экземпляр (объект) конкретного типа, а не через тип ссылки на него. На этапе компиляции программы реальный тип объекта может оставаться неизвестен. — *Примеч. науч. ред.*

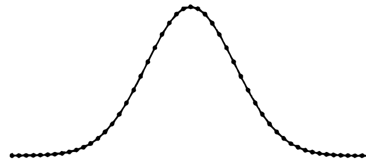
```
public static void plot(Function f, double a, double b, int n)
{
   double[] y = new double[n+1];
   double delta = (b - a) / n;
   for (int i = 0; i <= n; i++)
      y[i] = f.evaluate(a + delta*i);
   StdStats.plotPoints(y);
   StdStats.plotLines(y);
}
```

Преимущество объявления переменной `f` с интерфейсным типом `Function` заключается в том, что тот же вызов метода `f.evaluate()` будет работать для объекта `f` *любого* типа данных, реализующего интерфейс `Function`, включая `Square` или `GaussianPDF`. А это означает, что нам не нужно писать перегруженные методы для каждого типа — одна и та же функция `plot()` может использоваться для многих типов! Таким образом, возможность написания клиента для построения графика произвольной функции — наглядный пример эффекта от наследования интерфейса.

```
Function f1 = new Square();
plot(f1, -0.6, 0.6, 50);
```



```
Function f2 = new GaussianPDF();
plot(f2, -4.0, 4.0, 50);
```



Построение графика функции

Вычисления с использованием функций[1]

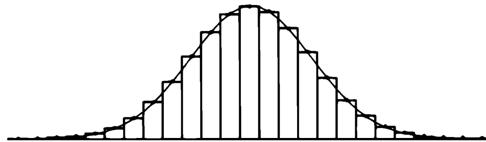
В вычислениях (особенно научных) часто используются функции: мы интегрируем функции, дифференцируем функции, находим корни функций и т. д. В некоторых языках программирования, относящихся к *функциональным*, на этой идее строится весь язык, а вычисления с использованием функций существенно упрощают клиентский код. К сожалению, в Java методы не являются полноценными объектами. Однако, как мы уже видели на примере функции `plot()`, некоторых из этих целей можно достичь при помощи интерфейсов. Возьмем задачу оценки интеграла Римана положительной функции с вещественными значениями (площади под кривой) на интервале `[a, b]`. Эти вычисления называются *квадратурой*, или *численным интегрированием*. Существует ряд методов вычисления квадратуры. Вероятно, простейшим из них является метод прямоугольников, когда значение интеграла аппроксимируется вычислением общей площади n прямоугольников равной ширины, лежащих под кривой. Функция `integrate()`, определение которой приводится

---

[1] Здесь имеются в виду функции в смысле *функционального программирования*. В ряде «обычных» языков (например, С) функция — это подпрограмма. — *Примеч. науч. ред.*

ниже, вычисляет интеграл функции f с вещественными значениями на интервале `[a, b]`, используя правило прямоугольника с n прямоугольниками:

```
public static double integrate(Function f,
                               double a, double b, int n)
{
   double delta = (b - a) / n;
   double sum = 0.0;
   for (int i = 0; i < n; i++)
      sum += delta * f.evaluate(a + delta * (i + 0.5));
   return sum;
}
```



Аппроксимация интеграла

Неопределенный интеграл x^2 равен $x^3/3$, так что определенный интеграл от 0 до 10 равен 1000/3. Вызов `integrate(new Square(), 0, 10, 1000)` возвращает результат `333.33324999999996` — правильный ответ с точностью до шести значащих цифр. Аналогичным образом вызов `integrate(new GaussianPDF(),-1,1,1000)` возвращает `0.6826895727940137` — правильный ответ с точностью до семи значащих цифр (вспомните гауссову функцию плотности вероятности и программу 2.1.2).

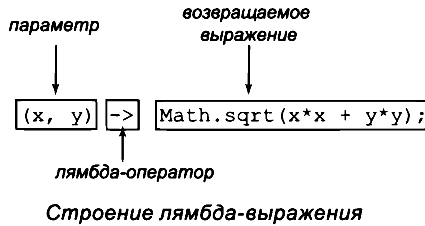
Численное интегрирование не всегда является самым эффективным или точным способом вычисления функции. Например, функция `Gaussian.cdf()` в листинге 2.1.2 предоставляет более быстрый и точный способ вычисления гауссовой функции плотности вероятности. Однако у квадратуры есть свое преимущество: она может использоваться для любой функции, удовлетворяющей лишь некоторым техническим условиям в отношении гладкости.

Лямбда-выражения

Синтаксис для организации вычислений с использованием функций, который мы только что рассмотрели, не идеален. Например, неудобно определять новый класс, реализующий интерфейс `Function` для каждой функции, которую нужно проинтегрировать или нанести на график. Для упрощения синтаксиса в подобных выражениях в Java реализован мощный механизм функционального программирования — *лямбда-выражения*. Считайте, что лямбда-выражение — это блок кода, который можно передать для последующего выполнения. В простейшей форме лямбда-выражение состоит из трех элементов:

- список параметров, разделенных запятыми и заключенных в круглые скобки;
- *лямбда-оператор* `->`;
- возвращаемое выражение.

Например, лямбда-выражение `(x, y) -> Math.sqrt(x*x + y*y)` реализует функцию вычисления гипотенузы. Если параметр только один, круглые скобки необязательны. Таким образом, лямбда-выражение `x -> x*x` реализует функцию вычисления квадрата, а `x -> Gaussian.pdf(x)` реализует гауссовскую функцию плотности вероятности.



Строение лямбда-выражения

Нас лямбда-выражения интересуют прежде всего как компактный механизм выражения функционального интерфейса (интерфейса с одним абстрактным методом), а именно: лямбда-выражение может использоваться везде, где ожидается объект функционального интерфейса. Например, функцию вычисления квадрата можно интегрировать вызовом `integrate(x -> x*x, 0, 10, 1000)`, без определения класса `Square`. Вам не нужно явно объявлять, что лямбда-выражение реализует интерфейс `Function`; при условии, что сигнатура одного абстрактного метода совместима с лямбда-выражением, Java определит этот факт по контексту. В данном случае лямбда-выражение `x -> x*x` совместимо с абстрактным методом `evaluate()`.

выражение
`new Square()`
`new GaussianPDF()`
`x -> x*x`
`x -> Gaussian.pdf(x)`
`x -> Math.cos(x)`

Типичные выражения, реализующие интерфейс Function

Встроенные интерфейсы

Java включает три интерфейса, которые будут более подробно описаны в книге. В разделе 4.2 будет рассмотрен интерфейс Java `java.util.Comparable`, содержащий один абстрактный метод `compareTo()`. Метод `compareTo()` определяет естественный для каждого типа порядок сравнения объектов этого типа: алфавитный порядок для строк, упорядочение по возрастанию для целых и вещественных чисел и т. д. Это позволяет писать код для сортировки массивов объектов. В разделе 4.3 мы будем использовать интерфейсы, позволяющие перебирать элементы в коллекциях вне зависимости от низкоуровневого представления. Для этой цели Java предоставляет два интерфейса: `java.util.Iterator` и `java.lang.Iterable`.

Событийное программирование

Другой убедительный пример полезности наследования интерфейса — его применение в событийном программировании. Представьте знакомую ситуацию: вы рассматриваете возможность расширения `Draw` для обработки действий пользователя (таких, как щелчки кнопкой мыши и нажатия клавиш). Одно из возможных решений — определение интерфейса, который указывает, какой метод (или методы) `Draw` должен вызываться при таких действиях пользователя. Иногда для описания вызова методом одного класса метода другого класса посредством интерфейса используется термин «*обратный вызов*» (callback)[1]. На сайте книги можно найти пример интерфейса `DrawListener` и информацию о том, как пишется код реакции на щелчки мышью и нажатия клавиш в `Draw`. Вы увидите, как легко пишется код, который создает объект `Draw` и включает метод, который вызывается из `Draw` (происходит *обратный вызов* вашего кода) для передачи информации вашему методу о символе, введенном с клавиатуры, или позиции мыши при щелчке. Написание интерактивного кода — дело интересное, но непростое, потому что вы должны заранее спланировать все возможные действия пользователя.

Наследование интерфейса — одна из нетривиальных концепций программирования, активно применяемая многими опытными программистами, потому что она обеспечивает возможность повторного использования кода без ущерба для инкапсуляции. К поддерживаемому ею *функциональному стилю программирования* в некоторых кругах относятся неоднозначно, но лямбда-выражения и аналогичные конструкции появились еще в начальный период эпохи программирования[2], и они проникли во многие современные языки программирования. У этого стиля есть свои страстные поклонники, которые верят, что только он заслуживает изучения и практического применения. Мы не стали говорить об этом с самого начала, потому что большая часть кода, с которым вы будете иметь дело, написана без него. Тем не менее упомянуть о функциональном программировании все же стоит, потому что каждый программист должен знать о нем и искать возможности с пользой применить его.

Наследование реализации (субклассирование)

В Java также поддерживается другой механизм наследования, называемый *субклассированием*. Идея заключается в том, чтобы определить новый класс (*субкласс*, или *производный класс*), который наследует переменные экземпляров (*состояние*) и методы экземпляров (*поведение*) от другого класса (*суперкласс*, или *базовый*

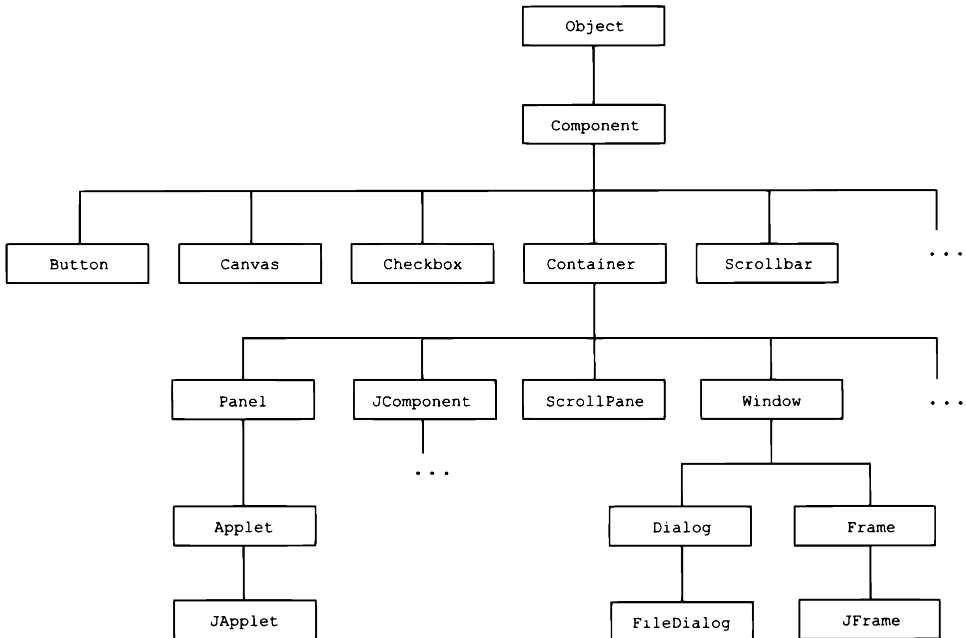
---

[1] О механизме «обратного» вызова говорится вообще в тех случаях, когда вызов некоторой подпрограммы (метода) предполагается не явным образом в рамках текущего алгоритма, из того же модуля, где находится эта подпрограмма, а откуда-либо извне: другим модулем, другой программой, системой. — *Примеч. науч. ред.*

[2] Парадигмы функционального и более привычного нам *императивного* программирования на начальном этапе развививались параллельно, и преимущества той или иной из них не были очевидны. — *Примеч. науч. ред.*

класс), открывая возможность повторного использования кода. Как правило, субкласс переопределяет некоторые методы суперкласса. Мы называем такую схему «наследованием реализации», потому что один класс наследует код от другого.

Системные программисты применяют субклассирование для построения так называемых *расширяемых библиотек* — любой программист (в том числе и вы) может добавить методы в библиотеку, построенную другим программистом (или, возможно, группой программистов сложных систем), что фактически позволяет ему повторно использовать код огромной библиотеки. Этот прием применяется довольно часто, особенно при разработке пользовательских интерфейсов, чтобы большая часть кода, необходимая для поддержки всей стандартной функциональности (окна, кнопки, полосы прокрутки, меню, копирование/вставка, операции с файлами и т. д.), могла использоваться повторно.



Иерархия наследования субклассов компонентов графического интерфейса (фрагмент)

В среде системных программистов по поводу субклассирования нет единого мнения, потому что его преимущества перед наследованием интерфейса неочевидны. В этой книге мы избегаем субклассирования, поскольку оно противоречит концепции инкапсуляции в двух отношениях. Во-первых, любое изменение суперкласса отражается на всех его субклассах. Субкласс не может разрабатываться независимо от суперкласса; более того, он *полностью зависит* от суперкласса. Эта проблема

называется проблемой *изменчивости базового класса*. Во-вторых, код субкласса, имеющий доступ к переменным экземпляра суперкласса, может противоречить намерениям кода суперкласса. Представьте, что проектировщик класса (такого, как `Vector`) потрудился, чтобы сделать класс `Vector` неизменяемым, а субкласс, обладающий полным доступом к этим переменным экземпляров, может изменять их по своему усмотрению.

Суперкласс Object в Java

Некоторые рудименты субклассирования встроены в язык Java, а следовательно, неизбежны. А именно любой класс является субклассом класса Java `Object`. Такая структура позволяет соблюдать «правило» о том, что каждый класс включает реализации `toString()`, `equals()`, `hashCode()` и нескольких других методов. Каждый класс наследует эти методы от `Object` посредством субклассирования. При программировании на Java разработчики часто переопределяют один или несколько из этих методов.

public class Object	
`String toString()`	строковое представление содержимого объекта
`boolean equals(Object x)`	данный объект равен объекту x?
`int hashCode()`	хеш-код объекта
`Class getClass()`	класс объекта

Методы, наследуемые всеми классами (используемые в книге)

Преобразование в строку

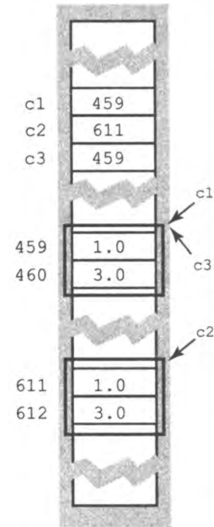
Каждый класс Java наследует метод `toString()`, так что любой клиент может вызвать `toString()` для любого объекта. Как и в случае с интерфейсами, Java знает, какой метод `toString()` следует вызвать (полиморфно), потому что Java знает тип вызывающего объекта. Это соглашение лежит в основе автоматического преобразования одного операнда оператора конкатенации строк + в строку, если другой операнд является строкой. Например, если `x` — ссылка на произвольный объект, то Java автоматически преобразует выражение `"x = " + x` в `"x = " + x.toString()`. Если класс не переопределяет метод `toString()`, то Java вызывает унаследованную реализацию `toString()`, чаще всего бесполезную (обычно она содержит строковое представление адреса объекта в памяти). Итак, хороший стиль программирования рекомендует переопределять метод `toString()` в каждом классе, который вы разрабатываете.

Равенство

Что следует понимать под равенством двух объектов? Если проверить равенство условием `(x == y)`, где `x` и `y` — ссылки на объекты, то вы тем самым проверите, имеют ли они одно и то же содержимое, — иначе говоря, равны ли ссылки на объекты. Напри-

мер, возьмем код на диаграмме, который создает два объекта `Complex` (листинг 3.2.6), на который ссылаются три переменные `c1`, `c2` и `c3`. Как показано на диаграмме, `c1` и `c3` ссылаются на один объект, отличный от объекта, на который ссылается `c2`. Соответственно условие `(c1 == c3)` истинно, но условие `(c1 == c2)` ложно. Такое поведение называется *эквивалентностью ссылок*, но обычно клиенту нужно нечто другое.

```
Complex c1, c2, c3;
c1 = new Complex(1.0, 3.0);
c2 = new Complex(1.0, 3.0);
c3 = c1;
```



Три ссылки на два объекта

Типичный клиент хочет проверить, совпадают ли *значения* типа данных (состояние объекта). Это поведение называется *равенством объектов*. В Java для этой цели существует метод `equals()`, наследуемый всеми классами. Например, тип данных `String` переопределяет этот метод естественным образом: если `x` и `y` ссылаются на объекты `String`, то результат `x.equals(y)` равен `true` в том и только в том случае, если две строки представляют одну последовательность символов (независимо от того, указывают ли ссылки на один и тот же объект `String`).

По правилам Java метод `equals()` должен реализовать отношение эквивалентности, обеспечив выполнение трех естественных свойств для любых ссылок на объекты `x`, `y` и `z`.

- Рефлексивность: условие `x.equals(x)` истинно.
- Симметричность: условие `x.equals(y)` истинно в том и только в том случае, если истинно условие `y.equals(x)`.
- Транзитивность: если условие`x.equals(y)` истинно и условие `y.equals(z)` тоже истинно, то условие `x.equals(z)` также истинно.

Кроме того, должны выполняться следующие два свойства.

- Многократные вызовы `x.equals(y)` возвращают один и тот же логический результат (при условии, что ни один из объектов не изменялся между вызовами).
- Условие `x.equals(null)` ложно.

Как правило, при определении собственных типов данных мы переопределяем метод `equals()`, потому что унаследованная реализация выполняет сравнение ссылок. Например, представьте, что два объекта `Complex` должны считаться равными тогда и только тогда, когда их действительные и мнимые части совпадают. Следующая реализация решает эту задачу:

```
public boolean equals(Object x)
{
   if (x == null) return false;
```

```
    if (this.getClass() != x.getClass()) return false;
    Complex that = (Complex) x;
    return (this.re == that.re) && (this.im == that.im);
}
```

Этот код неожиданно нетривиален, потому что аргумент `equals()` может содержать ссылку на объект любого типа (или `null`), поэтому мы кратко охарактеризуем смысл каждой операции.

- Первая строка возвращает `false` для аргументов `null`, как и требуется.
- Вторая строка использует унаследованный метод `getClass()`, чтобы вернуть `false`, если два объекта относятся к разным типам.
- Преобразование типа в третьей команде заведомо успешно из-за второй команды.
- Последняя команда реализует логику проверки равенства, сравнивая соответствующие переменные экземпляров двух объектов.

Вы можете использовать эту реализацию в качестве шаблона — стоит вам реализовать один метод `equals()`, и с реализацией других уже не будет проблем.

Хеширование

А сейчас мы рассмотрим фундаментальную операцию, связанную с проверкой равенства. Эта операция, называемая *хешированием*, связывает объект с целым числом, которое называется *хеш-кодом*. Хеширование обеспечивается методом `hashCode()`, который наследуется всеми классами. По правилам Java метод `hashCode()` должен выполнять следующие условия для любых ссылок на объекты `x` и `y`.

- Если результат `x.equals(y)` — истина, то значения `x.hashCode()` и `y.hashCode()` равны.
- Многократные вызовы `x.hashCode()` возвращают один и тот же целочисленный результат (при условии, что объект не изменялся между вызовами).

Например, в следующем фрагменте кода переменные `x` и `y` ссылаются на равные объекты `String` (выражение `x.equals(y)` истинно), поэтому они должны иметь одинаковый хеш-код; `x` и `z` ссылаются на разные объекты `String`, поэтому их хеш-коды должны различаться.

```
String x = new String("Java");   // x.hashCode() возвращает 2301506
String y = new String("Java");   // y.hashCode() возвращает 2301506
String z = new String("Python"); // z.hashCode() возвращает -1889329924
```

В типичном приложении хеш-код используется для связывания объекта `x` с целым числом в небольшом диапазоне (скажем, от 0 до `m-1`) с использованием следующей *хеш-функции*:

```
private int hash(Object x)
{  return Math.abs(x.hashCode() % m);  }
```

Вызов Math.abs() гарантирует, что возвращаемое значение не является отрицательным числом, что могло бы случиться при получении отрицательного значения от x.hashCode(). Значение хеш-функции может использоваться как целочисленный индекс в массиве длины m (полезность этих операций станет очевидной в листингах 3.3.4 и 4.4.3). По правилам объекты с равными значениями должны иметь одинаковые хеш-коды, чтобы они также имели одинаковые значения хеш-функции. Объекты с различающимися значениями могут иметь одинаковые значения хеш-функций, но предполагается, что хеш-функция делит n типичных объектов класса на m групп приблизительно одинакового размера. Многие неизменяемые типы данных Java (включая String) включают реализации hashCode(), специально разработанные для разумного распределения объектов.

Построение хорошей реализации hashCode() для типа данных требует искусного сочетания научных и технических знаний и выходит за рамки книги. Вместо этого мы опишем простой рецепт реализации на языке Java, эффективный во многих ситуациях.

- Убедитесь в том, что тип данных является неизменяемым.
- Импортируйте класс java.util.Objects.
- Реализуйте equals() сравнением всех значимых переменных экземпляров.
- Реализуйте hashCode(), передавая все значимые переменные экземпляров в аргументах статического метода Objects.hash().

```
import java.util.Objects;

public class Complex
{
   private final double re, im;
   ...

   public boolean equals(Object x)
   {
      if (x == null) return false;
      if (this.getClass() != x.getClass())
         return false;
      Complex that = (Complex) x;
      return (this.re == that.re)
          && (this.im == that.im);
   }

   public int hashCode()
   {  return Objects.hash(re, im);  }

   public String toString()
   {  return re + " + " + im + "i";  }

}
```

Переопределение методов equals(), hashCode() и toString()

Статический метод `Objects.hash()` генерирует хеш-код для последовательности своих аргументов. Например, следующая реализация `hashCode()` для типа данных `Complex` (листинг 3.2.1) составит компанию только что рассмотренной реализации `equals()`:

Типы-обертки

Одной из главных выгод от использования наследования является повторное использование кода. Впрочем, оно ограничивается ссылочными типами, но неприменимо для примитивных. Например, выражение `x.hashCode()` действительно для любой ссылки на объект `x`, но если `x` является переменной примитивного типа, происходит ошибка компиляции. Для ситуаций, когда значение примитивного типа должно быть представлено в виде объекта, Java предоставляет встроенные ссылочные типы — так называемые *типы-обертки*, по одному для каждого из 8 примитивных типов. Например, типы-обертки `Integer` и `Double` представляют `int` и `double` соответственно. Тип-обертка «упаковывает» значение примитивного типа в объект, чтобы для него можно было вызывать такие методы экземпляров, как `equals()` и `hashCode()`. Каждый из этих типов-оберток является неизменяемым и включает как методы экземпляров (например, `compareTo()` для числового сравнения двух объектов), так и статические методы (например, `Integer.parseInt()` и `Double.parseDouble()` для преобразования строк в примитивные типы).

примитивный тип	тип-обертка
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Автоупаковка и распаковка

Java автоматически выполняет преобразования между типами-обертками и соответствующими значениями примитивных типов (в операциях присваивания, аргументах методов, математических/логических выражениях), чтобы вы могли писать код следующего вида:

```
Integer x = 17;  // Автоупаковка (int -> Integer)
int a = x;       // Распаковка (Integer -> int)
```

В первой команде Java автоматически преобразует значение 17 типа `int` в объект типа `Integer` перед тем, как присваивать его переменной `x`. Аналогичным обра-

зом во второй строке Java автоматически преобразует объект `Integer` в значение типа `int`, прежде чем присваивать его переменной `a`. Автоупаковка и распаковка удобны при написании кода, но они сопровождаются непроизводительными затратами, которые могут повлиять на быстродействие.

Для ясности и производительности кода мы используем примитивные типы для вычислений с числами там, где это возможно. Однако в главе 4 будут приведены наглядные примеры (особенно с типами данных, в которых хранятся коллекции объектов) того, как типы-обертки и автоматическая упаковка/распаковка позволяют писать код для ссылочных типов, а затем повторно использовать этот код (без изменений) с примитивными типами.

Приложение: анализ данных

Чтобы продемонстрировать некоторые концепции, представленные в этом разделе, мы рассмотрим технологию программирования, которая играет важную роль в решении непростых задач *анализа данных* (этим термином обозначается процесс обнаружения закономерностей при поиске в огромных объемах информации). Эта технология может послужить основой для радикального повышения качества результатов веб-поиска, при выборке мультимедийной информации, создании биомедицинских баз данных, исследованиях в геномике, внедрении инноваций в коммерческих приложениях, выявлении планов злоумышленников и многих других целей. По этим причинам область анализа данных пользуется повышенным интересом специалистов, и в ней ведутся интенсивные исследования.

Вам напрямую доступны тысячи файлов на вашем компьютере и косвенно — миллиарды файлов в Интернете. Как вы знаете, эти файлы чрезвычайно разнообразны: коммерческие веб-страницы, музыка и видео, электронная почта, программный код и всевозможные виды информации. Для простоты мы ограничимся текстовыми документами (хотя рассматриваемый метод применим к графике, музыке и другим файлам). Даже с учетом таких ограничений типы документов чрезвычайно разнообразны. Для сведения вы найдете примеры таких документов на сайте книги.

имя файла	описание	пример текста
`Constitution.txt`	юридический документ	`... of both Houses shall be determined by ...`
`TomSawyer.txt`	американская повесть	`..."Say, Tom, let ME whitewash a little." ...`
`HuckFinn.txt`	американская повесть	`...was feeling pretty good after breakfast...`
`Prejudice.txt`	английский роман	`... dared not even mention that gentleman....`
`Picture.java`	код Java	`...String suffix = filename.substring(file...`
`DJIA.csv`	финансовые данные	`...01-Oct-28,239.43,242.46,3500000,240.01 ...`
`Amazon.html`	код веб-страницы	`...<table width="100%" border="0" cellspac...`
`ACTG.txt`	геном вируса	`...GTATGGAGCAGCAGACGCGCTACTTCGAGCGGAGGCATA...`

Примеры текстовых документов

Итак, нас интересуют эффективные механизмы поиска в файлах с использованием их содержимого для получения характеристик документов. Одно из эффективных решений этой задачи состоит в том, чтобы снабдить документ *конспектом* — вектором, который определяется как функция содержимого документа. Основная идея заключается в том, что конспект должен характеризовать документ таким образом, чтобы документы с разными конспектами различались, а у похожих документов конспекты тоже были похожи. Естественно, этот метод способен отличить роман от кода Java или генома, но интересно другое: поиск по содержимому позволяет различать тексты, написанные разными авторами, и может лежать в основе многих других нетривиальных критериев поиска.

Для начала потребуется абстракция для представления текстовых документов. Что же такое текстовый документ? Какие операции должны выполняться с текстовыми документами? Ответы на эти вопросы дадут необходимую информацию для проектирования, а в конечном итоге и для кода, который мы напишем. В контексте анализа данных ответ на первый вопрос очевиден: текстовый документ определяется как строка. Что касается второго вопроса, нам потребуется возможность вычисления метрики, оценивающей степень сходства документа с любым другим документом. Из этих соображений мы приходим к следующему API:

public class Sketch	
Sketch(String text, int k, int d)	
double similarTo(Sketch other)	степень сходства между этим конспектом и other
String toString()	строковое представление

API конспектов (см. листинг 3.3.4)

В аргументах конструктора передается текстовая строка и два целых числа, управляющие качеством и размером конспекта. Клиенты могут использовать метод `similarTo()` для определения степени сходства между текущим объектом `Sketch` и любым другим объектом `Sketch` по шкале от 0 (сходство отсутствует) до 1 (полное сходство). Метод `toString()` прежде всего предназначен для целей отладки. Этот тип данных обеспечивает хорошее разделение между реализацией метрики сходства и реализацией клиентов, использующих метрику сходства для поиска в документах.

Вычисление конспектов

Первая задача — вычисление конспекта текстовой строки. Для представления конспекта документа будет использоваться последовательность вещественных чисел (или `Vector`). Но какие данные должны войти в конспект и как вычислить конспект `Vector`? Существует много разных методов, и теоретики продолжают активно искать эффективные алгоритмы для этой задачи. В нашей реализации `Sketch` (листинг 3.3.4) используется простое решение со счетчиками частот. Конструктор имеет два аргумента: целое число k и длину вектора d. Он сканирует документ и анализирует все k-граммы в документе, то есть подстроки длины k, начинающиеся в каждой позиции строки. В простейшей форме конспект пред-

ставляет собой вектор с относительными частотами вхождения k-грамм в строке; элемент, соответствующий каждой возможной k-грамме, содержит количество таких k-грамм в тексте. Предположим, мы используем $k = 2$ в данных геномов с $d = 16$ (4 возможных значения символов и, следовательно, $4^2 = 16$ возможных 2-грамм). 2-грамма AT встречается в строке ATAGATGCATAGCGCATAGC 4 раза, поэтому, например, элемент вектора, соответствующий AT, будет равен 4. Чтобы построить вектор частот, необходимо иметь возможность преобразовать каждую из 16 возможных k-грамм в целое число от 0 до 15 (*хеш-код*). Для данных генома это делается легко (см. упражнение 3.3.28). Затем массив для построения вектора частот вычисляется одним проходом по тексту с увеличением элемента массива, соответствующего каждой из обнаруженных k-грамм. Возможно, игнорирование порядка k-грамм приведет к потере части информации, но информационная ценность порядка ниже, чем у частоты. Чтобы учитывать порядок следования, можно воспользоваться парадигмой модели Маркова, отчасти похожей на ту, которая была описана для модели случайного серфинга в разделе 1.6, — такие модели эффективны, но их реализация требует гораздо больших усилий. Инкапсуляция вычислений в `Sketch` предоставляет необходимую гибкость, чтобы вы могли поэкспериментировать с различными решениями, не переписывая клиенты `Sketch`.

		ATAGATGCAT AGCGCATAGC		CTTTCGGTTT GGAACCGAAG CCGCGCGTCT TGTCTGCTGC AGCATCGTTC	
2-грамма	**хеш**	**количество**	**единичный вектор**	**количество**	**единичный вектор**
AA	0	0	0	2	.137
AC	1	0	0	1	.069
AG	2	4	.508	1	.069
AT	3	3	.381	2	.137
CA	4	3	.381	3	.206
CC	5	0	0	2	.137
CG	6	0	0	4	.275
CT	7	1	.127	6	.412
GA	8	3	.381	0	0
GC	9	0	0	5	.343
GG	10	0	0	6	.412
GT	11	1	.127	4	.275
TA	12	1	.127	2	.137
TC	13	4	.508	6	.412
TG	14	0	0	4	.275
TT	15	0	0	2	.137

«Портрет» генома

Листинг 3.3.4. Конспект документа

```
public class Sketch
{
   private final Vector profile;
   public Sketch(String text, int k, int d)
   {
      int n = text.length();
      double[] freq = new double[d];
      for (int i = 0; i < n-k; i++)
      {
         String kgram = text.substring(i, i+k);
         int hash = kgram.hashCode();
         freq[Math.abs(hash % d)] += 1;
      }
      Vector vector = new Vector(freq);
      profile = vector.direction();
   }
   public double similarTo(Sketch other)
   {  return profile.dot(other.profile);  }
   public static void main(String[] args)
   {
      int k = Integer.parseInt(args[0]);
      int d = Integer.parseInt(args[1]);
      String text = StdIn.readAll();
      Sketch sketch = new Sketch(text, k, d);
      StdOut.println(sketch);
   }
}
```

profile	единичный вектор

name	имя документа
k	длина «-граммы»
d	длина
text	весь документ
n	длина документа
freq[]	частоты хеш-кодов
hash	хеш-код k-граммы

Клиент Vector создает на основании k-грамм документа d-мерный единичный вектор, который может использоваться клиентами для оценки его сходства с другими документами (см. текст). Метод toString() приведен в упражнении 3.3.15.

```
% more genome20.txt
ATAGATGCATAGCGCATAGC
% java Sketch 2 16 < genome20.txt
(0.0, 0.0, 0.0, 0.620, 0.124, 0.372, ..., 0.496, 0.372, 0.248, 0.0)
```

Хеширование

У текстовых строк в кодировке ASCII каждый символ может иметь 128 различных значений[1], поэтому существуют 128^k разных k-грамм, а длина d в только что описанной схеме будет равна 128^k. Эта величина оказывается недопустимо большой даже для умеренных k. В кодировке Юникод, содержащей более 65 536 символов, векторы конспектов получаются огромными уже с 2-граммами. Для решения этой проблемы применяется *хеширование* — фундаментальная операция, связанная с поисковыми алгоритмами, которая совсем недавно упоминалась при обсуждении

[1] Для стандартной таблицы ASCII. Расширенная ASCII-таблица, включающая не только латиницу, содержит 256 значений. — *Примеч. науч. ред.*

наследования. Напомним, что все объекты наследуют метод `hashCode()`, который возвращает целое число в диапазоне от -2^{31} до $2^{31} - 1$. Для произвольной строки `s` выражение `Math.abs(s.hashCode() % d)` дает целочисленный хеш-код в диапазоне от 0 до `d-1`, который может использоваться как индекс в массиве длины `d` для вычисления частот. Конспект, который мы будем использовать, представляет собой *направление* вектора, определяемого всеми этими значениями для всех k-грамм в документе (единичный вектор с тем же направлением). Так как разные строки должны иметь разные хеш-коды, текстовые документы с похожими распределениями k-грамм будут иметь похожие конспекты, а у документов с разными распределениями k-грамм конспекты с очень большой вероятностью будут различаться.

Сравнение конспектов

Вторая задача — вычисление метрики сходства между двумя конспектами. Два вектора можно сравнить многими разными способами. Вероятно, самый простой из них — вычисление евклидова расстояния между ними. Для заданных векторов **x** и **y** это расстояние определяется формулой

$$|\mathbf{x} - \mathbf{y}| = ((x_0 - y_0)^2 + (x_1 - y_1)^2 + \ldots + (x_{d-1} - y_{d-1})^2)^{1/2}$$

Вам уже знакома эта формула для $d = 2$ или $d = 3$. С классом `Vector` евклидово расстояние вычисляется просто. Если x и y — два объекта `Vector`, то евклидово расстояние между ними определяется выражением `x.minus(y).magnitude()`. Если документы похожи, их конспекты должны быть похожими, а расстояние между ними — небольшим. Другая распространенная метрика сходства — *косинусное сходство* — еще проще: так как наши конспекты являются единичными векторами с неотрицательными координатами, их *скалярное произведение*

$$\mathbf{x} \bullet \mathbf{y} = x_0 y_0 + x_1 y_1 + \ldots + x_{d-1} y_{d-1}$$

является вещественным числом от 0 до 1. С геометрической точки зрения эта величина равна косинусу угла между двумя векторами (см. упражнение 3.3.10). Чем больше сходства между документами, тем ближе эта метрика к 1.

Сравнение всех пар

Программа `CompareDocuments` (листинг 3.3.5) — простой и полезный клиент `Sketch`, который предоставляет информацию, необходимую для решения следующей задачи: найти в заданном множестве документов два самых похожих документа. Так как постановка задачи немного субъективна, `CompareDocuments` выводит косинусную метрику сходства для всех пар документов из входного списка. Для умеренных значений k и d конспекты отлично справляются с описанием документов из множества. Из результатов следует не только то, что данные геномов, финансовые данные, код Java и код веб-страниц заметно отличаются от юридических документов и романов, но и то, что у «Тома Сойера» и «Гекльберри Финна» намного больше общего друг с другом, чем с «Гордостью и предубеждением». Ученые, занимающиеся сравнительным литературоведением, могут использовать эту программу для

Листинг 3.3.5. Выявление сходства

```
public class CompareDocuments
{
   public static void main(String[] args)
   {
      int k = Integer.parseInt(args[0]);
      int d = Integer.parseInt(args[1]);
      String[] filenames = StdIn.readAllStrings();
      int n = filenames.length;
      Sketch[] a = new Sketch[n];
      for (int i = 0; i < n; i++)
         a[i] = new Sketch(new In(filenames[i]).readAll(), k, d);
      StdOut.print(„    „);
      for (int j = 0; j < n; j++)
         StdOut.printf(„%8.4s",filenames[j]);
      StdOut.println();
      for (int i = 0; i < n; i++)
      {
         StdOut.printf(„%.4s", filenames[i]);
         for (int j = 0; j < n; j++)
            StdOut.printf(„%8.2f", a[i].similarTo(a[j]));
         StdOut.println();
      }
   }
}
```

k	длина «-граммы»
d	размерность
n	количество документов
a[]	конспекты

Клиент Sketch читает список документов из стандартного ввода, вычисляет конспекты на основании частот k-грамм для всех документов и выводит таблицу метрик сходства между всеми параметрами документов. Программа получает в командной строке два аргумента: значение k и размерность d конспектов.

```
% java CompareDocuments 5 10000 < documents.txt
        Cons    TomS    Huck    Prej    Pict    DJIA    Amaz    ACTG
Cons    1.00    0.66    0.60    0.64    0.20    0.18    0.21    0.11
TomS    0.66    1.00    0.93    0.88    0.12    0.24    0.18    0.14
Huck    0.60    0.93    1.00    0.82    0.09    0.23    0.16    0.12
Prej    0.64    0.88    0.82    1.00    0.11    0.25    0.19    0.15
Pict    0.20    0.12    0.09    0.11    1.00    0.04    0.39    0.03
DJIA    0.18    0.24    0.23    0.25    0.04    1.00    0.16    0.11
Amaz    0.21    0.18    0.16    0.19    0.39    0.16    1.00    0.07
ACTG    0.11    0.14    0.12    0.15    0.03    0.11    0.07    1.00
```

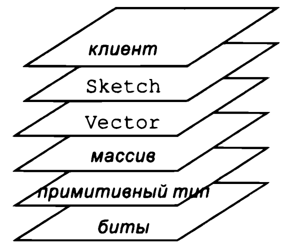
выявления связей между текстами; также такие программы могут применяться для поиска плагиата в работах студентов (собственно, многие преподаватели регулярно пользуются такими программами); биологу такая программа поможет в выявлении связей между геномами. Возьмите подборку документов на сайте книги (или соберите ее самостоятельно) для проверки эффективности `CompareDocuments` с различными значениями параметров.

```
% more documents.txt
Consititution.txt
TomSawyer.txt
HuckFinn.txt
Prejudice.txt
Picture.java
DJIA.csv
Amazon.html
ACTG.txt
```

Поиск похожих документов

Другая естественная задача для клиента `Sketch` — поиск среди большого количества документов похожих на заданный документ. Например, поисковые системы в Интернете используют программы такого типа для выдачи страниц, похожих на посещавшиеся ранее; книжные интернет-магазины — чтобы порекомендовать вам книги, похожие на купленные ранее; социальные сети — чтобы найти людей с интересами, похожими на ваши. Так как тип `In` может получать веб-адреса вместо имен файлов, ничто не мешает вам написать программу, которая обходит веб-сайты, вычисляет конспекты и возвращает ссылки на веб-страницы с конспектами, близкими к конспекту искомой. При желании вы можете реализовать такой клиент самостоятельно.

Другие решения



Уровни абстракции

Ученые продолжают изобретать, изучать и сравнивать многочисленные сложные алгоритмы для эффективного вычисления конспектов. Мы хотим познакомить вас с этой фундаментальной областью и одновременно продемонстрировать мощь абстракций при решении вычислительных задач. Векторы принадлежат к числу важнейших математических абстракций, а клиент для проверки сходства строится наращиванием *уровней абстракции*: `Vector` строится на базе массива Java, `Sketch` строится на базе `Vector`, клиентский код использует `Sketch`. Как обычно, мы избавим вас от описания наших многочисленных попыток разработки таких API, но вы видите, что типы данных разрабатываются под специфику задачи, с учетом потребностей реализаций. Выявление и реализация подходящих абстракций — ключ к эффективному объектно-ориентированному программированию. Мощь абстракции — в математике, физических моделях и компьютерных программах — постоянно проявляется в рассмотренных нами примерах. Когда у вас появится опыт разработки типов данных для решения ваших собственных вычислительных задач, вы оцените это в полной мере.

Контрактное проектирование

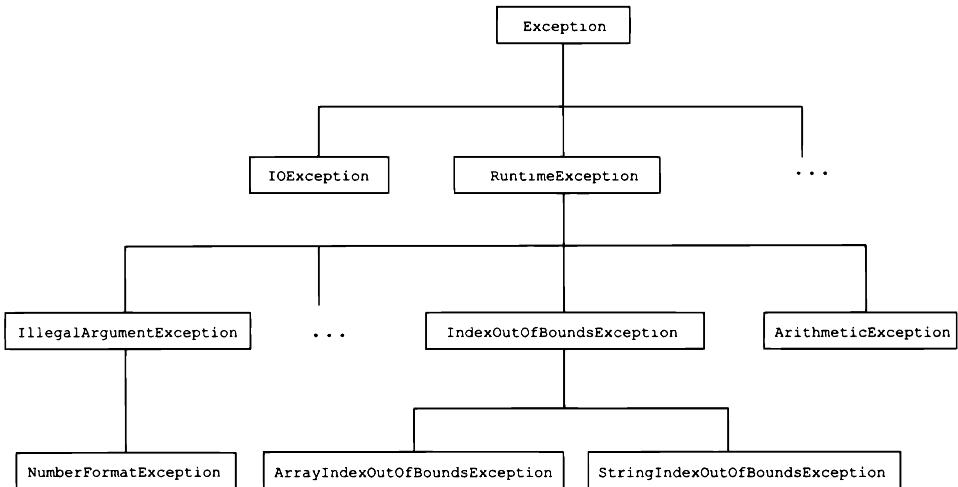
В завершение мы кратко рассмотрим механизмы языка Java, позволяющие проверять предположения относительно вашей программы *во время ее выполнения*.

Например, если у вас имеется тип данных, представляющий частицу, можно утверждать, что масса частицы положительна, а ее скорость меньше скорости света. А если имеется метод суммирования двух векторов одинаковой длины, то и длина полученного вектора должна быть такой же, как у исходных векторов.

Исключения

Исключением называется аномальное событие, которое происходит во время выполнения программы и часто сигнализирует о возникновении ошибки. Исключения, инициируемые системными методами Java, уже встречались вам при изложении основ программирования: типичные примеры — `ArithmeticException`, `IllegalArgumentException`, `NumberFormatException` и `ArrayIndexOutOfBounds Exception`.

Вы также можете создавать и инициировать собственные исключения. Java содержит тщательно выработанную иерархию наследования встроенных исключений; каждый класс исключения является субклассом `java.lang.Exception`. Часть этой иерархии представлена на следующей диаграмме.



Иерархия наследования субклассов исключений (фрагмент)

Пожалуй, самой простой разновидностью исключений является `RuntimeException`. Следующая строка кода создает исключение `RuntimeException`; как правило, это приводит к завершению программы и выводу заданного сообщения об ошибке:

```
throw new RuntimeException("Custom error message here.");
```

На практике исключения рекомендуется использовать тогда, когда они могут пригодиться пользователю. Например, в программе `Vector` (листинг 3.3.3) в методе

`plus()` должно инициироваться исключение при несовпадении размеров векторов. Для этого в начало `plus()` вставляется следующий код:

```
if (this.coords.length != that.coords.length)
   throw new IllegalArgumentException("Dimensions disagree.");
```

Таким образом клиент получает точное описание нарушения API (вызов метода `plus()` с векторами разной длины), чтобы программисту было проще найти и исправить ошибку. Без этой команды метод `plus()` работает хаотично: он либо инициирует исключение `ArrayIndexOutOfBoundsException`, либо возвращает невалидный результат в зависимости от длин двух векторов (см. упражнение 3.3.16).

Диагностические утверждения

Диагностическое утверждение (assertion)[1] представляет собой логическое выражение, истинность которого вы проверяете в некоторой точке выполнения программы. Если выражение ложно, то программа выдает ошибку `AssertionError`, что приводит к завершению программы и выводу сообщения об ошибке. Ошибки похожи на исключения, но они свидетельствуют о катастрофических сбоях; ранее мы уже встречали два примера такого рода: `StackOverflowError` и `OutOfMemoryError`.

Диагностические утверждения широко используются программистами для выявления ошибок и обретения уверенности в правильности программ. Они также служат для документирования намерений программиста. Например, в программе `Counter` (листинг 3.3.2) мы можем убедиться в том, что счетчик не принимает отрицательные значения, добавив следующее диагностическое утверждение последней строкой в метод `increment()`:

```
assert count >= 0;
```

Эта команда выявляет отрицательные значения `count`. Также вы можете добавить собственное сообщение, которое упростит поиск ошибок:

```
assert count >= 0 : "Negative count detected in increment()";
```

По умолчанию диагностические утверждения отключены, но вы можете включить их, используя в командной строке флаг `-enableassertions` (сокращенно `-ea`). Диагностические утверждения предназначены только для отладки; ваша программа не должна использовать их в нормальной работе, потому что они могут быть отключены.

[1] Буквальный перевод этого термина выглядит неуклюже, но более удачный не сформировался. Сам механизм нередко оказывается сложным для понимания. Важно хорошо представлять себе, что это не проверка как таковая, а своего рода «заявка» на обязательное выполнение некоторых условий в корректно работающей программе (об этом напоминает само название конструкции). Невыполнение условий «заявки» рассматривается как авария. — *Примеч. науч. ред.*

В учебном курсе программирования сложных систем вы научитесь пользоваться диагностическими утверждениями для того, чтобы гарантировать, что ваш код никогда не завершается системной ошибкой и не переходит в бесконечный цикл. Наша модель, называемая *моделью контрактного проектирования*, выражает эту идею. Проектировщик типа данных формулирует *предусловие* (условие, которое клиент обещает выполнять при вызове метода), *постусловие* (условие, которое реализация обещает выполнять при возврате управления из метода), *инвариант* (любое условие, которое реализация обещает выполнять во время выполнения метода) и *побочные эффекты* (любое изменение состояния, обусловленное методом). В ходе разработки эти условия могут проверяться при помощи диагностических утверждений. Многие программисты широко применяют этот механизм для упрощения отладки.

Языковые механизмы, представленные в этом разделе, показывают, что эффективное проектирование типов данных ведет к настоящим глубинам проектирования языков программирования. Эксперты все еще спорят по поводу того, как лучше выразить некоторые из обсуждаемых идей. Почему в Java нельзя передавать функции как аргументы методов? Почему в Python не обеспечивается гарантированная инкапсуляция? Почему в Matlab не поддерживаются изменяемые типы данных? Как упоминалось в начале главы 1, от жалоб на функциональность языка программирования — прямой путь к проектированию новых языков программирования. Если вы не собираетесь этим заниматься, лучшее, что можно посоветовать, — пользоваться распространенными языками. В большинстве систем существуют обширные библиотеки, которые, безусловно, можно использовать там, где это уместно, но часто для упрощения клиентского кода и защиты от проблем можно строить абстракции, легко переносимые в другие языки. Ваша главная цель — разрабатывать такие типы данных, чтобы большая часть работы выполнялась на уровне абстракции, соответствующем решаемой задаче.

Вопросы и ответы

В. Что произойдет, если попытаться обратиться к закрытой переменной экземпляра или методу экземпляра из класса в другом файле?

О. Вы получите ошибку компиляции, в которой говорится, что данная переменная экземпляра или метод экземпляра являются закрытыми, и доступ к ним разрешен только в пределах класса.

В. Переменные экземпляра в классе `Complex` являются закрытыми, но когда я выполняю метод `plus()` для объекта `Complex` вызовом `a.plus(b)`, я могу обратиться к переменным экземпляров не только `a`, но и `b`. Разве переменные экземпляра `b` не должны быть недоступными?

О. Доступность определяется на уровне класса, а не на уровне экземпляров. Объявление переменной экземпляра с ключевым словом `private` означает, что она недоступна напрямую из других классов. Однако методы класса `Complex` могут обра-

щаться (для чтения и записи) к переменным любого экземпляра этого класса. Было бы неплохо иметь более жесткое ограничение доступа (допустим, `superprivate`), определяющее детализацию на уровне экземпляров, чтобы к переменным экземпляров мог обращаться только тот объект, для которого вызывается метод, но в Java такая возможность не предусмотрена.

В. Методу `times()` в `Complex` (листинг 3.3.1) нужен конструктор, который получает в качестве аргументов полярные координаты. Как добавить такой конструктор?

О. Никак, потому что уже существует конструктор, получающий два аргумента с плавающей точкой. В альтернативном решении можно было бы создать в API два «фабричных» метода `createRect(x, y)` и `createPolar(r, theta)`, которые создают и возвращают новые объекты. Такое решение лучше, потому что оно позволяет клиенту создавать объекты с указанием как декартовых, так и полярных координат. Этот пример показывает, что при разработке типа данных желательно думать более чем об одной реализации.

В. Существует ли связь между типом данных `Vector` (листинг 3.3.3), определеным в этом разделе, и типом данных Java `java.util.Vector`?

О. Нет. Мы используем это имя, потому что термин «вектор» находит употребление как в линейной алгебре, так и в вычислительных методах.

В. Что должен делать метод `direction()` в программе `Vector` (листинг 3.3.3), если вызвать его с вектором из одних нулей?

О. В полноценном API должно быть определено поведение каждого метода в каждой ситуации. В таком случае будет правильно инициировать исключение или вернуть `null`.

В. Что такое «устаревший метод»?

О. Метод, который уже не поддерживается в полной мере, но сохраняется в API для обеспечения совместимости. Например, в Java когда-то входил метод `Character.isSpace()`, и программисты писали программы, зависевшие от поведения этого метода. Когда проектировщики Java позднее решили поддержать дополнительные символы-пропуски Юникода, они не могли изменить поведение `isSpace()` без нарушения работоспособности клиентских программ. Чтобы решить проблему, они добавили новый метод `Character.isWhiteSpace()` и объявили старый метод *устаревшим*. С течением времени это начинает основательно загромождать API.

В. Что не так со следующей реализацией `equals()` для `Complex`?

```
public boolean equals(Complex that)
{
   return (this.re == that.re) && (this.im == that.im);
}
```

В. Этот код *перегружает* метод `equals()` вместо того, чтобы *переопределять* его. Иначе говоря, он определяет новый метод с именем `equals()`, получающий аргу-

мент типа `Complex`. Этот перегруженный метод отличается от унаследованного метода `equals()`, получающего аргумент типа `Object`. В некоторых ситуациях, например с библиотекой `java.util.HashMap`, рассмотренной в разделе 4.4, — вместо перегруженного метода будет вызываться унаследованный, что приводит к неожиданному поведению.

В. Что не так со следующей реализацией `hashCode()` для `Complex`?

```
public int hashCode()
{  return -17;  }
```

О. Формально она удовлетворяет контракту `hashCode()`: если два объекта равны, для них возвращаются одинаковые хеш-коды. Однако такая реализация создаст проблемы с быстродействием, потому что предполагается, что `Math.abs(x.hashCode() % m)` делит `n` объектов типа `Complex` на `m` групп приблизительно одинакового размера.

В. Может ли интерфейс иметь конструкторы?

О. Нет, потому что создать экземпляр интерфейса невозможно; вы можете только создавать объекты класса, реализующего этот интерфейс. Интерфейс может включать константы, сигнатуры методов, методы по умолчанию, статические методы и вложенные типы, но эти возможности выходят за рамки книги.

В. Может ли класс быть прямым субклассом для более чем одного класса?

О. Нет. Каждый класс (кроме `Object`) является прямым субклассом одного и только одного суперкласса. Это называется *простым* (*одиночным*) *наследованием*; в некоторых других языках (прежде всего C++) поддерживается *множественное наследование*, при котором класс может быть прямым субклассом двух и более суперклассов.

В. Может ли класс реализовать несколько интерфейсов?

О. Да. Для этого следует перечислить интерфейсы, разделенные запятыми, после ключевого слова `implements`.

В. Может ли тело лямбда-выражения содержать более одной команды?

О. Да, тело может содержать блок команд с объявлениями переменных, циклами и ветвлениями. В таких случаях для определения значения, возвращаемого лямбда-выражением, необходимо явно включить команду `return`.

В. В некоторых случаях лямбда-выражение просто вызывает метод из другого класса. Существует ли сокращенная запись для таких ситуаций?

О. Да, *ссылка на метод* — компактное, удобочитаемое лямбда-выражение для метода, уже обладающего именем. Например, в качестве сокращения для лямбда-выражения `x -> Gaussian.pdf(x)` может использоваться ссылка на метод `Gaussian::pdf`. За дополнительной информацией обращайтесь на сайт книги.

Упражнения

3.3.1. Напишите представление для момента времени, используя переменную `int` для хранения количества секунд с 1 января 1970 года. Когда в программе, использующей такое представление, сработает «бомба замедленного действия»? Как продолжить работу, когда это произойдет?

3.3.2. Создайте тип данных `Location` для представления точки поверхности Земли в сферических координатах (широта/долгота). Включите методы для генерирования случайной точки на поверхности Земли, разбора координат в формате `"25.344 N, 63.5532 W"` и вычисления расстояния (ортодромическое расстояние) по дуге большого круга между двумя точками.

3.3.3. Разработайте реализацию `Histogram` (листинг 3.2.3), использующую класс `Counter` (листинг 3.3.2).

3.3.4. Приведите реализацию `minus()` для `Vector`, основанную исключительно на других методах `Vector`, таких как `direction()` и `magnitude()`.

Ответ:

```
public Vector minus(Vector that)
{  return this.plus(that.scale(-1.0));  }
```

К достоинствам таких реализаций можно отнести то, что они ограничивают объем кода, требующего проверки, к недостаткам — возможную неэффективность. В данном случае оба метода `plus()` и `times()` создают новые объекты `Vector`, поэтому, возможно, лучше воспользоваться реализацией с копированим кода `plus()` и заменой знака «минус» знаком «плюс».

3.3.5. Реализуйте для класса `Vector` метод `main()`, который выполняет модульное тестирование своих методов.

3.3.6. Создайте тип данных для представления трехмерной частицы с текущей позицией (r_x,r_y,r_z), массой (m) и скоростью (v_x,v_y,v_z). Включите метод для вычисления кинетической энергии частицы, которая равна $1/2m(v_x^2 + v_y^2 + v_z^2)$. Используйте класс `Vector` (листинг 3.3.3).

3.3.7. Если вы хорошо знаете физику, разработайте альтернативную реализацию типа данных из предыдущего упражнения. В переменной экземпляра этой реализации должен храниться *импульс* (p_x, p_y, p_z).

3.3.8. Реализуйте тип данных `Vector2D` для двумерных векторов с таким же API, как и у `Vector`, за исключением того, что конструктор должен иметь два аргумента типа `double`. Используйте для переменных экземпляров два значения `double` (вместо массива).

3.3.9. Реализуйте тип данных `Vector2D` из предыдущего упражнения с использованием одного значения `Complex` как единственной переменной экземпляра.

3.3.10. Докажите, что скалярное произведение двух двумерных единичных векторов равно косинусу угла между ними.

3.3.11. Реализуйте тип данных `Vector3D` для трехмерных векторов с таким же API, как и у `Vector`, за исключением того, что конструктор должен иметь три аргумента типа `double`. Также добавьте метод для вычисления *векторного произведения*; векторное произведение двух векторов представляет собой другой вектор, определяемый формулой

$$\mathbf{a} \times \mathbf{b} = \mathbf{c}\,|\,\mathbf{a}\,|\,|\,\mathbf{b}\,|\sin\theta$$

где **c** — единичный нормальный вектор, перпендикулярный **a** и **b**, а θ — угол между **a** и **b**. В декартовых координатах векторное произведение вычисляется по следующей формуле:

$$(a_0, a_1, a_2) \times (b_0, b_1, b_2) = (a_1 b_2 - a_2 b_1, a_2 b_0 - a_0 b_2, a_0 b_1 - a_1 b_0).$$

Векторное произведение встречается в определении момента силы, момента вращения и оператора ротора вектора. Кроме того, $|\,\mathbf{a} \times \mathbf{b}\,|$ — площадь параллелограмма со сторонами **a** и **b**.

3.3.12. Переопределите метод `equals()` класса `Charge` (листинг 3.2.6), чтобы два объекта `Charge` считались равными, если они обладают одинаковыми значениями позиции и заряда. Переопределите метод `hashCode()`, используя решение с `Objects.hash()`, описанное в этом разделе.

3.3.13. Переопределите методы `equals()` и `hashCode()` класса `Vector` (листинг 3.3.3), чтобы два объекта `Vector` считались равными, если они обладают одинаковой длиной и их соответствующие координаты равны.

3.3.14. Добавьте в класс `Vector` метод `toString()`, который возвращает компоненты вектора, разделенные запятыми и заключенные в круглые скобки.

3.3.15. Добавьте в класс `Sketch` метод `toString()`, который возвращает строковое представление единичного вектора для заданного конспекта.

3.3.16. Опишите поведение вызовов методов `x.add(y)` и `y.add(x)` класса `Vector` (листинг 3.3.3), если `x` соответствует вектору (1, 2, 3), а `y` — вектору (5, 6).

3.3.17. Используйте диагностические утверждения и исключения для разработки реализации `Rational` (см. упражнение 3.2.7), защищенной от переполнения.

3.3.18. Добавьте в класс `Counter` (листинг 3.3.2) код, инициирующий исключение `IllegalArgumentException`, если клиент пытается создать объект `Counter` с отрицательным значением `max`.

Упражнения по проектированию типов данных

Упражнения из этого списка помогут вам накопить опыт разработки типов данных. Для каждой задачи спроектируйте одну или несколько версий API с реализациями,

проверяющими принятые решения по проектированию на типичном клиентском коде. Некоторые упражнения потребуют знания определенной предметной области или поиска информации в Интернете.

3.3.19. *Статистика.* Разработайте тип данных для статистической обработки множества вещественных чисел (например, результатов замеров). Предоставьте метод для добавления новых точек данных и методы для получения количества точек, математического ожидания, среднеквадратичного отклонения и дисперсии. Разработайте две реализации: у одной в экземплярах хранится количество точек, сумма значений и сумма квадратов значений, а у другой — массив всех точек. Для простоты максимальное количество точек может передаваться конструктору. Вероятно, первая реализация будет работать намного быстрее и занимать существенно меньше памяти, но она также с большей вероятностью подвержена ошибкам округления. Качественно спроектированное альтернативное решение приведено на сайте книги.

3.3.20. *Геном.* Разработайте тип данных для хранения генома организма. Биологи часто абстрагируют геном последовательностью нуклеотидов (A, C, G или T). Тип данных должен поддерживать методы `addCodon(char c)` и `nucleotideAt(int i)`, а также метод `isPotentialGene()` (см. листинг 3.1.1). Разработайте три реализации. Первая должна использовать одну переменную экземпляра типа `String`, а реализация `addCodon()` должна базироваться на конкатенации. Каждый вызов метода занимает время, пропорциональное длине текущего генома. Вторая реализация должна использовать массив символов, длина которого удваивается при заполнении. Третья реализация должна использовать массив булевского типа, где каждый кодон представляется двумя битами, а длина массива удваивается при заполнении.

3.3.21. *Время.* Разработайте тип данных для времени суток. Предоставьте клиентские методы, которые возвращают текущий час, минуту и секунду, а также методы `toString()`, `equals()` и `hashCode()`. Разработайте две реализации: одна хранит время в отдельной переменной типа `int` (количество секунд с полуночи), а другая хранит три значения `int`: для секунд, минут и часов.

3.3.22. *Код VIN.* Разработайте тип данных для идентификационного номера автомобиля VIN (Vehicle Identiication Number). VIN описывает производителя, модель и год сборки, а также другие атрибуты автомобилей, автобусов и грузовиков в США.

3.3.23. *Генерирование псевдослучайных чисел.* Разработайте тип данных для генерирования псевдослучайных чисел. Иначе говоря, преобразуйте `StdRandom` в тип данных. Вместо использования `Math.random()` ваш тип данных должен базироваться на *линейном конгруэнтном генераторе*[1]. Этот метод появился на заре развития вычислительной техники и также является классическим приме-

[1] Библиотечные генераторы псевдослучайных чисел тоже очень часто используют именно этот метод. — *Примеч. науч. ред.*

ром использования хранимых состояний, реализованным в типе данных. Чтобы сгенерировать псевдослучайные значения int, храните целочисленное значение x (значение последнего возвращенного «случайного» числа). Каждый раз, когда клиент запрашивает новое значение, верните $a * x + b$ с соответствующим образом выбранными значениями a и b (игнорируя переполнение). Используйте дополнительные арифметические операции для преобразования этих значений в «случайные» значения других типов данных. Как рекомендует Дональд Кнут, используйте значения 3141592621 для a и 2718281829 для b. Предоставьте конструктор, позволяющий клиенту начинать последовательность с конкретного значения int. Эта возможность четко показывает, что числа вовсе не являются случайными (хотя и обладают многими свойствами случайных чисел), но это же и упрощает отладку программ, так как клиенты могут повторно получать одни и те же числа.

Упражнения повышенной сложности

3.3.24. *Инкапсуляция.* Является ли следующий класс неизменяемым?

```
import java.util.Date;
public class Appointment
{
   private Date date;
   private String contact;

   public Appointment(Date date)
   {
      this.date = date;
      this.contact = contact;
   }
   public Date getDate()
   {  return date;  }
}
```

Ответ: Нет. Класс Java `java.util.Date` является изменяемым. Метод `setDate(seconds)` изменяет значение объекта, для которого он был вызван, количеством миллисекунд с 1 января 1970 года, 00:00:00 GMT. Это обстоятельство имеет неприятное следствие: при получении даты вызовом `date = getDate()` клиентская программа может затем вызвать `date.setDate()` и изменить дату в типе `Appointment`, что может привести к конфликту. В типе данных за ссылками на изменяемые объекты необходимо тщательно следить, потому что вызывающая сторона может изменить их состояние. Одно из возможных решений — создание «защитной» копии `Date` перед ее возвращением `new Date(date.getTime())` и еще одной «защитной» копии при сохранении `this.date = new Date(date.getTime())`. Многие программисты считают изменяемость `Date` в языке Java недостатком проектирования. (`GregorianCalendar` — более современная библиотека Java для хранения дат, но она тоже не обеспечивает неизменяемости).

3.3.25. *Дата.* Разработайте реализацию API класса `java.util.Date`, которая является неизменяемой, а следовательно, исправляет недостатки, отмеченные в предыдущем упражнении.

3.3.26. *Календарь.* Разработайте API классов `Appointment` и `Calendar`, которые могут использоваться для отслеживания встреч (по дням) в календарном году. Предоставьте клиентам возможность назначать встречи без конфликтов и получать информацию о текущих встречах.

3.3.27. *Векторное поле.* Векторное поле связывает вектор с каждой точкой евклидова пространства. Напишите версию программы `Potential` (листинг 3.2.23), которая получает размер сетки n, вычисляет значение `Vector` потенциала, обусловленного присутствием точечных зарядов в каждой из $n \times n$ точек сетки, размещенных с постоянным шагом, и строит единичный вектор в направлении результирующего электрического поля в каждой точке. (Измените класс `Charge` так, чтобы он возвращал `Vector`.)

3.3.28. *«Портрет» генома.* Напишите функцию `hash()`, которая получает в аргументе k-грамму (строку длины k), состоящую из символов A, C, G или T, и возвращает значение `int` в диапазоне от 0 до $4^k - 1$. Это значение соответствует интерпретации строки как числа в системе счисления с основанием 4, в которой {A, C, G, T} заменяются на {0, 1, 2, 3} соответственно. Затем напишите функцию `unHash()`, которая отменяет преобразование. Используйте свои методы для создания класса `Genome`, который похож на `Sketch` (листинг 3.3.4), но при этом базируется на точном подсчете k-грамм в геномах. Наконец, напишите версию `CompareDocuments` (листинг 3.3.5) для объектов `Genome` и воспользуйтесь ею для поиска сходства в файлах с геномами на сайте книги.

3.3.29. *«Срезы» документов.* Выберите интересующий вас набор документов на сайте книги (или воспользуйтесь собственной подборкой) и выполните `CompareDocuments` с разными значениями аргументов командной строки k и d, чтобы понять, как они влияют на вычисления.

3.3.30. *Поиск мультимедийных материалов.* Разработайте стратегии получения аналогичных «срезов» для звука и графики. Используйте их для выявления интересных примеров сходства между песнями в вашей фонотеке и фотографиями в фотоальбоме на вашем компьютере.

3.3.31. *Анализ данных.* Напишите рекурсивную программу, которая ищет в Интернете, начиная со страницы, заданной первым аргументом командной строки, страницы, похожие на страницу, заданную вторым аргументом командной строки. Поиск осуществляется следующим образом: программа обрабатывает имя, открывает поток ввода, выполняет `readAll()`, строит конспект и выводит имя, если расстояние до целевой страницы больше порогового значения, заданного третьим аргументом командной строки. Затем программа ищет в странице все строки, начинающиеся с префикса *http://*, и (рекурсивно) обрабатывает страницы с этими именами. *Примечание*: количество страниц, читаемых программой, может быть очень большим!

3.4. Пример: моделирование задачи n тел

Некоторые примеры, рассматривавшиеся в главах 1 и 2, лучше выражаются в виде объектно-ориентированных программ. Например, BouncingBall (листинг 3.1.9) естественно выражается в виде типа данных, значениями которого являются позиция и скорость шара, и клиента, который вызывает методы экземпляра для перемещения и рисования шара. Например, такой тип данных открывает возможность создания клиентов, которые моделируют движение нескольких шаров одновременно (см. упражнение 3.4.1). Точно так же наш пример Percolation из раздела 2.4 может стать интересным упражнением в области объектно-ориентированного программирования, как и пример со случайным серфингом из раздела 1.6. Первому примеру будет посвящено упражнение 3.4.8, ко второму мы вернемся в разделе 4.5. А в этом разделе будет рассмотрен другой пример, демонстрирующий методологию объектно-ориентированного программирования.

Пусть требуется написать программу, динамически моделирующую движение *n* тел в условиях взаимного гравитационного притяжения. Эта задача была впервые сформулирована Исааком Ньютоном более 350 лет назад, и она продолжает интенсивно изучаться в наши дни.

Как выглядит множество значений и какие операции с этими значениями поддерживаются? Одна из причин, по которым эта задача рассматривается как интересный и убедительный пример объектно-ориентированного программирования, заключается в том, что в ней есть прямое и естественное соответствие между физическими объектами реального мира и абстрактными объектами в программе. Переход от решения задач последовательностью выполняемых команд к проектированию типов данных обычно вызывает у новичков проблемы. Но с накоплением практического опыта вы поймете, насколько этот подход полезен для решения вычислительных задач.

Вспомните некоторые базовые концепции и формулы из школьного курса физики. Собственно, вам даже не обязательно полностью понимать эти формулы, чтобы понять код, — благодаря *инкапсуляции* эти формулы задействованы только в нескольких методах, а благодаря *абстракции данных* большая часть кода выглядит логично и осмысленно. В каком-то смысле это идеальная объектно-ориентированная программа.

Моделирование задачи N тел

Моделирование движения шара из раздела 1.5 основано на первом законе Ньютона: движущееся тело продолжает двигаться с постоянной скоростью, если на него не действует внешняя сила. Дополнение модели вторым законом Ньютона (объясняющим, как внешние силы влияют на движение) приводит к базовой задаче, которая веками занимала умы ученых. Имеется система из *n* тел, находящихся под взаимным воздействием гравитационных сил; требуется описать движение тел. Эта

базовая модель используется в задачах в различных областях — от астрофизики до молекулярной динамики.

В 1687 году Ньютон сформулировал в своих знаменитых «Началах» принципы, которым подчиняется движение двух тел под воздействием их взаимного гравитационного притяжения. Однако Ньютону не удалось разработать математическое описание движения трех тел. С тех пор ученые доказали не только то, что такого описания в контексте элементарных функций не существует, но и то, что в зависимости от исходных значений возможно хаотичное поведение. Для изучения таких задач ученым пришлось прибегнуть к имитационному моделированию. В этом разделе мы разработаем объектно-ориентированную программу, реализующую такую модель. Ученых интересуют подобные задачи с высокой степенью точности для большого количества тел, так что наше решение в лучшем случае дает начальное представление о теме. Тем не менее вас может удивить, насколько просто строятся реалистичные анимации для сложных движений.

Тип данных Body

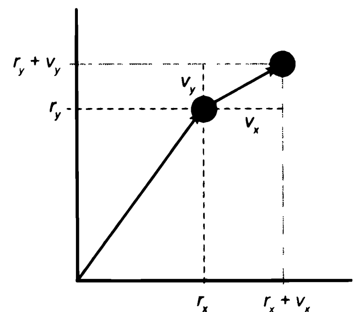
В программе `BouncingBall` (листинг 3.1.9) смещение от начала координат хранится в переменных `rx` и `ry` типа `double`, а скорость — в переменных `vx` и `vy` типа `double`. Перемещение шара за единицу времени осуществляется следующими командами:

```
rx = rx + vx;
ry = ry + vy;
```

При использовании класса `Vector` (листинг 3.3.3) позиция может храниться в переменной `r` типа `Vector`, а скорость — в переменной `v` типа `Vector`, тогда перемещение тела за `dt` единиц времени выполняется одной строкой:

```
r = r.plus(v.times(dt));
```

В модели n тел задействовано несколько операций такого рода, поэтому наше первое решение будет работать с объектами `Vector` вместо отдельных компонентов x и y. При таком решении код получается более четким, более компактным и более гибким, чем в альтернативном решении с отдельными компонентами. Класс Java `Body` (листинг 3.4.1) использует `Vector` для реализации типа данных для движения тел. Его переменные экземпляра — две переменные `Vector` с позицией и скоростью тела, а также переменная типа `double` для хранения массы. Определенные для типа данных операции позволяют клиентам перемещать тело и отображать его (а также вычислять вектор силы, обусловленный гравитационным воздействием другого тела), они определяются в следующем API:

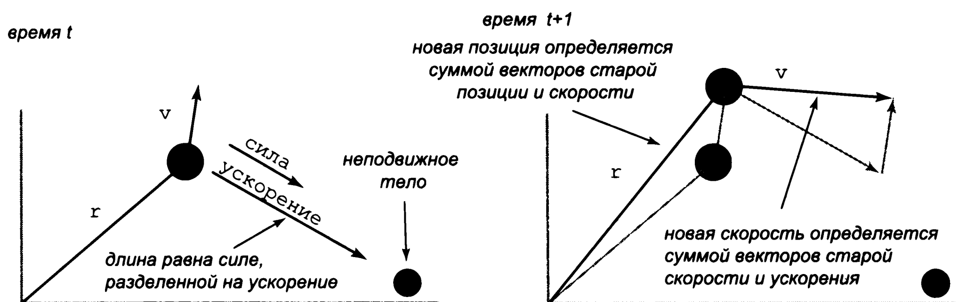


Суммирование векторов при перемещении

public class Body	
`Body(Vector r, Vector v, double mass)`	
`void move(Vector f, double dt)`	применяет силу f и перемещает тело в течение dt секунд
`void draw()`	отображает шар
`Vector forceFrom(Body b)`	вектор силы между этим телом и b

API для тел, двигающихся по законам Ньютона (см. листинг 3.4.1)

Формально позиция тела (смещение от точки координат) не является вектором (это точка пространства, не обладающая направлением или длиной), но ее удобно представлять в виде `Vector`, потому что операции `Vector` обеспечивают компактность кода, необходимого для перемещения тела, как обсуждалось ранее. А при перемещении тела необходимо изменить не только его позицию, но и скорость.



Движение вблизи от неподвижного тела

Сила и движение

Второй закон Ньютона гласит, что действующая на тело сила (вектор) равна произведению массы (скалярная величина) и его ускорения (также вектор): $F = ma$. Иначе говоря, для вычисления ускорения тела следует взять силу и разделить ее на массу. В классе `Body` сила задается аргументом `f` (тип `Vector`) метода `move()`, так что мы можем сначала вычислить вектор ускорения, разделив ее на массу (скалярное значение, хранящееся в переменной экземпляра `double`), а затем вычислить изменение скорости, прибавив к нему изменение вектора за интервал времени (по аналогии с тем, как мы использовали скорость для изменения позиции). Этот закон элементарно преобразуется в следующий код обновления позиции и скорости тела под воздействием заданного вектора силы `f` за время `dt`:

```
Vector a = f.scale(1/mass);
v = v.plus(a.scale(dt));
r = r.plus(v.scale(dt));
```

Листинг 3.4.1. Гравитационное тело

```
public class Body
{
   private Vector r;
   private Vector v;
   private final double mass;

   public Body(Vector r0, Vector v0, double m0)
   {  r = r0; v = v0; mass = m0;  }

   public void move(Vector force, double dt)
   {  // Обновление позиции и скорости.
      Vector a = force.scale(1/mass);
      v = v.plus(a.scale(dt));
      r = r.plus(v.scale(dt));
   }

   public Vector forceFrom(Body b)
   {  // Вычисление силы, действующей на тело со стороны b.
      Body a = this;
      double G = 6.67e-11;
      Vector delta = b.r.minus(a.r);
      double dist = delta.magnitude();
      double magnitude = (G * a.mass * b.mass)
                         / (dist * dist);
      Vector force = delta.direction().scale(magnitude);
      return force;
   }

   public void draw()
   {
       StdDraw.setPenRadius(0.0125);
       StdDraw.point(r.cartesian(0), r.cartesian(1));
   }
}
```

r	позиция
v	скорость
mass	масса

force	сила, действующая на тело
dt	временной интервал
a	ускорение

a	это тело
b	другое тело
c	постоянная всемирного тяготения
delta	вектор из b в a
dist	расстояние от b до a
magnitude	длина вектора силы

Тип данных предоставляет операции, необходимые для моделирования движения любых физических тел (как планет, так и атомных частиц). Это изменяемый тип, в переменных экземпляров которого хранятся позиция и скорость тела, которые изменяются в методе move() под воздействием внешних сил (масса тела не изменяется.) Метод forceFrom() возвращает вектор силы.

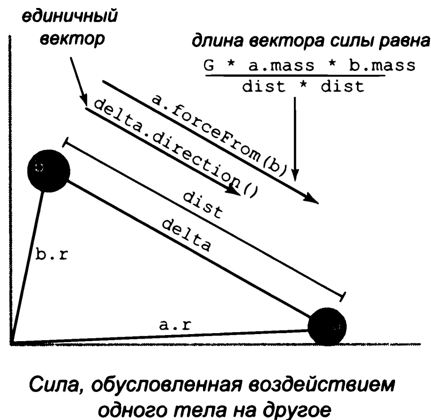
Этот код входит в метод экземпляра `move()` в `Body`: он изменяет его значения, чтобы отразить последствия воздействия силы за заданный интервал времени: тело перемещается, его скорость изменяется. Вычисления предполагают, что ускорение на этом отрезке времени остается постоянным.

Силы, обусловленные воздействием тел

Вычисление силы, обусловленной воздействием одного тела на другое, инкапсулировано в методе экземпляров `forceFrom()` класса `Body`. Метод получает объект

Body в своем аргументе и возвращает Vector. В основе вычисления лежит закон всемирного тяготения Ньютона, согласно которому два тела притягиваются друг к другу с силой, прямо пропорциональной произведению их масс и обратно пропорциональной квадрату расстояния между ними (коэффициентом является постоянная всемирного тяготения G, равная $6{,}67\times10^{-11}$ Н × м^2/кг^2), в направлении линии, соединяющей эти два тела. Закон преобразуется в следующий код вычисления a.forceFrom(b):

```
double G = 6.67e-11;
Vector delta = b.r.minus(a.r);
double dist = delta.magnitude();
double magnitude = (G * a.mass * b.mass) / (dist * dist);
Vector force = delta.direction().scale(magnitude);
return force;
```



Сила, обусловленная воздействием одного тела на другое

Длина вектора силы хранится в переменной magnitude типа double, а *направление* вектора силы совпадает с направлением вектора разности между позициями двух тел. Вектор силы force является единичным вектором, масштабированным по длине.

Тип данных Universe

Тип данных Universe (листинг 3.4.2) реализует следующий API:

public class Universe	
Universe(String filename)	инициализирует объект Universe по имени файла
void increaseTime(double dt)	моделирует прохождение dt секунд
void draw()	рисует пространство задачи

API пространства задачи (см. листинг 3.4.2)

Значения этого типа данных определяют пространство задачи (размер, количество тел, массив тел) и две операции с типом данных: `increaseTime()` изменяет позиции (и скорости) всех тел, а `draw()` рисует все тела. Ключевую роль в моделировании задачи *n* тел играет реализация `increaseTime()` в `Universe`. В вычислениях основное место занимает вложенный цикл, который вычисляет вектор силы, описывающий силу гравитационного взаимодействия между двумя телами. В программе применяется принцип суперпозиции, согласно которому вектор, представляющий результат действия всех сил, вычисляется суммированием всех векторов сил. После вычисления всех сил вызывается метод `move()` для каждого тела; этот метод применяет вычисленную силу в течение фиксированного промежутка времени.

```
% more 2body.txt
2
5.0e10
0.0e00  4.5e10  1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

% more 3body.txt
3
1.25e11
0.0e00  0.0e00 0.05e04 0.0e00 5.97e24
0.0e00  4.5e10  3.0e04 0.0e00 1.989e30
0.0e00 -4.5e10 -3.0e04 0.0e00 1.989e30

% more 4body.txt
4
5.0e10
-3.5e10 0.0e00 0.0e00  1.4e03 3.0e28
-1.0e10 0.0e00 0.0e00  1.4e04 3.0e28
 1.0e10 0.0e00 0.0e00 -1.4e04 3.0e28
 3.5e10 0.0e00 0.0e00 -1.4e03 3.0e28
```


Примеры формата файлов Universe

Формат файла

Как обычно, мы используем решение, управляемое данными, при котором входные данные загружаются из файла. Конструктор читает параметры пространства задачи и описания тел из файла, содержащего следующую информацию:

- количество тел;
- радиус пространства задачи;
- позиция, скорость и масса каждого тела.

Как обычно, все величины задаются в стандартных единицах СИ (постоянная всемирного тяготения G приводилась ранее в коде). С четко определенным форматом файла код конструктора `Universe` получается достаточно прямолинейным.

```
public Universe(String filename)
{
   In in = new In(filename);
   n = in.readInt();
   radius = in.readDouble();
   StdDraw.setXscale(-radius, +radius);
   StdDraw.setYscale(-radius, +radius);

   bodies = new Body[n];
   for (int i = 0; i < n; i++)
   {
      double rx = in.readDouble();
      double ry = in.readDouble();
      double[] position = { rx, ry };
      double vx = in.readDouble();
      double vy = in.readDouble();
      double[] velocity = { vx, vy };
      double mass = in.readDouble();
      Vector r = new Vector(position);
      Vector v = new Vector(velocity);
      bodies[i] = new Body(r, v, mass);
   }
}
```

Каждый объект `Body` описывается пятью значениями `double`: координатами x и y его позиции, компонентами x и y его начальной скорости и его массой.

Тестовый клиент `main()` класса `Universe` представляет собой программу, управляемую данными и моделирующую движение n тел, находящихся в условиях взаимного притяжения. Конструктор создает массив из n объектов `Body`; он читает исходную позицию каждого тела, его исходную скорость и массу из файла, имя которого передается в качестве аргумента. Метод `increaseTime()` вычисляет силы, воздействующие на каждое тело, и использует эту информацию для обновления ускорения, скорости и позиции каждого тела за интервал времени `dt`. Тестовый клиент `main()` вызывает конструктор, а затем моделирует движение, вызывая в цикле методы `increaseTime()` и `draw()`.

На сайте книги представлены разнообразные файлы, определяющие разные варианты пространства задачи. Попробуйте понаблюдать за движением тел при помощи `Universe`. Даже при малом количестве тел становится понятно, почему у Ньютона возникали проблемы с поиском формул, описывавших их траектории. На странице 475 приведены результаты выполнения `Universe` для примеров с 2, 3 и 4 телами для файлов данных, приведенных выше. В конфигурации с 2 телами наблюдается взаимно-орбитальное движение, конфигурация с 3 телами выглядит хаотично: «луна» переходит между двумя планетами, вращающимися по орбитам, а в конфигурации с 4 телами ситуация относительно проста: медленное вращение двух пар взаимно-орбитальных тел. Чтобы создать статические изображения на иллюстрациях, мы изменили код `Universe` и `Body`, чтобы тела рисовались белым и черным цветом на сером фоне, как в программе `BouncingBall` (листинг 3.1.9): динамические изображения, которые будут отображаться при запуске `Universe`

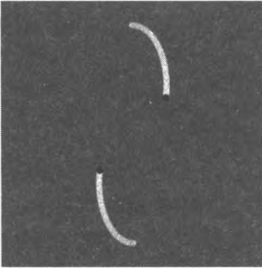
Листинг 3.4.2. Моделирование системы n тел

```
public class Universe
{
   private final double radius;
   private final int n;
   private final Body[] bodies;

   public void increaseTime(double dt)
   {
      Vector[] f = new Vector[n];
      for (int i = 0; i < n; i++)
         f[i] = new Vector(new double[2]);
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            if (i != j)
               f[i] = f[i].plus(bodies[i].forceFrom(bodies[j]));
      for (int i = 0; i < n; i++)
         bodies[i].move(f[i], dt);
   }

   public void draw()
   {
      for (int i = 0; i < n; i++)
         bodies[i].draw();
   }

   public static void main(String[] args)
   {
      Universe newton = new Universe(args[0]);
      double dt = Double.parseDouble(args[1]);
      StdDraw.enableDoubleBuffering();
      while (true)
      {
         StdDraw.clear();
         newton.increaseTime(dt);
         newton.draw();
         StdDraw.show();
         StdDraw.pause(20);
      }
   }
}
```

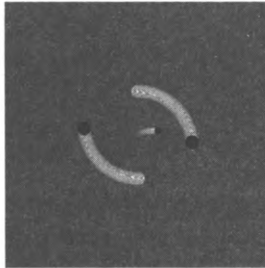
radius	радиус пространства задачи
n	количество тел
bodies[]	массив тел

Программа моделирует движение тел в пространстве задачи, определяемом в файле (первый аргумент командной строки), с заданным приращением времени (второй аргумент командной строки). Реализация конструктора приведена в тексте.

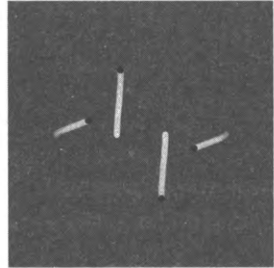
100 шагов



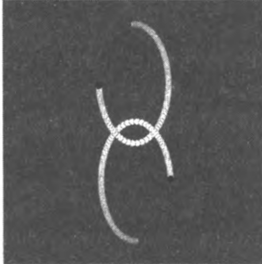
150 шагов



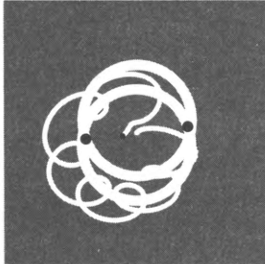
100 шагов



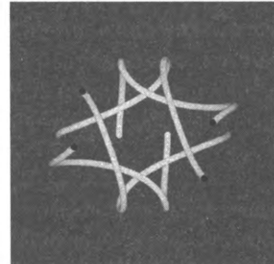
150 шагов



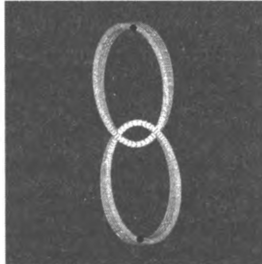
880 шагов



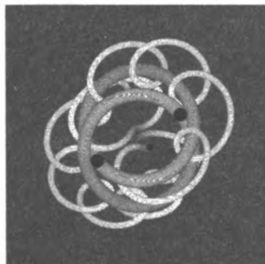
500 шагов



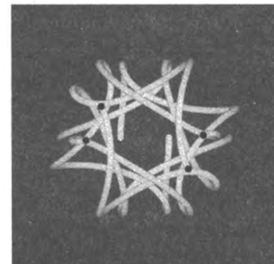
1000 шагов



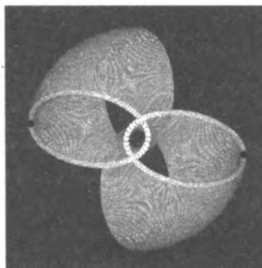
1600 шагов



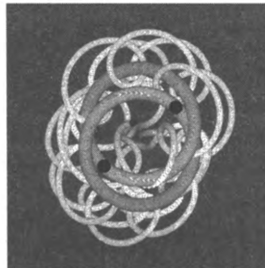
1000 шагов



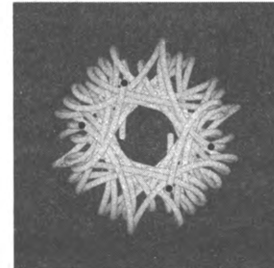
10 000 шагов



3100 шагов



3000 шагов



Моделирование конфигураций с 2 телами (левый столбец), 3 телами (средний столбец) и 4 телами (правый столбец)

в исходном виде, создают реалистичное ощущение небесных тел, находящихся в орбитальном движении. А если запустить `Universe` в примере с большим количеством тел, вы поймете, почему моделирование играет столь важную роль в работе ученых, пытающихся понять сложную задачу. Модель с n телами чрезвычайно многогранна, и эксперименты с готовыми файлами убедят вас в этом.

```
планетарный масштаб
% more 2body.txt
2
5.0e10
0.0e00  4.5e10  1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30
```

```
субатомный масштаб
% more 2bodyTiny.txt
2
5.0e-10
0.0e00  4.5e-10  1.0e-16 0.0e00 1.5e-30
0.0e00 -4.5e-10 -1.0e-16 0.0e00 1.5e-30
```

Наверняка вам захочется создать собственную мини-Вселенную (см. упражнение 3.4.7). Основные сложности при создании файла данных возникают с масштабированием чисел, чтобы радиус пространства задачи, временной масштаб, массы и скорости тел обеспечивали интересное поведение. Можно наблюдать за поведением планет, вращающихся вокруг звезды, или субатомных частиц, взаимодействующих друг с другом, но из попытки изучить взаимодействие планеты с субатомной частицей ничего не выйдет. Кроме того, при работе с собственными данными вы с большой вероятностью столкнетесь с тем, что одни тела улетают в бесконечность, а другие поглощаются третьими, — но это все равно интересно!

В этом примере мы стремились продемонстрировать полезность типов данных, а не предоставить код моделирования задачи n тел для практического применения. Если ученые захотят использовать это решение для изучения природных явлений, им придется учесть ряд потенциальных проблем. Первая проблема — точность: погрешности вычислений нередко накапливаются и приводят к драматическим последствиям, не наблюдаемым в природе. Например, наш код не предпринимает никаких особых действий, когда тела (почти) сталкиваются. Вторая проблема — эффективность: метод `move()` из `Universe` выполняется за время, пропорциональное n^2, поэтому для большого количества тел он не подойдет. Как и в геномике, для решения научных задач, связанных с задачей n тел, необходимо не только разбираться в предметной области задачи, но и понимать основные проблемы, давно изучаемые специалистами в области компьютерных технологий.

Для простоты мы работаем с двумерным пространством задачи. Это предположение реалистично только в том случае, если мы рассматриваем движение тел на плоскости. Но у того факта, что в основу реализации `Body` был заложен класс `Vector`, есть одно важное следствие: клиент может использовать *трехмерные векторы* для моделирования движения тел в трех измерениях (а на самом деле в любом коли-

честве измерений) без изменения кода! Метод `draw()` проецирует позицию тела на плоскость, определяемую первыми двумя измерениями.

Тестовый клиент `Universe` — всего лишь один из возможных вариантов; ту же базовую модель можно использовать в других ситуациях (например, с разными видами взаимодействий между телами). Например, вы можете наблюдать и измерить характеристики текущего движения некоторых существующих тел, а затем запустить моделирование в обратном направлении! Астрофизики применяют этот метод для изучения происхождения Вселенной. В науке мы пытаемся понять прошлое и спрогнозировать будущее; хорошая модель позволяет сделать и то и другое.

Вопросы и ответы

В. API класса `Universe` совсем невелик. Почему бы просто не реализовать этот код в тестовом клиенте `main()` класса `Body`?

О. Такая архитектура выражает представления многих людей о Вселенной: она была создана, и время в ней движется вперед. Наше решение проясняет код и обеспечивает максимальную гибкость в моделировании происходящих событий.

В. Почему `forceFrom()` является методом экземпляра? Разве не лучше сделать его статическим методом, получающим два объекта `Body` в аргументах?

О. Да, реализация `forceFrom()` в виде метода экземпляра — одна из возможных альтернатив, и статический метод, получающий в качестве аргументов два объекта `Body`, безусловно, является разумным решением. Некоторые программисты предпочитают полностью избегать статических методов в реализациях типов данных; еще один возможный вариант — хранить силу, действующую на каждый объект `Body`, в переменной экземпляра. Мы выбрали компромисс между этими двумя решениями.

Упражнения

3.4.1. Разработайте объектно-ориентированную версию программы `BouncingBall` (листинг 3.1.9). Включите конструктор, который запускает каждый шар в случайном направлении со случайной скоростью (в разумных пределах), и тестовое приложение-клиент, которое получает целое число n как аргумент командной строки и моделирует движение n шаров.

3.4.2. Добавьте в листинг 3.4.1 метод `main()` для модульного тестирования типа данных `Body`.

3.4.3. Измените класс `Body` (листинг 3.4.1), чтобы радиус круга, изображающего тело, был пропорционален его массе.

3.4.4. Что произойдет в пространстве задачи при отсутствии тяготения? Такая ситуация будет соответствовать тому, что метод `forceTo()` в `Body` всегда возвращает нуль-вектор.

3.4.5. Создайте тип данных `Universe3D` для моделирования трехмерных пространств задачи. Разработайте файл данных, который моделирует движение планет Солнечной системы вокруг Солнца.

3.4.6. Реализуйте класс `RandomBody`, который инициализирует свои переменные экземпляров (тщательно выбранными) случайными значениями вместо того, чтобы использовать конструктор, и клиент `RandomUniverse`, который получает один аргумент командной строки n и моделирует движение в случайном пространстве задачи с n телами.

Упражнения повышенной сложности

3.4.7. *Новая Вселенная.* Спроектируйте новую Вселенную с какими-нибудь необычными свойствами. Смоделируйте движение тел в этой Вселенной при помощи класса `Universe`. Вряд ли вам еще где-нибудь представится такая творческая возможность!

3.4.8. *Протекание.* Разработайте объектно-ориентированную версию программы `Percolation` (листинг 2.4.5). Тщательно продумайте архитектуру своей программы, прежде чем браться за работу, и будьте готовы обосновать свои решения.

Глава 4

Алгоритмы и структуры данных[1]

Эта глава посвящена фундаментальным структурам данных (и соответствующим им типам данных), которые становятся важнейшими элементами при конструировании самых разнообразных приложений. Также здесь приводятся рекомендации по их использованию независимо от того, собираетесь ли вы использовать готовые решения из библиотеки Java или разработать собственные версии, основанные на приведенном здесь коде.

Объекты могут содержать ссылки на другие объекты, что позволяет строить *связные структуры* произвольной сложности. Используя связные структуры и массивы, вы сможете строить структуры данных для хранения информации способом, обеспечивающим ее эффективную обработку соответствующими *алгоритмами*. В типе данных (*классе*) набор значений используется для построения структур данных, а методы, работающие с этими значениями, реализуют *алгоритмы*.

Алгоритмы и структуры данных, которые будут рассматриваться в этой главе, составляют комплекс знаний, разработанных за последние 50 лет. Эти знания формируют основу эффективного использования компьютеров в самых разнообразных ситуациях. Описанные здесь базовые методы играют ключевую роль в научных исследованиях в самых разных областях, от моделирования задачи n тел в физике до задач генетического секвенирования в биоинформатике; они заложены в основу многих коммерческих приложений, от баз данных до поисковых систем. А поскольку сфера применения вычислительных технологий продолжает расширяться, возрастает и важность этих базовых методов.

Сами алгоритмы и структуры данных становятся темой научных исследований. Соответственно сначала будет рассмотрен научный подход к анализу эффективности алгоритмов, который будет применяться в этой главе.

---

[1] Название этой главы можно рассматривать как своего рода отсылку к классической монографии Никлауса Вирта «Алгоритмы и структуры данных» (также «Алгоритмы + Структуры данных = Программы»). Работы со схожими названиями были и у других авторов. — *Примеч. науч. ред.*

4.1. Быстродействие

В этом разделе вы оцените принцип, лаконично выраженный еще в одной мантре, которую необходимо постоянно помнить при программировании: *учитывайте затраты.* У инженера это правило является частью его работы; у биолога или физика от затрат зависит то, за какие научные проблемы можно взяться; для бизнесмена или экономиста важность этого принципа не нуждается в объяснениях; наконец, для программиста от затрат зависит то, будет ли пользоваться его программный продукт спросом у пользователей.

Для оценки затрат, связанных с выполнением программ, применяется специальная *научная дисциплина* — набор методов, используемых учеными для получения знаний о реальном мире. Чтобы оценить затраты на выполнение программ, мы изучаем программы как таковые, применяя, в общем, те же *методы научных исследований*, что и для получения знаний о реальном мире. Также для создания точных математических моделей затрат применяются методы *математического анализа.*

Какие аспекты реального мира вы изучаете? В большинстве случаев нас интересует одна важнейшая характеристика: *время*. Каждый раз при запуске программы мы проводим эксперимент с участием реального мира: в сложной системе электронных схем происходит серия изменений состояния с огромным числом событий, которые в конечном итоге приводят к состоянию, результаты которого мы хотим интерпретировать. И хотя эти события разрабатываются в абстрактном мире программирования на языке Java, они определенно будут происходить в реальном мире. Сколько времени придется ожидать результата? Для нас очень важно, будет ли это время исчисляться миллисекундами, секундами, днями или неделями. Итак, мы хотим знать, как правильно контролировать ситуацию — запуск ракеты, построение моста или расщепление атома — научными методами.

С одной стороны, современные программы и среды программирования достаточно сложны; с другой стороны, они строятся на базе простого (но мощного) набора абстракций. То, что программа выдает один и тот же результат при каждом запуске, — маленькое чудо. Чтобы предсказать, сколько времени для этого понадобится, мы воспользуемся относительной простотой инфраструктуры, задействованной для построения программ. Вас удивит, как легко можно оценить затраты и спрогнозировать характеристики быстродействия многих из написанных вами программ.

Научный метод

Ниже приведена краткая сводка научного метода, состоящая из пяти шагов:

- *наблюдение* за некоторым аспектом реального мира;
- *выдвижение гипотезы* и построение на ее основе модели, соответствующей наблюдениям;
- *прогнозирование событий* в соответствии с гипотезой;
- *дополнительные наблюдения* для проверки прогноза;

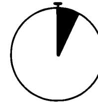
- *проверка* с повторением всей процедуры до тех пор, пока гипотеза не будет совпадать с наблюдениями.

Один из ключевых принципов научного метода требует, чтобы спроектированные эксперименты были воспроизводимыми, чтобы другие люди могли убедиться в правильности гипотезы. Кроме того, сформулированная гипотеза должна быть *опровергаемой* — вы должны иметь возможность убедиться в том, что гипотеза ошибочна (а следовательно, нуждается в пересмотре).

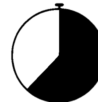
Наблюдения

Прежде всего нужно найти количественные характеристики для времени выполнения ваших программ. Измерить точное время выполнения программы непросто, но обычно вполне достаточно приближенной оценки. Существует целый ряд инструментов для получения таких приближенных оценок, самым простым из которых является, пожалуй, обычный секундомер или тип данных `Stopwatch` (см. листинг 3.2.2). Вы просто запускаете программу для разных вариантов входных данных и измеряете время на обработку каждого варианта ввода.

```
% java ThreeSum < 1Kints.txt
```




```
0
% java ThreeSum < 2Kints.txt
```




```
2
391930676 -763182495 371251819
-326747290 802431422 -475684132
```

Наблюдение за временем выполнения программы

Пожалуй, первое качественное наблюдение по поводу большинства программ состоит в том, что у задачи есть *размер*, от которого зависит ее вычислительная сложность. Обычно *размер задачи* определяется либо объемом входных данных, либо значением аргумента командной строки. Интуитивно понятно, что с увеличением размера задачи время ее выполнения должно возрастать, но вопрос в том, *насколько* оно возрастает, естественным образом возникает каждый раз, когда вы разрабатываете и запускаете программу.

Ко многим программам относится и другое качественное наблюдение: время выполнения относительно слабо зависит от содержимого входных данных; оно зависит прежде всего от размера задачи. Если это отношение не подтверждается, значит, вам нужно провести больше экспериментов, чтобы лучше понять зависимость времени выполнения от входных значений. Но поскольку эта связь часто не прослеживается, мы сосредоточимся на оценке количественного выражения зависимости времени выполнения от размера задачи.

Рассмотрим конкретный пример: начнем с программы `ThreeSum` (листинг 4.1.1), которая подсчитывает количество (неупорядоченных) троек в массиве из n чисел, сумма которых равна 0 (предполагается, что целочисленное переполнение не играет роли). На первый взгляд эта постановка задачи кажется искусственной, но она глубоко связана с фундаментальными задачами вычислительной геометрии, поэтому эта задача заслуживает тщательного изучения. Как размер задачи n связан с временем выполнения `ThreeSum`?

Листинг 4.1.1. Подсчет троек

```
public class ThreeSum
{
   public static void printTriples(int[] a)
   {  /* См. упражнение 4.1.1. */  }

   public static int countTriples(int[] a)
   {  // Подсчет троек с суммой 0.

      int n = a.length;
      int count = 0;
      for (int i = 0; i < n; i++)
         for (int j = i+1; j < n; j++)
            for (int k = j+1; k < n; k++)
               if (a[i] + a[j] + a[k] == 0)
                  count++;
      return count;
   }

   public static void main(String[] args)
   {
      int[] a = StdIn.readAllInts();
      int count = countTriples(a);
      StdOut.println(count);
      if (count < 10) printTriples(a);
   }
}
```

n	количество целых чисел
a[]	n целых чисел
count	количество троек с суммой 0

Метод countTriples() подсчитывает в a[] количество троек, сумма которых равна 0 (без учета целочисленного переполнения). Тестовый клиент вызывает countTriples() для целых чисел из стандартного ввода и выводит тройки, если количество чисел невелико. Файл 1Kints.txt содержит 1024 случайных значения типа int. Такой файл вполне может содержать нулевую тройку (см. упражнение 4.1.28).

```
% more 8ints.txt
 30
-30
-20
-10
 40
  0
 10
  5
```

```
% java ThreeSum < 8ints.txt
4
 30 -30   0
 30 -20 -10
-30 -10  40
-10   0  10

% java ThreeSum < 1Kints.txt
0
```

Гипотезы

На заре информатики Дональд Кнут[1] показал, что, несмотря на все факторы, усложняющие оценку времени выполнения программы, можно создать модель, которая помогает точно спрогнозировать время выполнения программы. Для анализа такого рода необходимы:

- хорошее понимание программы;
- хорошее понимание системы и компьютера;
- нетривиальные средства математического анализа.

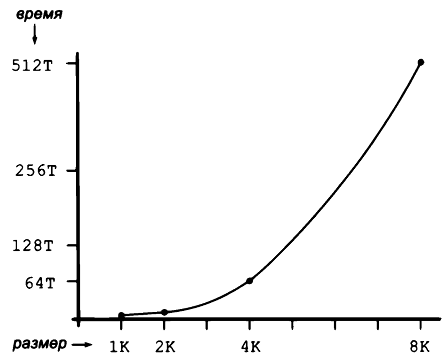
А это означает, что такой анализ лучше оставить для специалистов. Однако каждый программист должен уметь делать приближенную оценку быстродействия. К счастью, это умение не требует ничего, кроме эмпирических наблюдений и небольшого математического инструментария.

Гипотеза удвоения

Во многих программах можно быстро сформулировать гипотезу для следующего вопроса: *как удвоение размера входных данных отразится на времени выполнения*? Для простоты мы будем называть эту гипотезу *гипотезой удвоения*. Пожалуй, самый простой способ познакомиться с оценкой затрат — задавать себе этот вопрос о своих программах во время разработки. А теперь мы расскажем, как получить ответ на этот вопрос при помощи научного метода.

Эмпирический анализ

Очевидно, что для формулировки гипотезы удвоения проще всего начать с эксперимента: удвоить размер входных данных и посмотреть, как это повлияет на время выполнения. Например, программа `DoublingTest` (листинг 4.1.2) генерирует последовательность случайных входных массивов для `ThreeSum`, удваивая длину массива на каждом шаге, и выводит отношение времени выполнения `ThreeSum.countTriples()` для каждого варианта входных данных к времени для предыдущего (половинного) размера. Первые несколько строк выводятся очень быстро, но затем работа программы замедляется. Каждый раз, когда выводится очередная строка, вы задумываетесь над тем, сколько времени потребуется для решения вдвое большей задачи. Если



Стандартный график

[1] Еще один классический труд, с которым связан материал этой главы, — многотомник Дональда Кнута «Искусство программирования». — *Примеч. науч. ред.*

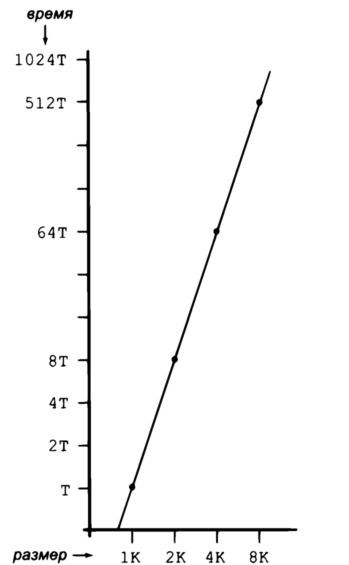
воспользоваться классом `Stopwatch` для проведения хронометража, вы увидите, что отношение сходится к значению, близкому к 8. Этот факт немедленно приводит нас к гипотезе о том, что при удвоении размера входных данных время выполнения увеличивается в 8 раз. Также можно построить график времени выполнения в стандартных координатах (этот график ясно показывает, что с увеличением размера входных данных увеличивается *скорость* роста) или в логарифмическом масштабе по обеим осям. В случае `ThreeSum` этот дважды логарифмический график представляет собой прямую линию с углом наклона 3. Этот факт очевидно указывает на то, что время выполнения удовлетворяет степенной зависимости в форме cn^3 (см. упражнение 4.1.6).

Математический анализ

Основная мысль Кнута по поводу построения математической модели для описания времени выполнения программы проста — общее время выполнения определяется двумя основными факторами:

- затратами на выполнение каждой операции;
- частотой[1] выполнения каждой операции.

Первый фактор является свойством системы, а второй — свойством алгоритма. Зная оба показателя для каждой операции в программе, можно просто перемножить их и просуммировать произведения для всех операций в программе. Результат определяет время выполнения.



Дважды логарифмический график

Основные трудности возникают с определением частоты выполнения операций. Некоторые из них анализируются легко: например, операция присваивания переменной `count` значения 0 в `ThreeSum.countTriples()` выполняется только один раз. Для других операций приходится рассуждать на более высоком уровне: например, операция ветвления `if` в `ThreeSum.countTriples()` выполняется ровно $n(n-1)(n-2)/6$ раз (количество вариантов выбора трех разных чисел из входного массива — см. упражнение 4.1.4).

Частотный анализ такого рода может привести к сложным и длинным математическим выражениям. Мы будем использовать более простые *приближенные* выражения, которые заметно упрощают эту задачу математического анализа в двух отношениях.

[1] Здесь и далее в этом контексте автор использует понятие «частота» (frequency), однако в большинстве случаев фактически подразумевается количество выполняемых операций. — *Примеч. науч. ред.*

Листинг 4.1.2. Проверка гипотезы удвоения

```
public class DoublingTest
{
   public static double timeTrial(int n)
   {  // Вычисление времени решения для случайных входных данных с размером n.
      int[] a = new int[n];
      for (int i = 0; i < n; i++)
         a[i] = StdRandom.uniform(2000000) - 1000000;
      Stopwatch timer = new Stopwatch();
      int count = ThreeSum.countTriples(a);
      return timer.elapsedTime();
   }

   public static void main(String[] args)
   {  // Вывод таблицы отношений при удваивании.
      for (int n = 512; true; n *= 2)
      {  // Вывод отношения для размера задачи n.
         double previous = timeTrial(n/2);
         double current  = timeTrial(n);
         double ratio = current / previous;
         StdOut.printf("%7d %4.2f\n", n, ratio);
      }
   }
}
```

`n`	размер задачи
`a[]`	случайные целые числа
`timer`	таймер

`n`	размер задачи
`previous`	время выполнения для n/2
`current`	время выполнения для n
`ratio`	отношение времен выполнения

Программа выводит в стандартный поток вывода таблицу отношений для задачи о сумме троек. Таблица показывает, как удвоение размера задачи влияет на время выполнения вызова метода ThreeSum.countTriples(); размер задачи изначально равен 512 и удваивается для каждой строки таблицы. Эксперимент приводит нас к гипотезе, что время выполнения увеличивается в 8 раз при удвоении размера входных данных. Запустите программу и убедитесь в том, что результаты подтверждают эту гипотезу.

```
% java DoublingTest
    512 6.48
   1024 8.30
   2048 7.75
   4096 8.00
   8192 8.05
   ...
```

Во-первых, мы будем учитывать только *доминирующий компонент* математического выражения, для чего будет использоваться математическая *запись с тильдой* (~). Запись $\sim f(n)$ представляет любую величину, которая при делении на $f(n)$ с увеличением n стремится к 1. Также будет использоваться запись $g(n) \sim f(n)$, которая означает, что с увеличением n $g(n)/f(n)$ стремится к 1. С такой записью можно игнорировать сложные части выражения, которые представляют малые значения. Например, если операция `if` в `ThreeSum` выполняется $\sim n^3/6$ раз, потому что $n(n-1)(n-2)/6 = n^3/6 - n^2/2 + n/3$, а это выражение при делении на $n^3/6$ стремится к 1 с ростом n. Эта запись полезна в том случае, если слагаемые после доминирующего

компонента относительно малозначимы (например, при $n = 1000$ это предположение равносильно утверждению о том, что подвыражение $-n^2/2 + n/3 \approx 499\ 667$ относительно малозначимо по сравнению с $n^3/6 \approx 166\ 666\ 667$, а это безусловно так).

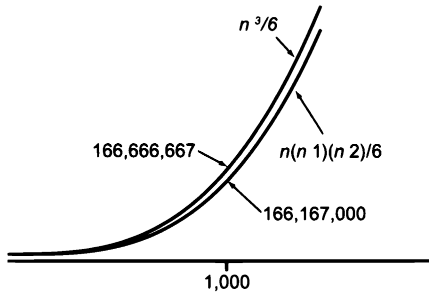
```
public class ThreeSum
{
   public static int count(int[] a)
   {
      int n = a.length;
      int count = 0;                                   <- 1
      for (int i = 0; i < n; i++)
         for (int j = i+1; j < n; j++)                 <- n
            for (int k = j+1; k < n; k++)              <- ~n²/2
               if (a[i] + a[j] + a[k] == 0)            <- ~n³/6
                  count++;
      return count;
   }

   public static void main(String[] args)
   {
      int[] a = StdIn.readAllInts();
      int count = count(a);
      StdOut.println(count);
   }
}
```

внутренний цикл (указывает на `if (a[i] + a[j] + a[k] == 0)`)

зависит от входных данных (указывает на `count++;`)

Частота выполнения отдельных команд программы

Во-вторых, мы сосредоточимся на том программном коде, который выполняется наиболее часто; иногда эту часть кода называются *внутренним циклом* программы. В этой программе разумно предположить, что время, потраченное на выполнение кода за пределами внутреннего цикла, относительно незначительно.



Аппроксимация по доминирующему компоненту

Ключевой момент анализа времени выполнения программы заключается в следующем: для очень многих программ время выполнения описывается отношением

$$T(n) \sim cf(n)$$

где c — константа, а $f(n)$ — функция, называемая *порядком роста* времени выполнения. Для типичных программ $f(n)$ представляет собой такую функцию, как $\log n$, n, $n \log n$, n^2 или n^3, как вы вскоре увидите (функции порядка роста принято приводить без постоянных коэффициентов). Если $f(n)$ является степенью n, как это часто бывает, такое утверждение эквивалентно тому, что время выполнения подчиняется степенной зависимости. В случае с `ThreeSum` это гипотеза, уже подтвержденная нашими эмпирическими наблюдениями: *порядок роста времени выполнения* `ThreeSum` составляет n^3. Значение константы c зависит как от затрат на выполнение операций в коде программы, так и от подробностей частотного анализа, но, как вы вскоре убедитесь, обычно вычислять это значение не обязательно.

Порядок роста — простая, но эффективная модель оценки времени выполнения. Например, зная порядок роста, обычно вы можете немедленно сформулировать гипотезу удвоения. В случае с `ThreeSum` порядок роста n^3 сообщает нам о том, что время выполнения при удвоении размера задачи будет возрастать в 8 раз, потому что

$$T(2n)/T(n) = c(2n)^3/(cn^3) = 8.$$

Оценка соответствует результатам, полученным в ходе эмпирического анализа, тем самым подтверждая как модель, так и эксперименты. Внимательно изучите этот пример, потому что этот метод поможет вам лучше понять быстродействие любой написанной вами программы.

Кнут показал, что для времени выполнения любой программы можно разработать точную математическую модель, и многие специалисты прилагают значительные усилия для разработки таких моделей. Но для понимания быстродействия программы подробная модель обычно не нужна: затраты на инструкции за пределами внутреннего цикла обычно можно спокойно проигнорировать (потому что они пренебрежимо малы по сравнению с затратами на инструкции внутреннего цикла), а знать значение константы в аппроксимации времени выполнения не обязательно (потому что при прогнозировании с использованием гипотезы удвоения оно сокращается).

количество инструкций	**время на инструкцию в секундах**	**частота**	**общее время**
6	2×10^{-9}	$n^3/6 - n^2/2 + n/3$	$(2n^3 - 6n^2 + 4n) \times 10^{-9}$
4	3×10^{-9}	$n^2/2 - n/2$	$(6n^2 + 6n) \times 10^{-9}$
4	3×10^{-9}	n	$(12n) \times 10^{-9}$
10	1×10^{-9}	1	10×10^{-9}
		итого:	$(2n^3 + 22n + 10) \times 10^{-9}$
		запись с тильдой	$\sim 2n^3 \times 10^{-9}$
		порядок роста	n^3

Анализ времени выполнения программы (пример)

В этой приближенной модели характеристики конкретной машины не играют особой роли — анализ отделяет алгоритм от системы. Порядок роста времени выполнения `ThreeSum` составляет n^3 независимо от того, будет ли эта программа написана на Java или Python, будет ли она выполняться на ноутбуке, мобильном телефоне или суперкомпьютере; в первую очередь все зависит от того факта, что программа анализирует все тройки. Свойства компьютера и системы обобщаются в различных предположениях о соотношениях строк кода программы и машинных инструкций, а также в фактическом времени выполнения, которое вы наблюдаете и закладываете в основу гипотезы удвоения. Порядок роста определяется используемым *алгоритмом*[1]. Такое разделение чрезвычайно эффективно, потому что оно позволяет накопить знания о быстродействии алгоритмов, а затем применить это знание к любому компьютеру. Собственно, большая часть знаний о быстродействии классических алгоритмов была получена десятилетия назад, но эти знания все еще остаются актуальными и для современных компьютеров.

Эмпирический и математический анализ наподобие описанного нами образует модель (объяснение происходящего), которая может быть формализована перечислением всех допущений (каждая операция всегда выполняется за одинаковое время, время выполнения имеет заданную форму и т. д.). Лишь немногие программы заслуживают подробной модели, но вы должны иметь представление о времени выполнения, которого можно ожидать от каждой написанной вами программы. *Учитывайте затраты.* Формулировка гипотезы удвоения — посредством эмпирического изучения, математического анализа или (желательно) с их совмещением — станет хорошей отправной точкой. Информация о быстродействии программы чрезвычайно полезна, и вскоре вы начнете формулировать и проверять такие гипотезы при каждом запуске программы. Кстати говоря, это хороший способ провести время, пока вы дожидаетесь завершения программы!

Классификация порядка роста

Для построения программ Java используется ограниченный набор структурных примитивов (обычные операции, условные переходы, циклы и вызовы методов), поэтому порядок роста программ очень часто определяется одной из немногих функций размера задачи, сводка которых приведена в следующей таблице. На эти функции нас выводит гипотеза удвоения, и это можно проверить, запустив программу. На самом деле, как будет показано в следующем описании, вы *уже* запускали программы, характеризуемые такими функциями порядка роста.

[1] Сказанное выше справедливо для «обычных», однопоточных алгоритмов. Если же алгоритм предполагает распараллеливание, то важную роль начинают играть и другие факторы. — *Примеч. науч. ред.*

порядок роста		
описание	**функция**	**множитель для гипотезы удвоения**
фиксированный	1	1
логарифмический	$\log n$	1
линейный	N	2
линейно-логарифмический	$n \log n$	2
квадратичный	n^2	4
кубический	n^3	8
экспоненциальный	2^n	2^n

Часто встречающиеся категории порядка роста

Фиксированный порядок роста

Программа, порядок роста времени выполнения которой является константой, всегда выполняет фиксированное количество операций для решения задачи; соответственно ее время выполнения не зависит от размера задачи. Наши первые программы из главы 1, такие как `HelloWorld` (листинг 1.1.1) и `LeapYear` (листинг 1.2.4), относятся к этой категории. Каждая из этих программ выполняет сразу несколько операций. Все операции Java с примитивными типами выполняются за фиксированное время, как и функции библиотеки `Math`. Обратите внимание: размер константы при этом не указывается. Например, константа для `Math.tan()` много больше, чем константа для `Math.abs()`.

Логарифмический порядок роста

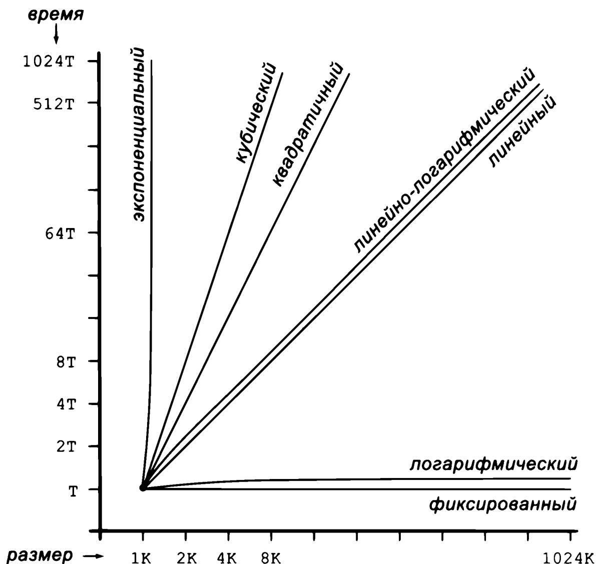
Программа с логарифмическим порядком роста времени выполнения работает намного медленнее программы с фиксированным временем. Классический пример программы с логарифмическим временем выполнения — поиск значения в отсортированном массиве — будет рассмотрен в следующем разделе (см. `BinarySearch` в листинге 4.2.3). Основание логарифма для порядка роста несущественно (так как все логарифмы с основаниями-константами связаны множителем-константой), поэтому для обозначения порядка роста используется запись $\log n$. Если нас интересует константа в доминирующем компоненте (как при использовании записи с тильдой), основание логарифма должно быть указано. Для двоичного логарифма (по основанию 2) используется запись $\lg n$, а для натурального логарифма (основание e) — запись $\ln n$.

Линейный порядок роста

Программы, требующие одинакового времени для обработки каждого блока входных данных или содержащие единственный цикл for, встречаются достаточно часто. Порядок роста времени выполнения такой программы называется линейным — время выполнения пропорционально размеру задачи. Классический пример — программа `Average` (листинг 1.5.3), вычисляющая среднее арифметическое чисел из стандартного ввода, а также код случайной перестановки элементов массива из раздела 1.4. Такие фильтры, как `PlotFilter` (листинг 1.5.5), также относятся к этой категории, как и различные фильтры обработки изображений из раздела 3.2, выполняющие одинаковое количество арифметических операций на пиксел.

Линейно-логарифмический порядок роста

К *«линейно-логарифмическим»* относятся программы, время выполнения которых для задачи с размером n имеет порядок роста $n \log n$. И снова основание логарифма роли не играет. Например, программа `CouponCollector` (листинг 1.4.2) является линейно-логарифмической. Классический пример такого рода — сортировка слиянием (см. листинг 4.2.6). Некоторые важные задачи имеют естественные решения с квадратичным временем, но для них найдены умные алгоритмы с линейно-логарифмическим временем. Такие алгоритмы (включая сортировку слиянием) чрезвычайно важны на практике, потому что они позволяют решать задачи с размером гораздо большим, чем это возможно с квадратичными решениями. В разделе 4.2 рассматривается общий принцип проектирования *«разделяй и властвуй»* для разработки линейно-логарифмических алгоритмов.



Порядки роста (двойной логарифмический масштаб)

Квадратичный порядок роста

Типичная программа, у которой время выполнения имеет порядок роста n^2, содержит два вложенных цикла `for`, перебирающих все пары множества из n элементов. Двойной вложенный цикл, вычисляющий попарные силы в программе `Universe` (листинг 3.4.2), — типичный пример программ этой категории, как и алгоритм сортировки вставкой (листинг 4.2.4), который будет рассмотрен в разделе 4.2.

описание	порядок роста	пример	конструкция
Фиксированный порядок	1	`count++;`	Элементарная операция (инкремент целого числа)
Логарифмический порядок	$\log n$	`for (int i = n; i > 0; i /= 2)` `    count++;`	Разбиение на две половины (биты в двоичном представлении)

Линейный порядок	n	`for (int i = 0; i < n; i++)`
		`    if (a[i] == 0)`
		`        count++;`
			Одиночный цикл (проверка каждого элемента)

Линейно-логарифмический порядок	$n \log n$	Сортировка слиянием (листинг 4.2.6)	«Разделяй и властвуй» (сортировка слиянием)

Квадратичный порядок	n^2	`for (int i = 0; i < n; i++)`
		`    for (int j = i+1; j < n; j++)`
		`        if (a[i] + a[j] == 0)`
		`            count++;`
			Двойной вложенный цикл (проверка всех пар)

Кубический порядок	n^3	`for (int i = 0; i < n; i++)`
		`    for (int j = i+1; j < n; j++)`
		`        for (int k = j+1; k < n; k++)`
		`            if (a[i] + a[j] + a[k] == 0)`
		`                count++;`
			Тройной вложенный цикл (проверка всех троек)

Экспоненциальный порядок	2^n	Код Грея (листинг 2.3.3)	Исчерпывающий поиск (проверка всех подмножеств)

Сводка типичных гипотез порядка роста

Кубический порядок роста

Пример `ThreeSum` из этого раздела является кубическим (его время выполнения имеет порядок роста n^3), потому что он использует три вложенных цикла `for` для обработки всех троек из n элементов. Время выполнения матричного умножения

в реализации из раздела 1.4 для умножения двух матриц $m \times m$ имеет порядок роста m^3, поэтому базовый алгоритм умножения матриц часто считается кубическим. Однако размер входных данных (количество элементов в матрицах) пропорционально $n = m^2$, поэтому алгоритм лучше охарактеризовать порядком $n^{3/2}$.

Экспоненциальный порядок роста

Как упоминалось в разделе 2.3, время выполнения программ `TowersOfHanoi` (листинг 2.3.2) и `Beckett` (листинг 2.3.3) пропорционально 2^n, потому что эти программы обрабатывают все подмножества n элементов. Обычно термином «экспоненциальный» обозначаются алгоритмы, у которых порядок роста составляет $2^{a \times n^b}$ для любых положительных констант a и b, хотя время выполнения сильно различается при разных значениях a и b.

Экспоненциальные алгоритмы чрезвычайно медленные — никогда не запускайте их для большой задачи. Они играют важную роль в теории алгоритмов, потому что существует большой класс задач, для которых алгоритм с экспоненциальным временем на сегодняшний день является наилучшим из известных.

Приведенная классификация является самой распространенной, но, безусловно, не полной. В самом деле, подробный анализ алгоритмов может потребовать полного спектра математических методов, которые разрабатывались веками. Чтобы понять время выполнения таких программ, как `Factors` (листинг 1.3.9), `PrimeSieve` (листинг 1.4.3) и `Euclid` (листинг 2.3.1), необходимо знать фундаментальные положения теории чисел. Классические алгоритмы, такие как `HashST` (листинг 4.4.3) и `BST` (листинг 4.4.4), требуют тщательного математического анализа. У таких программ, как `Sqrt` (листинг 1.3.6) и `Markov` (листинг 1.6.3), время выполнения зависит от скорости схождения вычислений к нужному числовому результату. Такие вычисления, как программа `Gambler` (листинг 1.3.8) и ее разновидности, представляют интерес именно потому, что точные математические модели не всегда доступны.

Тем не менее многие программы, которые вы напишете, будут обладать простыми характеристиками быстродействия, которые точно описываются одной из перечисленных категорий порядка роста. Соответственно мы можем работать с простыми высокоуровневыми гипотезами — например, что порядок роста времени выполнения алгоритма сортировки слиянием является линейно-логарифмическим. Для экономии места такие утверждения будут сокращаться: мы будем говорить, что *алгоритм сортировки является алгоритмом с линейно-логарифмическим временем*. Большинство наших гипотез по поводу затрат будет иметь либо такую форму, либо форму «*сортировка слиянием быстрее сортировки методом вставки*». И снова замечательная особенность таких гипотез заключается в том, что утверждения относятся к алгоритмам, а не к конкретным программам.

Конечно, вы всегда можете попытаться проанализировать время выполнения программы, просто запустив ее, однако при большом размере задачи этот способ может оказаться неприемлемым. Это все равно, что запустить ракету, чтобы узнать, где

она приземлится; определять разрушительную силу бомбы, взрывая ее; или строить мост, чтобы проверить его на прочность.

Зная порядок роста времени выполнения, мы можем принимать решения по большим задачам и выделять ресурсы для конкретных задач, которые действительно необходимо решать. Как правило, результаты гипотез относительно порядка роста времени выполнения программ используются одним из нескольких способов.

Оценка возможности решения больших задач

Чтобы учесть возможные затраты, вы должны ответить на следующий основной вопрос для каждой программы, которую вы напишете: сможет ли эта программа обработать свои входные данные за разумное время? Например, алгоритм с кубическим временем, который выполняется за пару секунд для задачи с размером n, будет выполняться несколько недель для задачи с размером $100n$, потому что он работает в миллион (100^3) раз медленнее, а пара миллионов секунд — это несколько недель. Если потребуется решить задачу с таким размером, вам придется поискать лучший метод. Зная порядок роста времени выполнения алгоритма, вы будете понимать ограничения на размер той задачи, которую вы решаете. Эта информация — самая важная причина для анализа быстродействия. Без нее вы вряд ли будете представлять, сколько времени займет выполнение программы; а располагая такой информацией, вы сможете на скорую руку прикинуть потенциальные затраты и выбрать подходящий курс действий.

порядок роста	прогнозируемое время выполнения при увеличении размера задачи в 100 раз
линейный	несколько минут
линейно-логарифмический	несколько минут
квадратичный	несколько часов
кубический	несколько недель
экспоненциальный	бесконечно

Последствия возрастания размера задачи для программы, выполняемой несколько секунд

Оценка последствий от использования более быстрого компьютера

Возможно, вы также столкнетесь с базовым вопросом: насколько быстрее будет решаться задача на более производительном компьютере? И снова порядок роста времени выполнения предоставляет именно ту информацию, которая вам нужна. Знаменитое эмпирическое правило, называемое *законом Мура*, гласит, что за 1,5 года быстродействие компьютеров и объем памяти примерно удваиваются, а за 5 лет скорость и объем память увеличиваются в 10 раз. Естественно предположить, что на компьютере, который в 10 раз быстрее старого, можно решать задачи в 10 раз большие, но для алгоритмов с квадратичным или кубическим временем

выполнения это *не так*. Регулярный запуск программ, работающих по несколько часов, — вполне рядовое явление как для инвестиционного брокера, ежедневно запускающего программы финансового моделирования, так и для ученого, запускающего программы для анализа экспериментальных данных, или инженера, запускающего программы моделирования для проверки проектировочных решений. Предположим, вы используете программу с кубическим временем выполнения, а затем покупаете новый компьютер, который работает в 10 раз быстрее и оснащен 10-кратным объемом памяти, — не потому, что вам нужен новый компьютер, а потому, что вы столкнулись с задачей, размер которой увеличился в 10 раз. И тут вас ждет разочарование: результатов придется дожидаться несколько недель, потому что большая задача выполняется на старом компьютере в 1000 раз медленнее, а на новом компьютере быстродействие возросло всего в 10 раз. Подобные ситуации — главная причина, по которой мы так ценим линейные и линейно-логарифмические алгоритмы: с таким алгоритмом на новом компьютере, который работает в 10 раз быстрее, задача с 10-кратным размером будет решаться за то же время. Другими словами, вы не сможете идти в ногу с законом Мура, если будете использовать алгоритм с квадратичным или кубическим временем.

порядок роста	коэффициент возрастания времени выполнения
линейный	1
линейно-логарифмический	1
квадратичный	10
кубический	100
экспоненциальный	бесконечно

Последствия использования компьютера с 10-кратным быстродействием при возрастании размера задачи в 10 раз

Сравнение программ

Мы всегда стремимся совершенствовать наши программы и часто расширяем или изменяем наши гипотезы для проверки эффективности различных усовершенствований. Имея возможность прогнозировать быстродействие, мы можем принимать в ходе разработки решения, которые ведут к более качественному, более эффективному коду. Например, неопытный программист может написать вложенные циклы `for` для программы `ThreeSum` (листинг 4.1.1) следующим образом:

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
         if (i < j && j < k)
            if (a[i] + a[j] + a[k] == 0)
               count++;
```

С таким кодом частота выполнения операций во внутреннем цикле составит ровно n^3 (вместо приблизительно $n^3/6$). Можно легко сформулировать и проверить

гипотезу, что эта версия работает в 6 раз медленнее `ThreeSum`. Обратите внимание: усовершенствования кода, не находящегося во внутреннем цикле, дают меньший эффект (или эффект вовсе отсутствует).

В более общей постановке задачи для двух алгоритмов, решающих одну задачу, нужно определить, какой из двух алгоритмов решает задачу с меньшими затратами вычислительных ресурсов. Во многих случаях можно определить порядок роста времени выполнения и выработать точные гипотезы относительно сравнительного быстродействия. Порядок роста в этом процессе исключительно важен, потому что он позволяет сравнить один конкретный алгоритм с целыми классами алгоритмов. Например, располагая линейно-логарифмическим алгоритмом для решения задачи, вы уже не интересуетесь алгоритмами с квадратичным или кубическим временем (даже если они сильно оптимизированы) для решения той же задачи.

Предостережения

Попытки подробного анализа быстродействия программ по многим причинам могут приводить к непоследовательным или ошибочным результатам. Все эти причины связаны с тем, что одно или несколько базовых предположений, лежащих в основе наших гипотез, могут оказаться неправильными. В таком случае можно разработать новые гипотезы на основании новых предположений, но чем больше подробностей вы учитываете, тем осторожнее необходимо действовать при анализе.

Время выполнения операций

Предположение о том, что каждая операция всегда выполняется за одинаковое время, не всегда правильно. Например, во многих современных компьютерных системах для работы с памятью применяется механизм *кэширования*, из-за которого обращение к элементам больших массивов, находящимся в несмежных позициях, может занимать гораздо больше времени[1]. Вы можете понаблюдать за эффектом кэширования для `ThreeSum`, запустив программу `DoublingTest` на некоторое время. После первоначального схождения к 8 показатель времени выполнения для больших массивов вдруг резко возрастает до большей величины; это объясняется эффектом кэширования.

Недоминирующий внутренний цикл

Предположение о доминировании внутреннего цикла не всегда оказывается правильным. Размер задачи n может быть недостаточно большим, а доминирующий компонент анализа не настолько превышает компоненты более низкого порядка. чтобы их можно было игнорировать. В некоторых программах большой объем кода находится за пределами внутреннего цикла, и этот код тоже необходимо принимать во внимание.

[1] Точнее, благодаря кэшированию существенно ускоряется доступ к *смежным* ячейкам памяти: они с большой вероятностью оказываются уже считанными в кэш, который намного быстрее основной оперативной памяти. — *Примеч. науч. ред.*

Особенности системы

Обычно на вашем компьютере происходит много, очень много всего. Java — лишь одно из приложений, конкурирующих за ресурсы, и у Java есть много параметров и средств управления, существенно влияющих на быстродействие. Эти соображения могут противоречить основополагающему принципу научного метода — воспроизводимости экспериментов, поскольку то, что происходит в конкретный момент на вашем компьютере, воспроизвести уже никогда не удастся. Влияние всего остального, что происходит в вашей системе (и вам неподконтрольно), должно быть сведено к пренебрежимо малому.

Трудности оценки результатов

Когда мы сравниваем две разные программы для одной задачи, одна программа может быть быстрее в одних ситуациях и медленнее в других. Различия в поведении могут зависеть от одного или нескольких из только что упоминавшихся факторов. У некоторых программистов (и студентов) существует естественная склонность тратить невероятно много сил на проведение сравнительного анализа с целью поиска «лучшей» реализации, но такую работу лучше оставить специалистам.

Сильная зависимость от входных значений

Одно из первых предположений, которые мы сделали для определения порядка роста времени выполнения программы, заключалось в том, что время выполнения должно зависеть прежде всего от размера задачи (и быть относительно независимым от входных значений). Если это предположение не выполняется, результаты могут быть непоследовательными или гипотезу не удастся проверить. В нашем примере `ThreeSum` такой проблемы нет, но во многих программах она, безусловно, присутствует. Несколько примеров такого рода еще встретятся вам в этой главе. Часто одной из основных целей проектирования становится устранение зависимости от входных значений. Если это невозможно, следует более тщательно моделировать данные, которые должны обрабатываться в решаемых задачах, а это может быть достаточно сложно. Например, если вы пишете программу для обработки генома, как узнать, как она поведет себя с другим геномом? Но хорошая модель, описывающая геномы, встречающиеся в природе, — именно то, что ищут ученые, поэтому оценка времени выполнения программ для данных, найденных в природе, на самом деле помогает в построении этой модели!

Множественные параметры задачи

Мы сосредоточились на оценке быстродействия как функции *одного* параметра (обычно значения аргумента командной строки или размера входных данных). Тем не менее таких параметров может быть несколько. Например, предположим, что `a[]` — массив длины `m`, а `b[]` — массив длины `n`. Возьмем следующий фрагмент кода для подсчета количества (неупорядоченных) пар `i` и `j`, для которых сумма `a[i] + b[j]` равна 0:

```
for (int i = 0; i < m; i++)
   for (int j = 0; j < n; j++)
      if (a[i] + b[j] == 0)
         count++;
```

Порядок роста времени выполнения зависит от двух параметров — `m` и `n`. В таких случаях параметры рассматриваются по отдельности: один фиксируется, пока вы анализируете другой. Например, порядок роста времени выполнения в приведенном фрагменте равен *mn*. Аналогичным образом в программе `LongestCommonSubsequence` (листинг 2.3.6) задействованы два критических параметра — *m* (длина первой строки) и *n* (длина второй строки), и порядок роста ее время выполнения составит *mn*.

Несмотря на все затруднения, понимание порядка роста времени выполнения каждой программы полезно для любого программиста, а только что описанные методы эффективны и универсальны. Кнут пришел к выводу, что, *в принципе*, эти методы можно довести до учета мельчайших подробностей, чтобы получить подробные, точные прогнозы. Типичные компьютерные системы чрезвычайно сложны, и подробный анализ лучше оставить специалистам, но те же методы позволяют получить приблизительную оценку времени выполнения любой программы. Ракетостроитель должен хотя бы примерно представлять, где завершится испытательный полет — в океане или в городе; врач-исследователь должен знать, вылечит или убьет новый лекарственный препарат пациентов; а любой ученый или инженер, запускающий программу, должен представлять, как долго она будет выполняться — секунду или год.

Гарантии быстродействия

Для некоторых программ требуется, чтобы время выполнения программы было меньше некоторого порогового значения для любого варианта ввода заданного размера. Чтобы получить такие гарантии, теоретики рассматривают задачу под крайне пессимистичным углом: каким будет время выполнения *в худшем случае*?

Например, такой консервативный подход может быть уместен для программы, которая управляет ядерным реактором, системой управления воздушным движением или тормозами машины. Необходимо гарантировать, что программа завершит свою работу в заданных границах, потому что в противном случае результат будет катастрофическим. Ученые обычно не рассматривают худший случай при изучении реального мира: в биологии худшим случаем может быть вымирание человеческой расы, в физике — конец Вселенной. Но в компьютерных системах худший случай может быть вполне реальным, если данные генерируются не природой, а другим пользователем (и, возможно, злоумышленником). Например, сайты, не использующие алгоритмы с гарантиями быстродействия, подвержены атакам отказа в обслуживании (DoS), когда хакеры заваливают их потоком вырожденных запросов, катастрофически снижающих быстродействие.

Проверить гарантии быстродействия научным методом трудно: вы не сможете экспериментально проверить гипотезу о том, что *сортировка слиянием гарантированно выполняется с линейно-логарифмическим временем*, не опробовав ее на всех возможных вариантах входных данных, а это сделать не удастся, потому что их слишком много. Чтобы опровергнуть гипотезу, достаточно привести семейство входных данных, для которых сортировка слиянием выполняется медленно, но как доказать ее истинность? Для этого следует применять не эксперименты, а математический анализ.

Задача специалиста по анализу алгоритмов — найти как можно больше актуальной информации об алгоритме, а задача прикладного программиста — применить эти знания для разработки программ, эффективно решающих имеющуюся задачу. Например, если вы применяете алгоритм с квадратичным временем для решения задачи, но можете найти для нее алгоритм с гарантированным линейно-логарифмическим временем, последнее решение обычно оказывается предпочтительным. В отдельных случаях алгоритм с квадратичным временем все равно оказывается предпочтительным, потому что он быстрее работает именно с теми данными, которые вам нужны, или потому что линейно-логарифмический алгоритм слишком сложен в реализации.

В идеале нам нужны алгоритмы, приводящие к ясному и компактному коду, которые обеспечивают нормальную гарантию для худшего случая и хорошее быстродействие для интересующих вас входных данных. Многие классические алгоритмы, рассмотренные в этой главе, играют важную роль в широком спектре приложений именно потому, что они обладают всеми этими свойствами. Взяв эти алгоритмы за образец, вы можете самостоятельно разработать хорошие решения для типичных задач, с которыми вы столкнетесь в ходе работы.

Память

Затраты памяти, как и время выполнения, напрямую связаны с реальным миром: немалая часть электроники вашего компьютера работает именно на обеспечение хранения данных и их последующей выборки. Чем больше данных должно храниться в любой конкретный момент времени, тем больше микросхем памяти вам понадобится. Чтобы учесть затраты памяти, необходимо знать, сколько памяти расходует алгоритм. Вероятно, вы понимаете ограничения на затраты памяти на своем компьютере хотя бы потому, что вы платили за расширение памяти на своем компьютере.

Затраты памяти для программ Java на вашем компьютере достаточно четко определены (каждое значение занимает одинаковый объем памяти при каждом запуске программы), но язык Java реализован на множестве разнообразных вычислительных устройств, а потребление памяти зависит от реализации. Для экономии места мы будем использовать термин «*типичный*» для машинно-зависимых значений. На типичной 64-разрядной машине компьютерная память разбита на слова, при этом каждое 64-разрядное слово состоит из 8 байтов, каждый байт состоит из 8 битов, а каждый бит представляет собой двоичную цифру.

Анализ затрат памяти несколько отличается от анализа времени выполнения прежде всего из-за того, что одной из самых значительных особенностей Java является система управления памятью, которая должна освободить программиста от рутинных операций с памятью. Конечно, вы должны пользоваться этим механизмом там, где это возможно. Тем не менее это не избавляет вас от обязанности понимать (хотя бы приближенно), когда требования программ к памяти не позволят вам решить некоторую задачу.

тип	байты
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

Типичные требования к памяти для примитивных типов

Примитивные типы

Оценить затраты памяти для простых программ вроде тех, которые рассматривались в главе 1, несложно: нужно подсчитать количество переменных и назначить им веса по количеству байтов, соответствующему типам. Например, так как тип данных Java `int` представляет множество целых чисел от –2 147 483 648 до 2 147 483 647, всего 2^{32} разных значения, типичные реализации Java используют 32 бита (4 байта) для представления каждого значения `int`. Аналогичным образом, в типичных реализациях Java каждое значение `char` представляется 2 байтами (16 битов), каждое значение `double` представляется 8 байтами (64 бита) и каждое значение `boolean` представляется 1 байтом (так как компьютеры обычно обращаются к памяти по байтам)[1]. Если на компьютере установлен 1 Гбайт памяти (около 1 миллиарда байтов), в любой момент времени в памяти не может храниться более 256 миллионов значений `int` или 128 миллионов значений `double`.

Объекты

Чтобы определить затраты памяти на хранение объекта, следует просуммировать объем памяти, используемой всеми переменными экземпляра, с *дополнительными затратами* для каждого объекта (обычно 16 байтов). При необходимости память обычно *выравнивается* (с округлением вверх) до значения, кратного 8 байтам (целому количеству машинных слов).

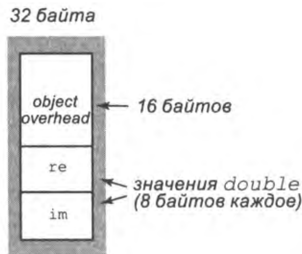
[1] В зависимости от реализации языка и от архитектуры компьютера реальный расход памяти может быть больше из-за выравнивания значений на границы машинных слов. — *Примеч. науч. ред.*

Например, в типичной системе объект `Complex` (листинг 3.2.6) использует 32 байта (16 байтов дополнительных затрат и 8 байтов для каждой из двух переменных экземпляров типа `double`). Так как многие программы создают миллионы объектов `Color`, типичные реализации Java упаковывают необходимую информацию в одно 32-разрядное значение `int`. Таким образом, объект `Color` использует 24 байта (16 байтов дополнительных затрат, 4 байта для переменной экземпляра `int`, 4 байта для выравнивания).

Ссылка на объект в типичной реализации использует 8 байтов (1 слово) памяти. Если класс включает ссылку на объект в переменной экземпляра, необходимо отдельно учитывать как память для ссылки на объект (8 байтов), так и память самого

Объект `Complex` *(PROGRAM 3.2.6)*

```
public class Complex
{
   private double re;
   private double im;
...
}
```

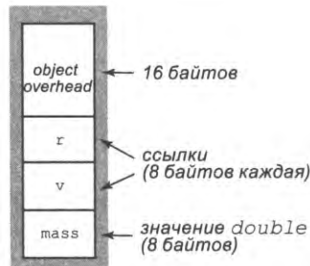


Объект `Color` *(Java library)*

```
public class Color
{
   private int value;
...
}
```

Объект `Body` *(листинг 3.4.1)* *40 байтов + два вектора*

```
public class Body
{
   private Vector r;
   private Vector v;
   private double mass;
...
}
```



Типичные затраты памяти для хранения объектов

объекта. Например, объект `Body` (листинг 3.4.1) использует 168 байтов: дополнительные затраты памяти для объекта (16 байтов), одно значение `double` (8 байтов) и две ссылки (8 байтов каждая), а также память, необходимую для хранения объекта `Vector`, — об этом далее.

Массивы

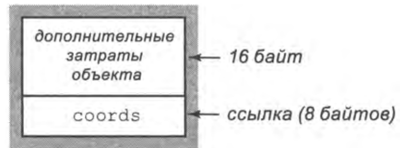
Массивы в языке Java реализуются в виде объектов, обычно содержащих переменную экземпляра `int` для хранения длины. Для примитивных типов для хранения массива из n элементов требуются 24 байта дополнительных затрат (16 байтов дополнительной памяти для объекта, 4 байта для длины и 4 байта выравнивания) плюс количество байтов, необходимое для хранения каждого элемента, умноженное на n. Например, массив `int` из `Sample` (листинг 1.4.1) использует $4n$+ 24 байта; массивы `boolean` из программы `Coupon` (листинг 1.4.2) используют n + 24 байта. Учтите, что массив `boolean` использует 1 байт памяти для каждого элемента (при этом 7 из 8 битов теряются) — если немного постараться, вы сможете справиться с делом, используя только 1 бит на элемент (см. упражнение 4.1.26).

Массив *объектов* представляет собой массив ссылок на объекты, поэтому учитывать нужно как память для ссылок, так и память для объектов. Например, массив n объектов `Charge` занимает $48n$ + 24 байта: дополнительные затраты массива (24 байта), ссылки на `Charge` ($8n$ байтов) и память для объекта `Charge` ($40n$ байтов). Анализ также предполагает, что все объекты различны, хотя несколько элементов массива могут ссылаться на один объект `Charge` (совмещение имен).

Объект `Vector` ***(листинг 3.3.3)***

```
public class Vector
{
   private double[] coords;
...
}
```

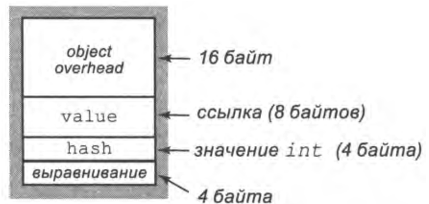
24 байта + массив `double` ***(8n + 24 байта)***



Объект `String` ***(библиотека Java)***

```
public class String
{
   private int hash;
   private char[] value;
...
}
```

40 байтов + массив `char` ***(2n + 24 байта)***



Типичные затраты памяти для объектов `Vector` ***и*** `String`

Класс `Vector` (листинг 3.3.3) включает массив в переменной экземпляра. В типичной системе объект `Vector` длины n занимает $8n$ + 48 байтов: дополнительные затраты объекта (16 байтов), ссылка на массив `double` (8 байтов) и память для массива `double` ($8n$ + 24 байта). Таким образом, каждый из объектов `Vector` в `Body` использует 64 байта памяти (так как n = 2).

Объекты String

Память для хранения объектов `String` рассчитывается так же, как и для любого другого объекта. Объект `String` длины n обычно занимает $2n$+56 байтов: дополнительные затраты объекта (16 байтов), ссылка на массив `char` (8 байтов), память для массива `char` ($2n$ + 24 байта), одно значение `int` (4 байта) и выравнивание (4 байта). В переменной экземпляра `int` в объектах `String` хранится *хеш-код* для экономии времени на повторных вычислениях в некоторых ситуациях, которые нас пока не интересуют. Если количество символов в строке не кратно 4, память символьного массива будет выравниваться, чтобы количество байтов массива `char` было кратно 8.

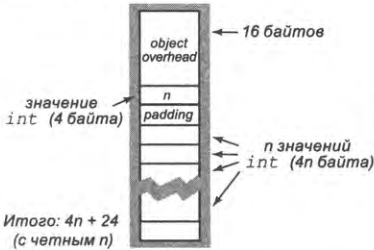
Двумерные массивы

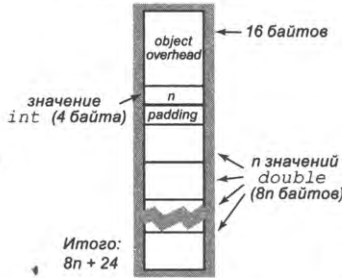
Как было показано в разделе 1.4, двумерный массив в Java представляет собой массив массивов. В результате двумерный массив в программе `Markov` (листинг 1.6.3) занимает $8n^2 + 32n + 24$, или ~ $8n^2$ байтов: дополнительные затраты для массива массивов (24 байта), n ссылок на массивы строк ($8n$ байтов) и n массивов строк ($8n$ + 24 байта каждый). Если элементы массива являются объектами, то аналогичные подсчеты для массива массивов, заполненного ссылками на объекты, дают ~ $8n^2$ байтов, к которым необходимо добавить память самих объектов.

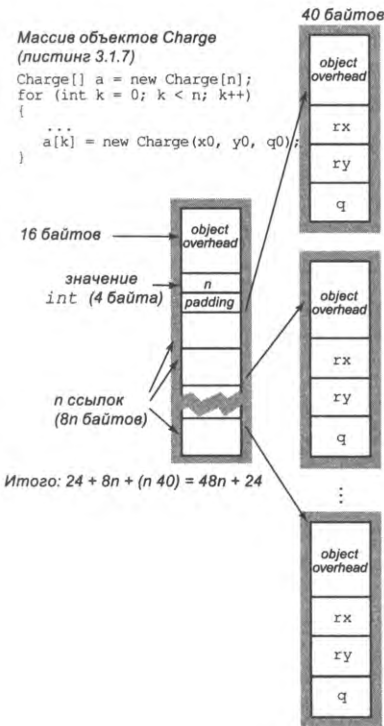
тип	байты
`boolean[]`	$n + 24 \sim n$
`int[]`	$4n + 24 \sim 4n$
`double[]`	$8n + 24 \sim 8n$
`Charge[]`	$40n + 24 \sim 40n$
`Vector`	$8n + 48 \sim 8n$
`String`	$2n + 56 \sim 2n$
`boolean[][]`	$n^2 + 32n + 24 \sim n^2$
`int[][]`	$4n^2 + 32n + 24 \sim 4n^2$
`double[][]`	$8n^2 + 32n + 24 \sim 8n^2$

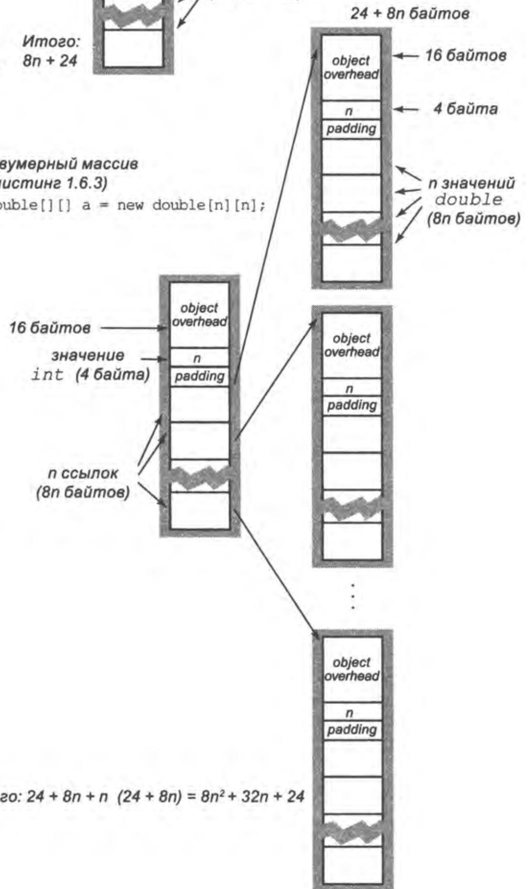
Типичные затраты памяти для типов данных переменной длины

Эти базовые приемы помогают оценить затраты памяти во многих программах, но существует много факторов, способных усложнить эту задачу. Мы уже упоминали о возможном эффекте совмещения имен. Кроме того, при вызове функций потребление памяти становится сложным динамическим процессом; механизм выделения памяти начинает играть более важную роль, и в нем проявляется больше системных зависимостей. Например, когда ваша программа вызывает метод, система выделяет память, необходимую для метода (его локальных переменных), из особой области памяти, называемой *стеком*; а когда метод возвращает управление на сторону вызова, эта память возвращается в стек. По этой причине создавать









Типичные затраты памяти для массивов, элементами которых являются int, double, объекты и массивы

массивы или другие большие объекты в рекурсивных программах опасно, поскольку каждый рекурсивный вызов приводит к значительным затратам памяти. Когда вы создаете объект ключевым словом new, система выделяет память, необходимую для хранения объекта, из другой специальной области памяти — *кучи*. Помните, что каждый объект продолжает существовать до того момента, пока не исчезнет

последняя ссылка на него. Тогда системный процесс, называемый *уборкой мусора*, может вернуть память в кучу. Эта динамика усложняет задачу точной оценки затрат памяти в программе.

Выводы

Хорошее быстродействие — важный фактор успеха программы. Недопустимо медленная программа почти так же бесполезна, как и неправильная. *Учитывайте затраты* с самого начала, и вы будете хотя бы примерно представлять, над какими задачами вообще стоит работать. В частности, всегда старайтесь понять, какой код будет входить во внутренний цикл ваших программ.

Пожалуй, самая распространенная ошибка в программировании — излишнее внимание к характеристикам быстродействия. Прежде всего постарайтесь сделать свой код понятным и правильным. Модификацию программы только для того, чтобы ускорить ее работу, лучше оставить специалистам. Дело в том, что в результате подобных модификаций часто появляется слишком запутанный и сложный для понимания код. Чарльз Хоар (изобретатель алгоритма быстрой сортировки и известный сторонник написания четкого и правильного кода) однажды кратко сформулировал эту идею: он сказал, что «преждевременная оптимизация — корень всего зла», к чему Кнут добавил уточнение: «(или по крайней мере его большей части) в программировании». Улучшение времени выполнения не стоит затраченных усилий, если выигрыш по затратам незначителен. Например, даже ускорение программы в 10 раз не столь существенно, если программа выполняется за долю секунды. И даже если на выполнение уходит несколько минут, общее время реализации и отладки улучшенного алгоритма может существенно превышать время, необходимое для запуска медленной версии, — с таким же успехом можно доверить работу компьютеру. Еще хуже, если вы потратите время и усилия на реализацию идей, которые должны были улучшить программу... но не улучшили.

Пожалуй, вторая распространенная ошибка при разработке алгоритмов — безразличие к характеристикам быстродействия. Более быстрые алгоритмы часто сложнее решений методом «грубой силы», и у вас может возникнуть искушение согласиться на более медленный алгоритм, чтобы не пришлось возиться с более сложным кодом. Тем не менее иногда всего несколько строк хорошего кода обеспечивают огромную экономию. Пользователи многих компьютерных систем тратят время, дожидаясь, пока алгоритм с квадратичным временем завершит работу, хотя немногим более сложный линейный или линейно-логарифмический алгоритм решит ту же задачу намного быстрее. А когда вы имеете дело с большим размером задачи[1], часто не остается другого выхода, кроме поиска лучшего алгоритма.

Каждый раз, когда вы работаете над кодом, подумайте над тем, как сделать его более понятным, эффективным и элегантным. *Учитывайте затраты* на протяже-

[1] Или с часто решаемой задачей. — *Примеч. науч. ред.*

нии всего цикла разработки, и ваши усилия будут приносить пользу при каждом выполнении этого кода.

Вопросы и ответы

В. Как определить, сколько времени занимает сложение или умножение двух чисел с плавающей точкой в моей системе?

О. Экспериментируйте! Программа `TimePrimitives` на сайте книги использует класс `Stopwatch` для измерения времени выполнения различных арифметических операций с примитивными типами. Это решение оценивает фактические затраты времени, как и при измерении времени по часам. Если количество одновременно выполняемых приложений в вашей системе невелико, результат может быть достаточно точным. За дополнительной информацией о том, как уточнить результаты таких экспериментов, обращайтесь на сайт книги.

В. Сколько времени требуется для вызова таких функций, как `Math.sin()`, `Math.log()` и `Math.sqrt()`?

О. Проведите эксперименты! С классом `Stopwatch` вы сможете легко писать программы, подобные `TimePrimitives`, чтобы получать ответы на такие вопросы самостоятельно. Если у вас это войдет в привычку, вы начнете использовать свой компьютер гораздо более эффективно.

В. Сколько времени занимают строковые операции?

О. Проведите эксперименты! (Вы уже уловили намек?) Тип данных `String` реализован так, чтобы методы `length()` и `charAt()` выполнялись за фиксированное время. Такие методы, как `toLowerCase()` и `replace()`, выполняются за время, линейное по отношению к длине строки. Методы `compareTo()`, `equals()`, `startsWith()` и `endsWith()` выполняются за время, пропорциональное количеству символов, необходимых для получения ответа (фиксированное время в лучшем случае, линейное в худшем), но операция `indexOf()` может быть медленной. Конкатенация строк и метод `substring()` выполняются за время, пропорциональное общему количеству символов в результате.

В. Почему выделение памяти для массива длины n требует времени, пропорционального n?

О. В Java элементы массива автоматически инициализируются значениями по умолчанию (0, `false` или `null`). Теоретически эта операция может выполняться с фиксированным временем, если система отложит инициализацию каждого элемента до первого непосредственного обращения к элементу, но в большинстве реализаций Java выполняется перебор всего массива с инициализацией всех элементов.

В. Как определить, сколько памяти доступно для моих программ Java?

О. При нехватке памяти Java сообщит вам об этом, поэтому вы можете получить ответ, проведя эксперименты. Например, если запустить программу `PrimeSieve` (листинг 1.4.3) командой

```
% java PrimeSieve 100000000
```

вы получите результат

```
50847534
```

Если затем выполнить команду

```
% java PrimeSieve 1000000000
```

результат будет выглядеть так:

```
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
```

Следовательно, в памяти хватает места для размещения массива `boolean` со 100 миллионами элементов, но не для массива `boolean` с 1 миллиардом элементов. Объем доступной для Java памяти можно увеличить параметрами командной строки. Следующая команда выполняет `PrimeSieve` с аргументом командной строки `1000000000` и ключом `-Xmx1110m`, который запрашивает для Java до 1100 мегабайт памяти (если она доступна):

```
% java -Xmx1100mb PrimeSieve 1000000000
```

В. Мне попадалась запись вида «время выполнения равно $O(n^2)$». Что это значит?

О. Это пример так называемой *записи «О-большое»*. Мы говорим, что время выполнения $f(n)$ равно $O(g(n))$, если существуют такие константы c и n_0, для которых $|f(n)| \le c|g(n)|$ для всех $n > n_0$. Иначе говоря, функция $f(n)$ ограничена сверху $g(n)$ до постоянного множителя и для достаточно больших значений n. Например, время выполнения $30n^2 + 10n + 7$ равно $O(n^2)$. Мы говорим, что *худшее время выполнения алгоритма* равно $O(g(n))$, если время выполнения как функция размера входных данных n равно $O(g(n))$ для всех возможных вариантов ввода. Запись «О-большое» и понятие худшего времени выполнения широко используются учеными для доказательства теорем, относящихся к алгоритмам, поэтому вы наверняка столкнетесь с этой записью в учебном курсе алгоритмов и структур данных.

В. Могу ли я как-то использовать тот факт, что худшее время выполнения алгоритма равно $O(n^3)$ или $O(n^2)$, для прогнозирования быстродействия?

О. Не всегда, потому что фактическое время выполнения может быть гораздо ниже. Например, функция $30n^2 + 10n + 7$ имеет время выполнения $O(n^2)$, но ее с таким же успехом можно охарактеризовать временем $O(n^3)$ и $O(n^{10})$, потому что запись «О-большое» предоставляет только верхнюю границу. Более того, даже если существует некоторое семейство входных данных, для которого время выполнения пропорционально заданной функции, может оказаться, что такие данные не встре-

чаются на практике. Следовательно, запись «О-большое» не должна использоваться для прогнозирования быстродействия. Запись с тильдой и классификация порядка роста точнее записи «О-большого», потому что они предоставляют как верхнюю, так и нижнюю границу для скорости роста функции. Многие программисты ошибочно используют запись «О-большое» для обозначения обеих границ, верхней и нижней.

Упражнения

4.1.1. Реализуйте статический метод `printTriples()` для программы `ThreeSum` (листинг 4.1.1). Метод выводит в стандартный поток вывода все тройки с нулевой суммой.

4.1.2. Измените программу `ThreeSum`, чтобы она получала целое число как аргумент командной строки и находила в стандартном вводе три числа, сумма которых ближе всего к заданному числу.

4.1.3. Напишите программу `FourSum`, которая читает целые числа типа `long` из стандартного потока ввода и считает количество четверок с нулевой суммой. Используйте четверной вложенный цикл. Каков порядок роста времени выполнения вашей программы? Оцените самый большой объем данных, с которыми ваша программа справится за час. Запустите программу и проверьте свою гипотезу.

4.1.4. Докажите методом индукции, что количество (неупорядоченных) пар целых чисел в диапазоне от 0 до $n - 1$ равно $n(n - 1)/2$. Затем докажите по индукции, что количество (неупорядоченных) троек целых чисел от 0 до $n - 1$ равно $n(n - 1)(n - 2)/6$.

Ответ для пар: формула верна для $n = 1$, так как количество пар равно 0. Для $n > 1$ подсчитаем все пары, не включающие $n - 1$; их количество равно $(n - 1)(n - 2)/2$ по индукционной гипотезе. К ним следует прибавить все пары, включающие $n - 1$; их количество равно $n - 1$. В результате получаем

$$(n - 1)(n - 2)/2 + (n - 1) = n(n - 1)/2.$$

Ответ для троек: формула верна для $n = 2$. Для $n > 2$ подсчитаем все тройки, не включающие $n - 1$; их количество равно $(n - 1)(n - 2)(n - 3)/6$ по индукционной гипотезе. К ним прибавляются все тройки, включающие $n - 1$; их количество равно $(n - 1)(n - 2)/2$. В результате получаем

$$(n - 1)(n - 2)(n - 3)/6 + (n - 1)(n - 2)/2 = n(n - 1)(n - 2)/6.$$

4.1.5. Покажите, используя аппроксимацию с интегралами, что количество разных троек целых чисел от 0 до n равно приблизительно $n^3/6$.

Ответ:

$$\sum_0^n \sum_0^i \sum_0^j 1 \approx \int_0^n \int_0^i \int_0^j dk\, dj\, di = \int_0^n \int_0^i j\, dj\, di = \int_0^n (2i^2/2) di = n^3/6.$$

4.1.6. Покажите, что дважды логарифмический график функции cn^b имеет наклон b и пересекает ось x в точке $\log c$. Каким углом наклона и точкой пересечения характеризуется график $4n^3(\log n)^2$?

4.1.7. Как выражается значение переменной `count` как функции n после выполнения следующего фрагмента?

```
long count = 0;
for (int i = 0; i < n; i++)
   for (int j = i + 1; j < n; j++)
      for (int k = j + 1; k < n; k++)
         count++;
```

Ответ: $n(n - 1)(n - 2)/6$.

4.1.8. Используйте запись с тильдой для упрощения следующих формул. В каждом случае приведите порядок роста.

a. $n(n - 1)(n - 2)(n - 3)/24$

b. $(n - 2) (\lg n - 2) (\lg n + 2)$

c. $n(n + 1) - n^2$

d. $n(n + 1)/2 + n \lg n$

e. $\ln ((n - 1)(n - 2) (n - 3))^2$

4.1.9. Определите порядок роста времени выполнения следующей команды `ThreeSum` как функции количества целых чисел n в стандартном вводе:

```
int[] a = StdIn.readAllInts();
```

Ответ: линейный. «Узкие места» этого кода — неявная инициализация массива и неявный цикл ввода. Впрочем, в зависимости от вашей системы затраты на подобный цикл ввода могут доминировать в программе с линейно-логарифмическим или даже квадратичным временем (при недостаточно большом размере входных данных).

4.1.10. Определите, за какое время (как функция n) выполняется следующий код: линейное, квадратичное или кубическое.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      if (i == j) c[i][j] = 1.0;
      else        c[i][j] = 0.0;
```

4.1.11. Предположим, время выполнения алгоритма для ввода с размером 1000, 2000, 3000 и 4000 составляет 5 секунд, 20 секунд, 45 секунд и 80 секунд соответственно. Оцените, сколько времени займет решение задачи с размером 5000. Является ли алгоритм линейным, линейно-логарифмическим, квадратичным, кубическим или экспоненциальным?

4.1.12. Алгоритм с каким порядком роста времени выполнения вы выберете: квадратичным, линейным или линейно-логарифмическим?

Ответ: быстрый выбор на основании одного лишь порядка роста выглядит соблазнительно, но при этом очень легко ошибиться. Вы должны хотя бы примерно представлять размер задачи и относительные значения постоянных коэффициентов времен выполнения. Допустим, имеется набор из следующих вариантов времени выполнения: n^2 секунд, $100\ n \log 2\ n$ секунд и $10\ 000n$ секунд. Квадратичный алгоритм будет самым быстрым для значений n примерно до 1000, а линейный алгоритм никогда не будет быстрее линейно-логарифмического (значение n должно быть больше 2^{100} — настолько большим, что его можно не принимать во внимание).

4.1.13. Примените научный подход для разработки и проверки гипотезы о порядке роста времени выполнения следующего фрагмента кода (как функции аргумента n):

```
public static int f(int n)
{
   if (n == 0) return 1;
   return f(n-1) + f(n-1);
}
```

4.1.14. Примените научный подход для разработки и проверки гипотезы о порядке роста времени выполнения метода `collect()` программы `Coupon` (листинг 2.1.3) как функции аргумента *n*. *Примечание*: удвоение неэффективно для выбора между линейной и линейно-логарифмической гипотезой — попробуйте возвести в квадрат размер входных данных.

4.1.15. Примените научный подход для разработки и проверки гипотезы о порядке роста времени выполнения программы `Markov` (листинг 1.6.3) как функции аргументов командной строки `trials` и `n`.

4.1.16. Примените научный подход для разработки и проверки гипотезы о порядке роста времени выполнения каждого из двух следующих фрагментов кода как функции `n`.

```
String s = "";
for (int i = 0; i < n; i++)
   if (StdRandom.bernoulli(0.5)) s += "0";
   else                          s += "1";

StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++)
   if (StdRandom.bernoulli(0.5)) sb.append("0");
   else                          sb.append("1");
String s = sb.toString();
```

4.1.17. Каждая из четырех функций Java, приведенных ниже, возвращает строку длины `n`, содержащую только символы `x`. Определите порядок роста времени выполнения каждой функции. Вспомните, что конкатенация двух строк в Java выполняется за время, пропорциональное длине полученной строки.

```
public static String method1(int n)
{
   if (n == 0) return "";
   String temp = method1(n / 2);
   if (n % 2 == 0) return temp + temp;
   else            return temp + temp + "x";
}

public static String method2(int n)
{
   String s = "";
   for (int i = 0; i < n; i++)
      s = s + "x";
   return s;
}

public static String method3(int n)
{
   if (n == 0) return "";
   if (n == 1) return "x";
   return method3(n/2) + method3(n - n/2);
}

public static String method4(int n)
{
   char[] temp = new char[n];
   for (int i = 0; i < n; i++)
       temp[i] = 'x';
   return new String(temp);
}
```

4.1.18. Следующий фрагмент кода (взятый в измененном виде из книги по программированию на Java) создает случайную перестановку целых чисел в диапазоне от 0 до $n - 1$. Определите порядок роста времени выполнения как функции n. Сравните ее порядок роста с кодом случайной перестановки из раздела 1.4.

```
int[] a = new int[n];
boolean[] taken = new boolean[n];
int count = 0;
while (count < n)
{
   int r = StdRandom.uniform(n);
   if (!taken[r])
   {
       a[r] = count;
       taken[r] = true;
       count++;
   }
}
```

4.1.19. Каким порядком роста времени выполнения характеризуются следующие две функции? Каждая функция получает в качестве аргумента строку и возвращает новую строку, символы которой переставлены в обратном порядке.

```
public static String reverse1(String s)
{
   int n = s.length();
   String reverse = "";
   for (int i = 0; i < n; i++)
      reverse = s.charAt(i) + reverse;
   return reverse;
}

public static String reverse2(String s)
{
   int n = s.length();
   if (n <= 1) return s;
   String left = s.substring(0, n/2);
   String right = s.substring(n/2, n);
   return reverse2(right) + reverse2(left);
}
```

4.1.20. Приведите алгоритм перестановки символов строки в обратном порядке с линейным временем.

Ответ:

```
public static String reverse(String s)
{
   int n = s.length();
   char[] a = new char[n];
   for (int i = 0; i < n; i++)
      a[i] = s.charAt(n-i-1);
   return new String(a);
}
```

4.1.21. Напишите программу `MooresLaw`, которая получает аргумент командной строки n и выводит прогноз роста быстродействия процессоров за десятилетие, если она удваивается каждые n месяцев. Насколько возрастет скорость процессоров за следующее десятилетие, если она удваивается каждые $n = 15$ месяцев? 24 месяца?

4.1.22. Используя 64-разрядную модель памяти, описанную в тексте, приведите затраты памяти для объекта каждого из следующих типов данных из главы 3:

- `Stopwatch`
- `Turtle`
- `Vector`
- `Body`
- `Universe`

4.1.23. Приведите оценку затрат памяти (как функции размера сетки n), используемой программой `PercolationVisualizer` (листинг 2.4.3) при проверке вертикального протекания (листинг 2.4.2). *Дополнительное задание*: приведите

ответ для случая с использованием рекурсивного метода обнаружения протекания (листинг 2.4.5).

4.1.24. Оцените размер наибольшего двумерного массива с элементами `int`, который может храниться в памяти вашего компьютера. Попробуйте выделить память для этого массива.

4.1.25. Оцените объем памяти, используемой программой `CompareDocuments` (листинг 3.3.5), как функцию количества документов n и длины d.

4.1.26. Напишите версию программы `PrimeSieve` (листинг 1.4.3), которая использует массив с элементами `byte` вместо массива `boolean`. Эта версия должна использовать все биты в каждом байте, вследствие чего наибольшее поддерживаемое значение n увеличивается в 8 раз.

4.1.27. В следующей таблице приведены данные времени выполнения для трех программ с различными значениями n. Заполните пропуски оценками, которые кажутся вам разумными на основании приведенной информации.

программа	1000	10 000	100 000	1 000 000
A	0,001 секунды	0,012 секунды	0,16 секунды	? секунд
B	1 минута	10 минут	1,7 часа	? часов
C	1 секунда	1,7 минуты	2,8 часа	? дней

Приведите гипотезы для порядка роста времени выполнения каждой программы.

Упражнения повышенной сложности

4.1.28. *Анализ суммы троек.* Вычислите вероятность того, что среди n случайных 32-разрядных целых чисел нет ни одной тройки с нулевой суммой. *Дополнительное задание*: приведите приближенную формулу для ожидаемого числа таких троек (как функции n) и проведите эксперименты для подтверждения вашей оценки.

4.1.29. *Ближайшие пары значений.* Спроектируйте алгоритм с квадратичным временем, который находит в массиве целых чисел пару с наиболее близкими значениями. (В следующем разделе вам будет предложено найти линейно-логарифмический алгоритм для этой задачи.)

4.1.30. *Атака DoS.* Популярный веб-сервер поддерживает функцию `no2slash()`, предназначенную для устранения множественных символов `/`. Например, строка `/d1///d2////d3/test.html` сворачивается в `/d1/d2/d3/test.html`. Исходный алгоритм был основан на многократном поиске `/` и копировании оставшейся части строки:

```
int n = name.length();
int i = 1;
```

```
while (i < n)
{
   if ((c[i-1] == '/') && (c[i] == '/'))
   {
      for(int j = i+1; j < n; j++)
         c[j-1] = c[j];
       n--;
   }
   else i++;
}
```

К сожалению, этот код выполняется за квадратичное время (например, если строка содержит символ /, повторенный n раз). Одновременно отправляя множество запросов с большим количеством символов /, хакер может парализовать работу сервера и лишить другие процессы доступа к процессору, создавая таким образом DoS-атаку. Разработайте версию `no2slash()`, которая выполняется за линейное время и предотвращает атаки такого типа.

4.1.31. *Сумма подмножеств.* Напишите программу `SubsetSum`, которая читает целые значения типа `long` из стандартного ввода и подсчитывает количество подмножеств этого набора чисел, сумма которых равна 0. Укажите порядок роста времени выполнения вашей программы.

4.1.32. *Таблицы Юнга.* Предположим, имеется целочисленный массив $n \times n$ `a[][]`, в котором для всех `i` и `j` `a[i][j] < a[i+1][j]`, и `a[i][j] < a[i][j+1]`, как в следующем массиве 5×5.

```
 5  23  54  67  89
 6  69  73  74  90
10  71  83  84  91
60  73  84  86  92
99  91  92  93  94
```

Двумерный массив, обладающий таким свойством, называется *таблицей Юнга*. Напишите функцию, которая получает в аргументах таблицу Юнга $n \times n$ и целое число и определяет, входит ли целое число в таблицу Юнга. Порядок роста времени выполнения вашей функции в зависимости от n должен быть линейным.

4.1.33. *Циклический сдвиг массива.* Для заданного массива из n элементов приведите алгоритм с линейным временем для циклического сдвига элементов на k позиций. Иначе говоря, если массив содержит элементы $a_0, a_1, ..., a_{n-1}$, то после циклического сдвига он принимает вид $a_k, a_{k+1}, ..., a_{n-1}, a_0, ..., a_{k-1}$. Дополнительные затраты памяти не должны превышать постоянной величины. *Подсказка*: выполните обратную перестановку трех подмассивов.

4.1.34. *Поиск повторяющегося числа.* А. Для заданного массива из n целых чисел от 1 до n, в котором одно значение повторяется дважды и одно пропущено, приведите алгоритм поиска отсутствующего значения за линейное время с постоянными затратами дополнительной памяти. Целочисленное переполнение не допускается. В. Для заданного массива из n целых чисел от 1 до $n - 1$, доступного только для

чтения, в котором одно значение повторяется дважды и одно пропущено, приведите алгоритм поиска повторяющегося значения за линейное время с постоянными затратами дополнительной памяти. С. Для заданного массива из n целых чисел от 1 до $n - 1$, доступного только для чтения, приведите алгоритм поиска повторяющегося значения за линейное время с постоянными затратами.

4.1.35. *Факториал.* Разработайте быстрый алгоритм для вычисления $n!$ для больших значений n с использованием класса Java `BigInteger`. Используйте свою программу для вычисления самой длинной непрерывной серии из цифр 9 в 1000000!. Разработайте и проверьте гипотезу относительно порядка роста времени выполнения этого алгоритма.

4.1.36. *Максимальная сумма.* Разработайте алгоритм с линейным временем, находящий в массиве из n значений типа `long` смежный подмассив с длиной не более m, обладающий наибольшей суммой среди всех таких подмассивов. Реализуйте свой алгоритм и убедитесь в том, что порядок роста времени его выполнения линеен.

4.1.37. *Максимальное среднее.* Напишите программу для поиска в массиве из n значений типа `long` смежного подмассива с длиной не более m с наибольшим средним значением элементов, посредством перебора всех подмассивов. При помощи научного подхода докажите, что порядок роста времени выполнения вашей программы равен mn^2. Затем напишите программу для решения этой задачи, которая сначала вычисляет величину `prefix[i] = a[0]+...+a[i]` для каждого `i`, а затем среднее значение в интервале от `a[i]` до `a[j]` выражением `(prefix[j] - prefix[i]) / (j - i + 1)`. Используйте научный подход для подтверждения того, что это решение сокращает порядок роста в n раз.

4.1.38. *Поиск по образцу.* Для подмассива из $n \times n$ черных (1) и белых (0) пикселов разработайте алгоритм с линейным временем, который находит самый большой квадратный подмассив, не содержащий белых пикселов. В следующем примере самый большой такой подмассив 3 × 3 выделен жирным шрифтом.

```
1 0 1 1 1 0 0 0
0 0 0 1 0 1 0 0
0 0 1 1 1 0 0 0
0 0 1 1 1 0 1 0
0 0 1 1 1 1 1 1
0 1 0 1 1 1 1 0
0 1 0 1 1 0 1 0
0 0 0 1 1 1 1 0
```

Реализуйте свой алгоритм и убедитесь в том, что порядок роста времени его выполнения в зависимости от количества пикселов линеен. *Дополнительное задание*: разработайте алгоритм для поиска самого большого прямоугольного подмассива, содержащего только черные пикселы.

4.1.39. *Субэкспоненциальная функция.* Найдите функцию с порядком роста времени выполнения большим, чем у любой полиномиальной функции, но меньшим, чем

у любой экспоненциальной функции. *Дополнительное задание*: найдите программу с таким порядком роста времени выполнения.

4.2. Сортировка и поиск

Задача сортировки заключается в перестановке элементов массива по возрастанию. Эта задача, хорошо знакомая многим разработчикам, играет важную роль во многих приложениях: песни в вашей фонотеке упорядочены по возрастанию, сообщения электронной почты выводятся в порядке, обратном порядку их получения, и т. д. Хранение данных в некотором порядке естественно. Упорядочение полезно тем, что найти нужное значение в отсортированном массиве намного проще, чем в неотсортированном. Этот аспект особенно важен в приложениях с очень большими массивами, в которых эффективная организация поиска становится важным фактором качества решения.

Задачи сортировки и поиска важны в коммерческих приложениях (данные клиентов хранятся по порядку) и в научных приложениях (организация данных и вычислений); они находят самые разнообразные применения в областях, которые на первый взгляд вообще не связаны с упорядочением чего-либо: сжатии данных, компьютерной графике, вычислительной биологии, обработке числовых данных, комбинаторной оптимизации, криптографии и многих других.

Эти фундаментальные задачи наглядно показывают, что *эффективные алгоритмы* становятся ключевым аспектом эффективного решения вычислительных задач. Теоретики разработали много разных методов сортировки и поиска. Какие из них следует применить для решения конкретной задачи? Этот вопрос важен, потому что разные алгоритмы могут существенно различаться по своим характеристикам быстродействия — настолько, что в одном случае вы получаете успешное решение для реальной ситуации, а во втором ничего не выйдет даже на самом быстром компьютере.

В этом разделе подробно рассматриваются два классических алгоритма сортировки и поиска — бинарная сортировка и сортировка слиянием, а также примеры их практического применения, для которых эффективность особенно критична. Примеры убедят вас не только в практической полезности этих методов, но и в необходимости *учитывать затраты* при решении любых задач, связанных с большим объемом вычислений.

Бинарный поиск

Игра «Двадцать вопросов» (листинг 1.5.2) станет важным и полезным уроком по проектированию эффективных алгоритмов. Правила игры просты: игрок должен отгадать загаданное число из n целых чисел от 0 до $n - 1$. После каждого пред-

положения игрок сообщает, как названное число связано с загаданным: больше, меньше или равно. По причинам, которые будут ясны позднее, мы слегка изменим правила, чтобы вопросы задавались в форме: «Загаданное число больше либо равно x?» с ответами `true` или `false`. Временно будем считать, что n является степенью 2.

Как было показано в разделе 1.5, эффективная стратегия решения задачи заключается в последовательном сужении интервала, содержащего загаданное число. На каждом шаге задается вопрос, который позволяет уменьшить размер интервала вдвое. А именно, называется число из середины интервала, и в зависимости от ответа отбрасывается половина интервала, которая не может содержать загаданное число. Точнее говоря, используется *полуоткрытый* интервал, содержащий левую границу, но не правую. Мы будем использовать запись [*lo*,*hi*) для обозначения всех целых чисел, больших или равных *lo*, и меньших (но не равных) *hi*. Начнем с $lo = 0$ и $hi = n$ и будем применять следующую рекурсивную стратегию.

интервал	*длина*	*вопрос*	*ответ*
0 128	128	≥ 64 ?	true
0 64 128	64	≥ 96 ?	false
0 64 96	32	≥ 80 ?	false
0 64 80	16	≥ 72 ?	true
0 72 80	8	≥ 76 ?	true
0 76 80	4	≥ 78 ?	false
0 76 78	2	≥ 77 ?	true
0 77	1	= 77	

Отгадывание числа методом бинарного поиска

- Базовый случай: если результат $hi - lo$ равен 1, загадано число *lo*.
- Шаг свертки: в противном случае спросить: «Загаданное число больше или равно $mid = lo + (hi - lo)/2$?» Если ответ положителен, перейти к поиску числа в интервале [*mid*,*hi*); если нет, перейти к поиску в интервале [*lo*, *mid*).

Функция `binarySearch()` в программе `Questions` (листинг 4.2.1) реализует эту стратегию. Это пример общего метода решения задач, называемого *бинарным поиском*; этот метод находит много практических применений (другие названия — *метод деления пополам*, *дихотомический* поиск, а также *логарифмический*).

Листинг 4.2.1. Бинарный поиск (20 вопросов)

```
public class Questions
{
   public static int binarySearch(int lo, int hi)
   {  // Поиск числа в интервале [lo, hi)
      if (hi - lo == 1) return lo;
      int mid = lo + (hi - lo) / 2;
      StdOut.print("Greater than or equal to " + mid + "?  ");
      if (StdIn.readBoolean())
         return binarySearch(mid, hi);
      else
         return binarySearch(lo, mid);
   }

   public static void main(String[] args)
   {  // Игра "Двадцать вопросов".
      int k = Integer.parseInt(args[0]);
      int n = (int) Math.pow(2, k);
      StdOut.print("Think of a number ");
      StdOut.println("between 0 and " + (n-1));
      int guess = binarySearch(0, n);
      StdOut.println("Your number is " + guess);
   }
}
```

lo	наименьшее возможное значение
hi - 1	наибольшее возможное значение
mid	середина
k	количество вопросов
n	количество возможных значений

Программа использует бинарный поиск для игры из листинга 1.5.2, но на этот раз роли меняются: вы выбираете число, а программа пытается его угадать. Она получает целое число k в аргументе командной строки и предлагает загадать число из диапазона от 0 до $n - 1$, где $n = 2^k$. Программа гарантированно угадывает ответ за k вопросов.

```
% java Questions 7
Think of a number between 0 and 127
Greater than or equal to 64?  false
Greater than or equal to 96?  true
Greater than or equal to 80?  true
Greater than or equal to 72?  false
Greater than or equal to 76?  false
Greater than or equal to 78?  true
Greater than or equal to 77?  false
Your number is 77
```

Доказательство правильности

Сначала необходимо убедиться в том, что алгоритм работает правильно, то есть всегда приводит к загаданному числу. Для этого необходимо установить следующие факты.

- Интервал всегда содержит загаданное число.
- Размеры интервала представляют собой степени 2, уменьшающиеся начиная с n.

Первый из этих фактов следует непосредственно из кода. Чтобы убедиться в истинности второго, достаточно заметить, что если $(hi - lo)$ является степенью 2, то $(hi - lo)/2$ — следующая меньшая степень 2, а также размер обоих половинных интервалов $[lo, mid)$ и $[mid, hi)$. На основании этих фактов по индукции следует, что алгоритм работает так, как положено. Со временем размер интервала уменьшается до 1, поэтому число гарантированно будет найдено.

Анализ времени выполнения

Пусть n — число возможных значений. В программе 4.2.1 $n = 2^k$, где $k = \lg n$. Пусть $T(n)$ — количество вопросов. Из рекурсивной стратегии следует, что значение $T(n)$ должно удовлетворять следующему рекуррентному отношению:

$$T(n) = T(n/2) + 1$$

при $T(1) = 0$. Заменяя n на 2^k, мы можем получить выражение в закрытой форме:

$$T(2^k) = T(2^{k-1}) + 1 = T(2^{k-2}) + 2 = \ldots = T(1) + k = k.$$

Снова подставляя n вместо 2^k (и $\lg n$ вместо k), получаем результат:

$$T(n) = \lg n.$$

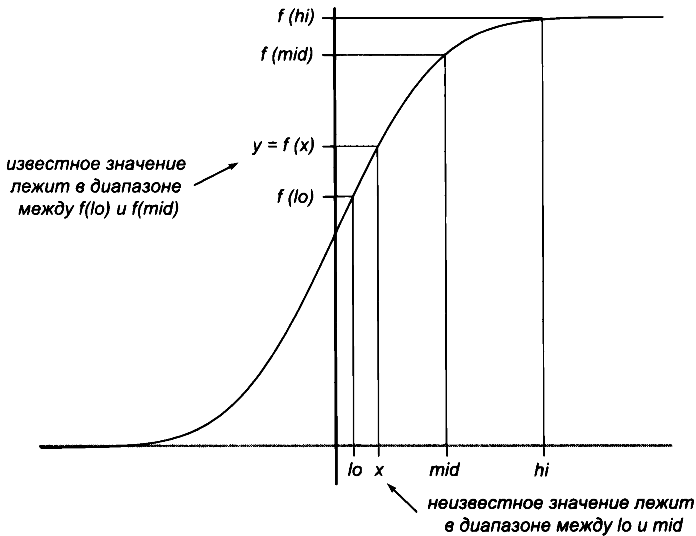
Это подтверждает гипотезу о том, что время выполнения бинарного поиска является логарифмическим. *Примечание*: бинарный поиск и программа `TwentyQuestions.binarySearch()` работают даже в том случае, если n не является степенью 2, — это предположение было сделано для упрощения доказательства (см. упражнение 4.2.1).

Пропасть между линейным и логарифмическим временем

Вместо бинарного поиска также можно действовать иначе: назвать 0, потом 1, 2, 3 и т. д., пока игрок не доберется до загаданного числа. Этот алгоритм, называемый *последовательным поиском*, является примером алгоритма «грубой силы»: он добивается своей цели, не обращая внимания на затраты. Время выполнения последовательного поиска зависит от загаданного числа: если загадано число 0, то последовательный поиск отрабатывает всего за 1 шаг, но для загаданного числа $n - 1$ потребуется n шагов. Если загаданное число выбирается случайно, то ожидаемое количество шагов составит $n/2$. При этом бинарный поиск гарантированно использует не более $\lg n$ шагов. Как вы вскоре убедитесь, различия между n и $\lg n$ в реальных приложениях просто огромны. Понимание масштаба этих различий — важный шаг на пути к пониманию важности проектирования и анализа алгоритмов. В текущем контексте представьте, что обработка предположения занимает 1 секунду. С бинарным поиском значение любого загаданного числа из миллиона займет менее 20 секунд; при последовательном поиске методом грубой силы может потребоваться до 1 миллиона секунд, то есть более 1 недели. Мы рассмотрим примеры, в которых от различий во времени выполнения зависит практическая возможность или невозможность решения задачи.

Двоичное представление

Если вернуться к листингу 1.3.7, вы сразу же поймете, что бинарный поиск по сути эквивалентен преобразованию числа в двоичную запись! Каждое предположение определяет один бит результата. В нашем примере информация о том, что число находится в интервале от 0 до 127, означает, что двоичное представление состоит из 7 битов, а ответ на первый вопрос (число больше либо равно 64?) сообщает значение старшего бита; ответ на второй вопрос определяет значение следующего бита и т. д. Например, если загадано число 77, последовательность ответов «нет да да нет нет да нет» дает 1001101 — двоичное представление 77. Двоичное представление — еще одна точка зрения, помогающая понять разрыв между линейным и логарифмическим временем: у программы с временем выполнения, линейным относительно n, время выполнения пропорционально n, а у логарифмической программы время выполнения пропорционально *количеству цифр* в n. Подумайте над следующим вопросом, который демонстрирует тот же вопрос в более привычном контексте: какую бы зарплату вы предпочли: $6 или шестизначную?



Бинарный поиск (половинное деление)

Обратная функции

В качестве примера применения бинарного поиска в научных вычислениях мы рассмотрим задачу *обращения* возрастающей функции $f(x)$, то есть нахождения функции, обратной $f(x)$. Для заданного значения y требуется найти значение x, для которого $f(x) = y$. В такой ситуации границы интервала определяются вещественными числами, но, по сути, используется тот же алгоритм, что и при отгадывании числа: на каждом шаге интервал сокращается вдвое так, чтобы значение x

оставалось внутри него, пока интервал не уменьшится настолько, что мы будем знать значение x с нужной точностью δ. Мы начинаем с интервала (lo, hi), заведомо содержащего x, и используем следующую рекурсивную стратегию.

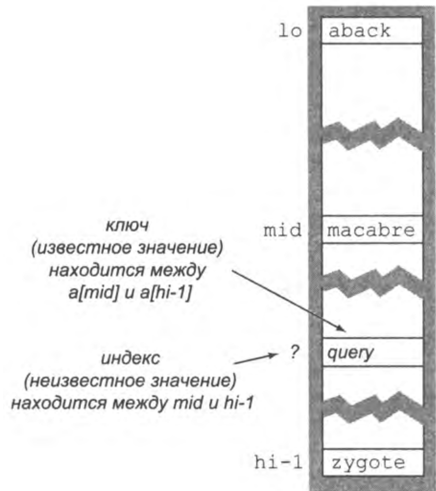
- Вычисление $mid = lo + (hi - lo)/2$.
- Базовый случай: если $hi - lo$ меньше δ, то mid возвращается как оценка x.
- Свертка: в противном случае проверить условие $f(mid) > y$. Если оно истинно, поиск x продолжается в (lo, mid), а если нет — в (mid, hi).

Чтобы вы лучше разобрались в происходящем, листинг 4.2.2 вычисляет обратную функцию для гауссовой функции распределения Φ, которая рассматривалась в программе `Gaussian` (листинг 2.1.2).

Для этого метода принципиально то, что функция является возрастающей: для значений a и b из того, что $f(a) < f(b)$, следует, что $a < b$, и наоборот. Рекурсивный шаг просто применяет эту информацию: из $y = f(x) < f(mid)$ следует, что $x < mid$, а значит, значение x должно находиться в интервале (lo, mid), а из $y = f(x) > f(mid)$ следует, что $x > mid$, следовательно, значение x должно находиться в интервале (mid, hi). Считайте, что алгоритм определяет, какой из $n = (hi - lo)/\delta$ крошечных интервалов размера δ внутри (lo, hi) содержит x, с временем выполнения, логарифмическим по n. Как и в случае с числовыми преобразованиями для целых чисел, каждая итерация определяет один бит x. В этом контексте бинарный поиск часто называется поиском методом половинного деления, потому что на каждой стадии интервал делится надвое.

Бинарный поиск в отсортированном массиве

Одним из важнейших практических применений бинарного поиска является поиск информации по ключу. Это применение настолько распространено в современной информатике, что вытеснило аналогичные «бумажные» технологии. Например, в прошлом веке люди использовали *телефонные книги* для поиска телефонного номера. В обоих случаях базовый механизм остается один и тот же: элементы следуют в порядке сортировки по ключу, который используется для их идентификации (слово в словаре, имя абонента в телефонной книге, в обоих случаях используется сортировка по алфавиту). Даже если вы привыкли искать такую информацию в компьютере, подумайте, как бы вы искали слово в словаре. Последовательный поиск начнет



Бинарный поиск в отсортированном массиве (один шаг)

Листинг 4.2.2. Обращение функции методом половинного деления

```
public static double inverseCDF(double y)
{  return bisectionSearch(y, 0.00000001, -8, 8);  }
private static double bisectionSearch(double y, double delta,
                                      double lo, double hi)
{  // Вычисление x с cdf(x) = y.
   double mid = lo + (hi - lo)/2;
   if (hi - lo < delta) return mid;
   if (cdf(mid) > y)
      return bisectionSearch(y, delta, lo, mid);
   else
      return bisectionSearch(y, delta, mid, hi);
}
```

y	аргумент
delta	нужная точность
lo	наименьшее возможное значение
mid	середина
hi	наибольшее возможное значение

Эта реализация inverseCDF() для нашей библиотеки Gaussian (листинг 2.1.2) использует метод половинного деления для вычисления точки x, для которой значение F(X) равно заданному значению y в пределах заданной погрешности delta. Эта рекурсивная функция делит пополам интервал x, содержащий заданную точку, вычисляет функцию в середине интервала и использует тот факт, что функция F возрастает, чтобы определить, находится ли нужная точка в левой или правой половине. Деление продолжается до тех пор, пока размер интервала не станет меньше заданной точности.

сначала и будет просматривать каждый элемент до тех пор, пока не найдет нужное слово. Никто не использует этот алгоритм: вместо этого вы открываете книгу на некоторой внутренней странице и ищете слово на этой странице. Если слово найдено, работа закончена; если нет — из рассмотрения исключается часть книги до текущей страницы или после нее, и весь процесс повторяется. Вероятно, вы уже узнали в этом методе тот же бинарный поиск (листинг 4.2.3).

Фильтр исключений

В разделе 4.3 будут рассмотрены подробности реализации компьютерных программ, которые могли бы использоваться вместо словаря или телефонной книги. Программа из листинга 4.2.3 использует бинарный поиск для простой проверки присутствия заданного ключа в отсортированном массиве ключей. Например, при проверке орфографии необходимо знать, встречается слово в словаре или нет, а определение вас не интересует. При компьютерном поиске информация хранится в массиве, отсортированная в порядке ключей (в одних приложениях информация поступает уже отсортированной, в других ее нужно сначала отсортировать с использованием одного из алгоритмов, описанных далее в этом разделе).

Листинг 4.2.3. Бинарный поиск (отсортированный массив)

```
public class BinarySearch
{
   public static int search(String key, String[] a)
   {  return search(key, a, 0, a.length);  }

   public static int search(String key, String[] a, int lo, int hi)
   {  // Поиск ключа в a[lo, hi).
      if (hi <= lo) return -1;
      int mid = lo + (hi - lo) / 2;
      int cmp = a[mid].compareTo(key);
      if      (cmp > 0) return search(key, a, lo, mid);
      else if (cmp < 0) return search(key, a, mid+1, hi);
      else              return mid;
   }

   public static void main(String[] args)
   {  // Вывод ключей из стандартного ввода,
      // не встречающихся в файле args[0].
      In in = new In(args[0]);
      String[] a = in.readAllStrings();
      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (search(key, a) < 0) StdOut.println(key);
      }
   }
}
```

key	ключ поиска
a[lo, hi)	отсортированный подмассив
lo	наименьший индекс
mid	средний индекс
hi	наибольший индекс

Метод search() использует бинарный поиск для возвращения индекса строкового ключа в отсортированном массиве (или –1, если ключ отсутствует в массиве). Тестовый клиент представляет собой фильтр исключений, который читает (отсортированный) «белый список» из файла, заданного аргументом командной строки, и выводит слова из стандартного ввода, не входящие в «белый список».

```
% more emails.txt
bob@office
carl@beach
marvin@spam
bob@office
bob@office
mallory@spam
dave@boat
eve@airport
alice@home
```

```
% more whitelist.txt
alice@home
bob@office
carl@beach
dave@boat

% java BinarySearch whitelist.txt < emails.txt
marvin@spam
mallory@spam
eve@airport
```

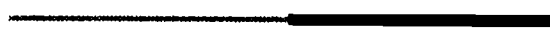
двадцать вопросов (преобразование в двоичную форму)

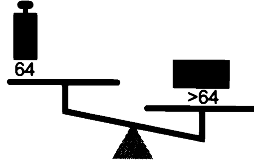
взвешивание предмета

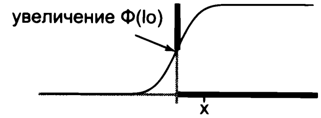
обратная функция

1??????

больше 64

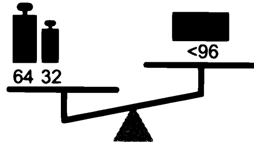


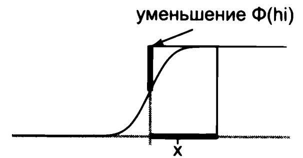




меньше 64 + 32

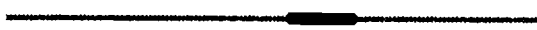
10?????

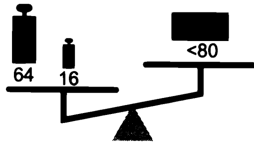


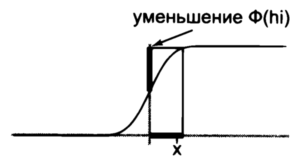


меньше 64 + 16

100????

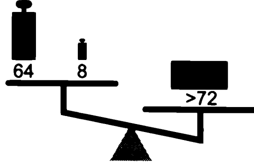


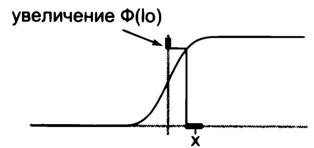




1001???

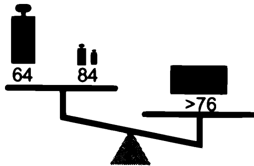
больше 64 + 8

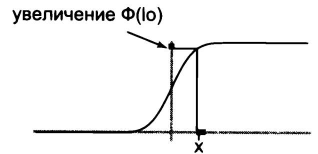




10011??

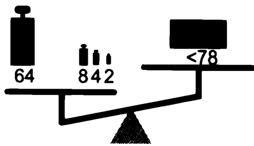
больше 64 + 8 + 4

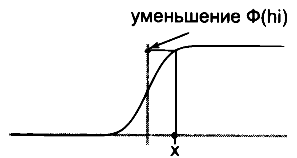




меньше 64 + 8 + 4 + 2

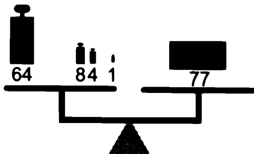
100110?

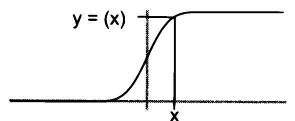




равно 64 + 8 + 4 + 2 + 1

1001101





Три применения бинарного поиска

Бинарный поиск в программе 4.2.3 отличается от других примеров в двух отношениях. Во-первых, длина массива n не обязана быть степенью 2. Во-вторых, программа должна предусмотреть возможность того, что искомый ключ в массиве отсутствует. Чтобы запрограммировать бинарный поиск с учетом этих аспектов, вы должны действовать осторожно (см. разделы «Вопросы и ответы» и «Упражнения»).

Тестовый клиент из листинга 4.2.3 называется *исключающим фильтром*: он читает из файла отсортированный список строк (который называется «белым списком») и произвольную последовательность строк из стандартного ввода и выводит те элементы последовательности, которые не входят в «белый список». Исключающие фильтры находят много практических применений. Например, если «белый список» содержит слова из словаря, а в стандартном вводе содержится текстовый документ, исключающий фильтр выводит неправильно записанные слова. Другой пример встречается в веб-приложениях: приложение электронной почты может использовать исключающий фильтр для отклонения сообщений, не входящих в «белый список» с адресами ваших друзей. Или операционная система может содержать такой фильтр, который запрещает сетевые подключения с вашего компьютера к любым устройствам с IP-адресами, не входящими в утвержденный «белый список».

Взвешивание

Бинарный поиск известен с древности — возможно, отчасти из-за следующей задачи. Предположим, вы хотите определить вес предмета, используя рычажные весы и набор гирь. С бинарным поиском можно воспользоваться гирями, веса которых представляют степени 2 (достаточно иметь только одну гирю каждого типа). Положите предмет на правую чашку весов и ставьте гири по убыванию веса на левую чашку. Если весы склоняются налево, гиря снимается, а если нет — остается на весах. Этот процесс полностью аналогичен вычислению двоичного представления числа посредством вычитания убывающих степеней 2, как в листинге 1.3.7.

Быстрые алгоритмы — важнейший элемент современного программирования, а бинарный поиск — классический пример, демонстрирующий эффект от применения быстрых алгоритмов. Проведя быстрые вычисления, можно убедиться в том, что для поиска всех ошибочных слов в документе или защиты компьютера от хакеров с использованием фильтра исключений потребуется быстрый алгоритм (такой, как быстрый поиск). Не жалейте времени на реализацию быстрых алгоритмов. Вы сможете моментально найти исключения в документе из миллиона элементов по словарю с миллионным «белым списком», тогда как с алгоритмом «грубой силы» на эту работу могут уйти дни и недели. В наши дни веб-компании предоставляют сервисы, основанные на использовании бинарного поиска в отсортированных списках с миллиардами элементов, — без быстрых алгоритмов (таких, как быстрый поиск) подобные сервисы были бы невозможны.

Современные ученые окружены данными со всех сторон, будь то обширные данные экспериментов или подробные представления некоторого аспекта реального мира.

Бинарный поиск и быстрые алгоритмы стали неотъемлемыми компонентами научного прогресса. Применение алгоритма «грубой силы» полностью аналогично поиску слова в словаре, когда пользователь начинает с первой страницы и переворачивает страницы одну за другой. С быстрым алгоритмом можно моментально провести поиск среди миллиардов информационных элементов. Время, проведенное за выявлением и использованием быстрого алгоритма поиска, наверняка окупится возможностью быстро решить задачу без значительных затрат ресурсов.

Сортировка методом вставки

Бинарный поиск требует, чтобы данные были заранее отсортированы, а сортировка имеет также и много других практических применений, поэтому мы обратимся к алгоритмам сортировки. Начнем с решения методом «грубой силы», а затем обратимся к более сложному алгоритму, который подходит для очень больших объемов данных.

Алгоритм «грубой силы», известный под названием *сортировки методом вставки*, основан на простом принципе, который иногда используется для упорядочения игральных карт при раздаче. Игрок последовательно перебирает карты и вставляет каждую в подходящее место среди уже рассмотренных карт (с сохранением порядка сортировки). Следующий код моделирует этот процесс в методе Java, который упорядочивает элементы строкового массива по возрастанию:

```
public static void sort(String[] a)
{
   int n = a.length;
   for (int i = 1; i < n; i++)
      for (int j = i; j > 0; j--)
         if (a[j-1].compareTo(a[j]) > 0)
              exchange(a, j-1, j);
         else break;
}
```

В начале каждой итерации внешнего цикла `for` первые `i` элементов массива располагаются в порядке сортировки; внутренний цикл `for` перемещает `a[i]` в правильную позицию массива, как в следующем примере с `i` = 6:

i	j	a[]							
		0	1	2	3	4	5	6	7
6	6	**and**	**had**	**him**	**his**	**was**	**you**	**the**	but
6	5	and	had	him	his	was	**the**	**you**	but
6	4	and	had	him	his	**the**	**was**	you	but
		and	had	him	**his**	**the**	was	you	but

Вставка a[6] в позицию сортировки (значение меняется местами с большим значением, находящимся слева от него)

То есть элемент a[i] помещается на свое место среди отсортированных элементов, для чего он меняется местами (с использованием метода exchange(), который встречался в разделе 2.1) с каждым из больших значений слева от него, пока не достигнет нужной позиции. В трех нижних строках таблицы элементы, которые сравниваются с a[i], выводятся жирным шрифтом.

Только что описанный процесс вставки выполняется сначала для i, равного 1, затем 2, затем 3 и т. д., как показано в следующей таблице.

i	j	a[]							
		0	1	2	3	4	5	6	7
		was	**had**	**him**	**and**	**you**	**has**	**the**	**but**
1	0	**had**	**was**	him	and	you	has	the	but
2	1	had	him	**was**	and	you	has	the	but
3	0	**and**	**had**	**him**	**was**	you	has	the	but
4	4	and	had	him	was	you	his	the	but
5	3	and	had	him	his	**was**	**you**	the	but
6	4	and	had	him	his	the	**was**	**you**	but
7	1	and	**but**	**had**	**him**	**his**	**the**	**was**	**you**
		and	**but**	**had**	**him**	**his**	**the**	**was**	**you**

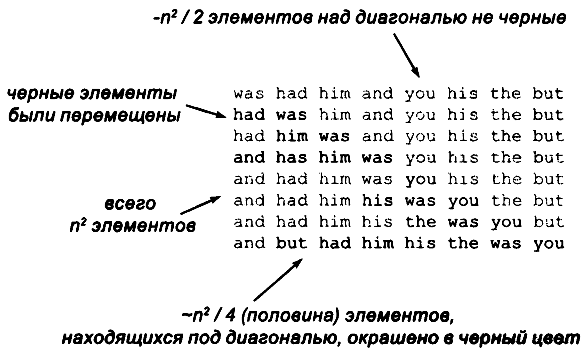
Вставка элементов с a[1] по a[n–1] в позицию (сортировка методом вставки)

В строке i этой таблицы приводится содержимое массива при завершении внешнего цикла for вместе со значением j на этот момент. Серым шрифтом выделена строка, которая находилась в a[i] в начале цикла, а жирным черным выделены другие строки, которые участвовали в замене и сместились вправо на одну позицию в цикле. Так как элементы с a[0] по a[i-1] хранятся в порядке сортировки при завершении цикла для каждого значения i, они также будут храниться в порядке сортировки и после завершения последней итерации, когда значение i равно a.length. Это обсуждение лишний раз подчеркивает то, с чего нужно начинать изучение или разработку нового алгоритма: вы должны убедиться в его правильности. При этом вы получите достаточно информации об алгоритме, чтобы анализировать его быстродействие и эффективно использовать его.

Анализ времени выполнения

Внутренний цикл сортировки методом вставки представляет собой цикл for, в который вложен второй цикл for, что наводит на мысль о квадратичном времени выполнения, но наличие операции break не позволяет сразу принять это предположение. Например, в лучшем случае, когда входной массив уже упорядочен нужным образом, вложенный цикл for ограничивается одиночным сравнением (чтобы узнать, что a[j-1] меньше или равно a[j] для каждого j от 1 до n - 1)

и командой `break`, так что общее время выполнения линейно. Напротив, если входной массив отсортирован в обратном порядке, этот цикл отрабатывает без единого прерывания. Таким образом, частота выполнения операций внутреннего цикла равна $1 + 2 + ... + n - 1 \sim ½\, n^2$, а время выполнения квадратично. Чтобы понять эффективность сортировки методом вставки для входных массивов со *случайным* порядком, внимательно присмотритесь к таблице: перед вами массив $n \times n$, в котором каждый черный элемент соответствует одной замене. Другими словами, количество таких элементов соответствует частоте выполнения операций во внутреннем цикле. Можно предположить, что каждый новый элемент с равной вероятностью попадет в любую позицию, поэтому в среднем элемент сдвинется влево на половину интервала. А следовательно, в среднем можно ожидать, что только половина элементов под диагональю (всего приблизительно $n^2/4$) будет черной. Это напрямую приводит к гипотезе, что ожидаемое время выполнения сортировки методом вставки для массива со случайным порядком будет квадратичным.



Анализ сортировки методом вставки

Сортировка данных других типов

В программах часто требуется сортировать данные других типов, не только строки. Например, в приложении для научных расчетов экспериментальные результаты могут сортироваться по числовым значениям; в коммерческом приложении для сортировки могут использоваться денежные суммы, время или дата; в системной программе данные могут сортироваться по IP-адресам или идентификаторам процессов. Концепция сортировки во всех этих ситуациях интуитивно понятна, но реализация метода сортировки, который будет работать во всех этих случаях, является классическим примером использования механизма функциональной абстракции вроде того, что предоставляют интерфейсы Java. Для сортировки объектов в массиве необходимо иметь возможность *сравнивать* два элемента и получать результат: первый объект больше, меньше или равен второму. Java предоставляет для этой цели интерфейс `java.util.Comparable`.

public interface Comparable<Key>	
int compareTo(Key b)	сравнивает объект с b и возвращает информацию об их упорядочении

API интерфейса Java java.util.Comparable

Класс, реализующий интерфейс `Comparable`, обязуется реализовать метод `compareTo()` для объектов своего типа, чтобы вызов `a.compareTo(b)` возвращал отрицательное целое число (обычно −1), если `a` меньше `b`; положительное целое число (обычно +1), если `a` больше `b`; и 0, если `a` и `b` равны. (Обозначение `<Key>`, представленное в разделе 4.3, гарантирует, что два сравниваемых объекта имеют одинаковый тип.)

Точный смысл операций «меньше», «больше» и «равно» зависит от типа данных, хотя реализации, не соблюдающие естественные математические законы, связанные с концепцией сравнения, приведут к непредсказуемым результатам. А если говорить более формально, метод `compareTo()` должен определять *линейный* (или *полный*) *порядок*. Это означает, что должны выполняться следующие свойства (здесь запись $x \leq y$ является сокращенным обозначением для `x.compareTo(y) <= 0`, а $x = y$ — для `x.compareTo(y) == 0`).

- Антисимметричность: если одновременно выполняются условия $x \leq y$ и $y \leq x$, то $x = y$[1].
- Транзитивность: если одновременно выполняются условия $x \leq y$ и $y \leq z$, то $x \leq z$.
- Полнота: либо $x \leq y$, либо $y \leq x$, либо и то и другое одновременно.

Эти три свойства выполняются для самых разнообразных известных способов упорядочения, включая алфавитное упорядочение для строк и упорядочение по возрастанию для целых и вещественных чисел. Мы будем называть линейный порядок, связанный с реализацией интерфейса `Comparable`, его *естественным порядком*. Тип Java `String` реализует `Comparable`, как и обертки для примитивных типов (такие, как `Integer` и `Double`), представленные в разделе 3.3.

Программа `Insertion` (листинг 4.2.4) реализует метод сортировки, который получает в аргументе массив объектов, реализующих `Comparable`, и переставляет элементы по возрастанию в соответствии с порядком, заданным методом `compareTo()`. Теперь мы можем использовать `Insertion.sort()` для сортировки массивов типа `String[]`, `Integer[]` или `Double[]`.

Обеспечить поддержку `Comparable` несложно, и это позволит вам сортировать пользовательские типы данных. Для этого необходимо включить в объявление класса раздел `implements Comparable`, а затем добавить метод `compareTo()`, определяющий

---

[1] Обычно свойство антисимметричности формулируют иначе: если $x < y$ — истинно, то $y < x$ — ложно (то есть предполагается строгое неравенство). Здесь уместно бы было сформулировать так: «Если $x \leq y$ — истинно, то $y \leq x$ истинно, *только* если $x = y$». — *Примеч. науч. ред.*

Листинг 4.2.4. Сортировка методом вставки

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {  // Сортировка a[] по возрастанию.
      int n = a.length;
      for (int i = 1; i < n; i++)
         // Вставка a[i] в позицию сортировки.
         for (int j = i; j > 0; j--)
            if (a[j].compareTo(a[j-1]) < 0)
                exchange(a, j-1, j);
            else break;
   }

   public static void exchange(Comparable[] a, int i, int j)
   {  Comparable temp = a[j]; a[j] = a[i]; a[i] = temp;  }

   public static void main(String[] args)
   {  // Чтение строк из стандартного ввода, их сортировка и вывод.
      String[] a = StdIn.readAllStrings();
      sort(a);
      for (int i = 0; i < a.length; i++)
         StdOut.print(a[i] + " ");
      StdOut.println();
   }
}
```

`a[]`	сортируемый массив
`n`	длина массива

Функция sort() реализует сортировку методом вставки. Она сортирует массивы любого типа данных, реализующего интерфейс Comparable (и, следовательно, содержит метод compareTo()). Реализация Insertion.sort() подходит только для небольших и/или почти упорядоченных массивов; для больших неупорядоченных массивов она работает слишком медленно.

```
% more 8words.txt
was had him and you his the but
% java Insertion < 8words.txt
and but had him his the was you
% java Insertion < TomSawyer.txt
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
```

линейный порядок. Например, чтобы тип данных `Counter` поддерживал `Comparable`, в листинг 3.3.2 вносятся следующие изменения:

```
public class Counter implements Comparable<Counter>
{
   private int count;
 ...
   public int compareTo(Counter b)
   {
     if      (count < b.count) return -1;
     else if (count > b.count) return +1;
     else                      return  0;
   }
 ...
}
```

Теперь можно использовать `Insertion.sort()` для сортировки массива объектов `Counter` по возрастанию поля `count`.

Эмпирический анализ

Программа `InsertionDoublingTest` (листинг 4.2.5) проверяет нашу гипотезу о том, что сортировка методом вставки для массивов со случайным исходным порядком выполняется за квадратичное время. Для этого она выполняет `Insertion.sort()` для *n* случайных объектов `Double` и вычисляет отношения времени выполнения при удвоении *n*. Это отношение сходится к 4, что подтверждает гипотезу о квадратичном времени выполнения (см. объяснения в последнем разделе). Мы рекомендуем вам запустить `InsertionDoublingTest` на вашем компьютере. Как обычно, для некоторых значений *n* может наблюдаться эффект кэширования или других системных механизмов, но квадратичное время выполнения должно быть достаточно очевидно, и вы быстро убедитесь в том, что сортировка методом вставки слишком медленна для больших объемов данных.

Зависимость от входных данных

Обратите внимание: программа `InsertionDoublingTest` получает аргумент командной строки `trials` и проводит для каждой длины массива `trials` испытаний (вместо одного). Как упоминалось ранее, одна из причин заключается в том, что время выполнения сортировки методом вставки зависит от входных значений. Это поведение заметно отличается от (например) программы `ThreeSum` и означает необходимость внимательной интерпретации результатов анализа. Было бы неправильно просто спрогнозировать, что время выполнения сортировки методом вставки будет квадратичным, потому некоторые входные данные могут приводить к линейному времени. Если быстродействие алгоритма зависит от входных данных, возможно, вы не сможете сделать точный прогноз без учета особенностей этих данных.

Поскольку во многих реальных приложениях время сортировки методом вставки *является* квадратичным, мы должны рассмотреть более быстрые алгоритмы

Листинг 4.2.5. Проверка гипотезы удвоения для сортировки методом вставки

```
public class InsertionDoublingTest
{
   public static double timeTrials(int trials, int n)
   {  // Сортировка случайных массивов размера n.
      double total = 0.0;
      Double[] a = new Double[n];
      for (int t = 0; t < trials; t++)
      {
         for (int i = 0; i < n; i++)
            a[i] = StdRandom.uniform(0.0, 1.0);
         Stopwatch timer = new Stopwatch();
         Insertion.sort(a);
         total += timer.elapsedTime();
      }
      return total;
   }
   public static void main(String[] args)
   {  // Вывод отношений удвоения для сортировки методом вставки.
      int trials = Integer.parseInt(args[0]);
      for (int n = 1024; true; n += n)
      {
         double prev = timeTrials(trials, n/2);
         double curr = timeTrials(trials, n);
         double ratio = curr / prev;
         StdOut.printf(„%7d %4.2f\n", n, ratio);
      }
   }
}
```

trials	количество испытаний
n	размер задачи
total	общие затраты времени
timer	таймер
a[]	сортируемый массив
prev	время выполнения для n / 2
curr	время выполнения для n
ratio	отношение времен выполнения

Метод timeTrials() выполняет сортировку Insertion.sort() для массивов случайных значений типа double. В первом аргументе n передается длина массива; второй аргумент trials определяет количество испытаний. Увеличение числа испытаний повышает точность результатов, потому что оно нивелирует эффекты, зависящие от системы, и фактическое время сортировки методом вставки зависит от входных данных.

```
% java InsertionDoublingTest 1
   1024 0.71
   2048 3.00
   4096 5.20
   8192 3.32
  16384 3.91
  32768 3.89
```

```
% java InsertionDoublingTest 10
   1024 1.89
   2048 5.00
   4096 3.58
   8192 4.09
  16384 4.83
  32768 3.96
```

сортировки. Как упоминалось в разделе 4.1, простейшие расчеты могут показать, что приобретение более быстрого компьютера может оказаться неэффективным. Словарь, научная или коммерческая база данных могут содержать миллиарды элементов; как отсортировать такой большой массив?

Сортировка слиянием

входные данные
```
was had him and you his the but
```
сортировка левой части
```
and had him was you his the but
```
сортировка правой части
```
and had him was but his the you
```
результат слияния
```
and but had him his the was you
```
Сортировка слиянием

Чтобы разработать более быстрый метод сортировки, мы воспользуемся рекурсией и принципом «разделяй и властвуй» (или декомпозиции). Каждый программист должен хорошо понимать этот принцип. Один из способов решения задачи заключается в том, чтобы разбить ее на две независимые задачи, справиться с ними по отдельности, а затем использовать решения для полной задачи. Чтобы отсортировать массив с применением этой стратегии, следует разделить его на две половины, отсортировать их независимо друг от друга, а затем произвести *слияние* половин для сортировки всего массива. Этот алгоритм известен под названием *сортировки слиянием.*

Алгоритм обрабатывает смежные подмассивы заданного массива; запись `a[lo,hi)` используется для обозначения элементов `a[lo]`, `a[lo+1]`, ..., `a[hi-1]` (как и при бинарном поиске, это полуоткрытый интервал, не включающий `a[hi]`). Сортировка `a[lo,hi)` производится по следующей рекурсивной стратегии.

- Базовый случай: если длина подмассива равна 0 или 1, массив уже отсортирован.
- Свертка: в противном случае вычислить `mid = lo + (hi - lo)/2`, провести рекурсивную сортировку двух подмассивов `a[lo,mid)` и `a[mid, hi)`, а затем объединить их.

Программа `Merge` (листинг 4.2.6) представляет собой реализацию этого алгоритма. Перестановка элементов массива осуществляется в коде, следующем за рекурсивными вызовами, где происходит слияние двух подмассивов, отсортированных рекурсивными вызовами. Как обычно, процесс слияния проще всего понять на примере трассировки. В коде индекс `i` используется для перебора первого подмассива, индекс `j` — для второго подмассива и еще один индекс, `k`, — для вспомогательного массива `aux[]`, который используется для временного хранения результата. Реализация слияния представляет собой цикл, в котором `aux[k]` присваивается либо `a[i]`, либо `a[j]` (с последующим увеличением `k` и индекса использованного подмассива). Если `i` или `j` достигает конца своего подмассива, то значение `aux[k]` присваивается из другого подмассива; в противном случае присваивается меньшее из значений `a[i]` или `a[j]`. После того как все значения из двух подмассивов будут скопированы в `aux[]`, отсортированный результат из `aux[]` копируется обратно в исходный массив. Просмотрите приведенную трассировку и убедитесь в том, что этот код всегда правильно объединяет два отсортированных подмассива для сортировки всего массива.

Рекурсивный метод гарантирует, что каждый из двух подмассивов будет упорядочен непосредственно перед слиянием. Чтобы понять суть происходящего, лучше всего изучить содержимое массива при возврате управления рекурсивным методом `sort()`. Далее приводится трассировка для нашего примера. Сначала `a[0]`

i	j	k	aux[k]	a[k]							
				0	1	2	3	4	5	6	7
				and	**had**	**him**	**was**	**but**	**his**	**the**	**you**
0	4	0	**and**	**and**	had	him	was	**but**	his	the	you
1	4	1	**but**	and	**had**	him	was	**but**	his	the	you
1	5	2	**had**	and	**had**	him	was	but	**his**	the	you
2	5	3	**him**	and	had	**him**	was	but	**his**	the	you
3	5	4	**his**	and	had	him	**was**	but	**his**	the	you
3	6	5	**the**	and	had	him	**was**	but	his	**the**	you
3	7	6	**was**	and	had	him	**was**	but	his	the	**you**
4	7	7	**you**	and	had	him	was	but	his	the	**you**

Трассировка слияния левого отсортированного подмассива с отсортированным правым подмассивом

объединяется с a[1] для создания отсортированного подмассива в a[0, 2), затем a[2] и a[3] объединяются для создания отсортированного подмассива в a[2, 4), затем два подмассива размера 2 объединяются для создания отсортированного подмассива в a[0, 4) и т. д. Если вы убеждены в том, что слияние работает правильно, остается только убедиться в правильности деления массива, и можно быть уверенным в том, что сортировка работает правильно. Обратите внимание: если количество элементов в сортируемом подмассиве нечетно, в левой половине будет на один элемент меньше, чем в правой.

	a[]							
	01234567							
	was	had	him	and	you	his	the	but
sort(a, aux, 0, 8)								
sort(a, aux, 0, 4)								
sort(a, aux, 0, 2)								
return	had	was	him	and	you	his	the	but
sort(a, aux, 2, 4)								
return	had	was	and	him	you	his	the	but
return	and	had	him	was	you	his	the	but
sort(a, aux, 4, 8)								
sort(a, aux, 4, 6)								
return	and	had	him	was	his	you	the	but
sort(a, aux, 6, 8)								
return	and	had	him	was	his	you	but	the
return	and	had	him	was	but	his	the	you
return	and	but	had	him	his	the	was	you

Трассировка рекурсивных вызовов при сортировке слиянием

Листинг 4.2.6. Сортировка слиянием

```
public class Merge
{
   public static void sort(Comparable[] a)
   {
      Comparable[] aux = new Comparable[a.length];
      sort(a, aux, 0, a.length);
   }

   private static void sort(Comparable[] a, Comparable[] aux,
                            int lo, int hi)
   {  // Сортировка a[lo, hi).
      if (hi - lo <= 1) return;
      int mid = lo + (hi-lo)/2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid, hi);
      int i = lo, j = mid;
      for (int k = lo; k < hi; k++)
         if      (i == mid)  aux[k] = a[j++];
         else if (j == hi)   aux[k] = a[i++];
         else if (a[j].compareTo(a[i]) < 0) aux[k] = a[j++];
         else                                aux[k] = a[i++];
      for (int k = lo; k < hi; k++)
         a[k] = aux[k];
   }

   public static void main(String[] args)
   {  /* См. листинг 4.2.4. */  }
}
```

a[lo, hi)	сортируемый подмассив
lo	наименьший индекс
mid	средний индекс
hi	наибольший индекс
aux[]	вспомогательный (временный) массив

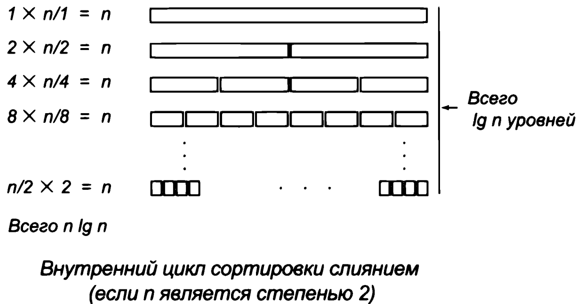
Функция sort() реализует алгоритм сортировки слиянием. Она сортирует массивы с любым типом данных, реализующим интерфейс Comparable. В отличие от Insertion. sort(), эта реализация подходит и для очень больших массивов.

```
% java Merge < 8words.txt
was had him and you his the but

% java Merge < TomSawyer.txt
... achievement aching aching acquire acquired ...
```

Анализ времени выполнения

Внутренний цикл сортировки слиянием напрямую связан со вспомогательным массивом. Два цикла `for` дают `n` итераций, а число выполняемых операций во внутреннем цикле пропорционально сумме длин подмассивов для всех вызовов рекурсивной функции. Чтобы оценить эту величину, следует расположить вызовы на уровнях в соответствии с размерами. Для простоты предположим, что n является степенью 2 и $n = 2^k$. На первом уровне имеем один вызов для размера n; на втором

уровне два вызова для размера $n/2$; на третьем четыре вызова для размера $n/4$ и т. д. до последнего уровня с $n/2$ вызовами для размера 2. Количество уровней равно $k = \lg n$, итого число операций во внутреннем цикле сортировки слиянием равно $n \lg n$. Эта формула подтверждает гипотезу о линейно-логарифмическом времени выполнения сортировки слиянием. *Примечание*: если n не является степенью 2, то подмассивы на каждом уровне не обязательно имеют одинаковые размеры, но количество уровней все равно остается логарифмическим, так что гипотеза о линейно-логарифмическом времени остается оправданной для любых n (см. упражнения 4.2.18 и 4.2.19).



Внутренний цикл сортировки слиянием (если n является степенью 2)

Мы рекомендуем провести тест гипотезы удвоения (как в листинге 4.2.5) для `Merge.sort()` на вашем компьютере. Тогда вы убедитесь, что для больших массивов он выполняется намного быстрее, чем `Insertion.sort()`, и большие массивы сортируются относительно быстро. Проверка гипотезы о линейно-логарифмическом времени выполнения требует чуть большего объема работы, но вы наверняка согласитесь, что сортировка слиянием позволяет решать задачи сортировки, совершенно неподвластные для алгоритмов «грубой силы», таких как сортировка методом вставки.

Пропасть между квадратичным и линейно-логарифмическим временем

Различия между временем n^2 и $n \log n$ имеют огромное значение для практических применений — как и аналогичный скачок, обеспечиваемый бинарным поиском по сравнению с линейным. *Понимание масштаба этих различий — важный шаг на пути к пониманию важности проектирования и анализа алгоритмов.* Для очень многих важных вычислительных задач сокращение времени выполнения от квадратичного до линейно-логарифмического — такое, которое обеспечивается сортировкой слиянием, — означает практическую возможность или невозможность решения задачи.

Алгоритмы «разделяй и властвуй»

Та же парадигма «разделяй и властвуй» эффективно работает во многих важных задачах; вы узнаете об этом в учебном курсе проектирования алгоритмов. А пока мы настоятельно рекомендуем изучить упражнения в конце этого раздела — среди них вы найдете подборку задач, для которых алгоритмы «разделяй и властвуй» предоставляют реальные решения, невозможные без применения таких алгоритмов.

Сведение к сортировке

Задача A *сводится* к задаче B, если решение B может использоваться для решения A. Проектирование нового алгоритма «разделяй и властвуй» сродни решению головоломки, требующей опыта и изобретательности, и на первых порах вы не всегда будете уверены в том, что это вообще возможно. Но, как нередко бывает, в этой ситуации оказывается эффективным более простой подход: столкнувшись с новой задачей, спросите себя, как бы вы стали решать ее, если бы данные были отсортированы. Часто в таком случае задача может быть решена относительно простым линейным перебором отсортированных данных. Например, представьте, что вам нужно определить, что все значения элементов в массиве различны. Задача сводится к сортировке, потому что вы можете отсортировать массив, а затем перебрать его элементы и проверить, совпадают ли значения любых двух соседних, и если нет, значит, все значения различны. Или другой пример: простой способ реализации `StdStats.median()` (см. раздел 2.2) основан на сведении выбора к сортировке. Сейчас будет рассмотрен более сложный пример, а в упражнениях в конце главы вы найдете много других примеров.

Алгоритм сортировки слиянием был предложен Джоном фон Нейманом — известным физиком, который одним из первых осознал важность обработки данных для научных исследований. Фон Нейман внес значительный вклад в теорию вычислительной техники, включая создание базовой концепции архитектуры компьютеров, которая используется с 1950-х годов. В том, что касается прикладного программирования, фон Нейман сформулировал ряд важных положений.

- Сортировка является важнейшим компонентом во многих приложениях.
- Алгоритмы с квадратичным временем слишком медленны для практических целей.
- Эффективны алгоритмы «разделяй и властвуй».
- Важно уметь доказывать правильность программ и знать, с какими затратами связано их выполнение.

Современные компьютеры на несколько порядков быстрее и оснащены на несколько порядков большим объемом памяти, чем компьютеры времен фон Неймана, но эти базовые концепции продолжают играть важную роль и в наше время. Люди,

эффективно и успешно использующие компьютеры, знают (как и фон Нейман), что алгоритмы «грубой силы» часто не подходят для решения задач.

Приложение: подсчет частот вхождения

Программа `FrequencyCount` (листинг 4.2.7) читает последовательность строк из стандартного потока ввода, а затем выводит таблицу различающихся строк и количества вхождений каждой строки в порядке убывания частоты. Такая задача может встречаться в разных контекстах: у лингвиста — при изучении частотных закономерностей слов в длинных текстах, у ученого — при поиске часто встречающихся событий в экспериментальных данных, у продавца — при определении клиентов, чаще всего встречающихся в длинном списке транзакций, а у сетевого аналитика — при поиске самых активных пользователей. В любой из этих задач могут быть задействованы миллионы строк, поэтому нам нужен линейно-логарифмический (или более эффективный) алгоритм. Программа `FrequencyCount` — пример разработки алгоритма сведением к сортировке. В действительности он выполняет *две* сортировки.

Вычисление частот

Первый шаг — сортировка входных строк. В данном случае нас интересует даже не столько упорядочивание строк, сколько то, что *сортировка сводит вместе одинаковые строки*. Если стандартный ввод содержит строки

```
to be or not to be to
```

то результат сортировки выглядит так:

```
be be not or to to to
```

Одинаковые строки — например, два вхождения *be* и три вхождения *to* — занимают смежные позиции массива. Теперь, когда все одинаковые строки находятся в массиве рядом друг с другом, для подсчета частот их появлений достаточно одного прохода по массиву. Тип данных `Counter`, который рассматривался в разделе 3.3, идеально подходит для этой цели. Напомним, что тип `Counter` (листинг 3.3.2) содержит строковую переменную экземпляра (инициализируется в аргументе конструктора), переменную экземпляра для счетчика (инициализируется 0) и метод экземпляра `increment()`, который увеличивает счетчик на 1. В программе определяется целочисленная переменная `m` и массив `zipf[]` с объектами `Counter`, а для каждой строки выполняются следующие действия.

- Если строка не равна предыдущей, создать новый объект `Counter` и увеличить `m`.
- Увеличить счетчик последнего объекта `Counter`.

В конце перебора значение `m` равно количеству разных строковых значений, а `zipf[i]` содержит строковое значение и частоту `i`-й строки.

i	M	a[i]	zipf[i].value()			
			0	1	2	3
	0					
0	1	be	**1**			
1	1	be	**2**			
2	2	not	2	**1**		
3	3	or	2	1	**1**	
4	4	to	2	1	1	**1**
5	4	to	2	1	1	**2**
6	4	to	2	1	1	**3**
			2	**1**	**1**	**3**

Подсчет частот

Сортировка частот

Затем объекты `Counter` сортируются по частотам. Это можно сделать в клиентском коде при условии, что `Counter` реализует интерфейс `Comparable`, а его метод `compareTo()` сравнивает объекты по значению счетчика (см. упражнение 4.2.14). После того как это будет сделано, остается просто отсортировать массив! Учтите, что `FrequencyCount` выделяет память для `zipf[]` под максимально возможную длину и сортирует подмассив вместо того, чтобы сделать дополнительный проход по массиву `words[]` для подсчета разных строк перед выделением памяти для `zipf[]`. Модификация программы `Merge` (листинг 4.2.6), сортирующая подмассивы, остается читателю для самостоятельной работы (см. упражнение 4.2.15).

	i	zipf[i]
	До	
0	2	be
1	1	not
2	1	or
3	3	to
	После	
0	1	not
1	1	or
2	2	be
3	3	to

Сортировка частот

Закон Ципфа

Ситуация, представленная в приложении `FrequencyCount`, по сути, представляет собой элементарный лингвистический анализ: какие слова чаще всего встречаются в тексте? Так называемый *закон Ципфа* гласит, что частота использования i-го по частоте слова в тексте, содержащем m разных слов, пропорциональна $1/i$, а коэффициент пропорциональности является обратной величиной по отношению к гармоническому числу H_m. Например, второе по частоте слово должно встречаться примерно вдвое реже, чем первое. Эта эмпирическая гипотеза действует в чрезвычайно широком спектре ситуаций, от финансовых данных до веб-статистики. Тестовый клиент из листинга 4.2.7 проверяет закон Ципфа для базы данных с миллиардом фраз, случайно загруженных из Интернета (см. сайт книги).

Весьма вероятно, что в будущем вам придется писать программу для простой задачи, которая легко решается предварительным применением сортировки. Сколько разных значений содержит массив? Какое значение встречается чаще других? Какое значение является медианой? С линейно-логарифмическим алгоритмом сортировки, таким как сортировка слиянием, эти (и многие другие похожие) задачи можно решать даже для очень больших наборов данных. Типичный пример такого рода — программа `FrequencyCount`, использующая две отдельные сортировки. Если сортировка неприменима напрямую, возможно, вам стоит применить другие алгоритмы «разделяй и властвуй» или иные, более сложные методы. Без хороших алгоритмов (и понимания их характеристик быстродействия) вы можете столкнуться с тем, что даже ваш дорогой и быстрый компьютер не справляется с вроде бы простой задачей. Если набор задач, которые вы умеете эффективно решать, будет постоянно расширяться, вы поймете, что ваш компьютер способен работать гораздо эффективнее, чем вы могли себе представить.

Выводы

В подавляющем большинстве программ, которые вы напишете, для ограничения сложности при решении новой задачи будет разрабатываться простое, правильное решение с разбивкой программы на модули умеренного размера, тестированием и отладкой решения. С самого начала в этой книге мы старались разрабатывать программы по этим правилам. Но со временем, когда вы перейдете к более сложным приложениям, станет понятно, что ясного и правильного решения не всегда достаточно, потому что ограничением может оказаться «цена» решения (необходимые для него ресурсы). Примеры в этом разделе демонстрируют этот факт.

- *Учитывайте затраты ресурсов на вычисления.* Если вам удастся быстро решить маленькую задачу с простым алгоритмом — хорошо. Но если вы занимаетесь задачей с большим объемом данных и трудоемкими вычислениями, затраты уже нельзя игнорировать.
- *Сведение к известной задаче.* Применение сортировки для подсчета частот показывает, как важно понимать фундаментальные алгоритмы и применять их для решения задач.

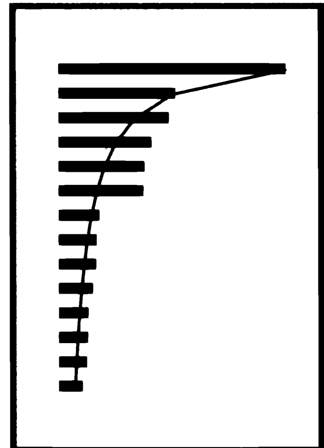
Листинг 4.2.7. Подсчет частот вхождения

```
public class FrequencyCount
{
   public static void main(String[] args)
   {  // Вывод входных строк в порядке убывания частот.
      String[] words = StdIn.readAllStrings();
      Merge.sort(words);
      Counter[] zipf = new Counter[words.length];
      int m = 0;
      for (int i = 0; i < words.length; i++)
      {  // Создание нового или увеличение предыдущего счетчика.
         if (i == 0 || !words[i].equals(words[i-1]))
            zipf[m++] = new Counter(words[i], words.length);
         zipf[m-1].increment();
      }
      Merge.sort(zipf, 0, m);
      for (int j = m-1; j >= 0; j--)
         StdOut.println(zipf[j]);
   }
}
```

s	входные данные
words[]	строки из входных данных
zipf[]	массив объектов Counter
m	количество разных строк

Программа сортирует слова из стандартного потока ввода, использует отсортированный список для подсчета вхождений каждого слова, после чего сортирует частоты. Тестовый файл, используемый в примере, содержит более 20 миллионов слов. На диаграмме отношение i-й частоты к первой (столбцы) сравнивается с графиком 1/i (линия).

```
% java FrequencyCount < Leipzig1M.txt
the: 1160105
of: 593492
to: 560945
a: 472819
and: 435866
in: 430484
for: 205531
The: 192296
that: 188971
is: 172225
said: 148915
on: 147024
was: 141178
by: 118429
...
```



- *«Разделяй и властвуй»*. Поразмыслите над эффективностью парадигмы «разделяй и властвуй», продемонстрированной на примере разработки линейно-логарифмического алгоритма сортировки (сортировки слиянием), лежащего в основе решения многих вычислительных задач. Впрочем, «разделяй и властвуй» не единственный подход к разработке эффективных алгоритмов.

Со времени появления первых вычислительных устройств специалисты разрабатывали алгоритмы для эффективного решения практических задач (такие, как алгоритм бинарного поиска или сортировки слиянием). Область исследований, называемая *проектированием и анализом алгоритмов*, включает изучение парадигм проектирования («разделяй и властвуй», динамическое программирование и т. д.), разработку алгоритмов для решения фундаментальных задач (сортировка, поиск и т. д.) и методы разработки гипотез об эффективности алгоритмов. Реализации многих из этих алгоритмов встречаются в библиотеках Java или других специализированных библиотеках, но понимание логики базовых вычислительных средств сродни пониманию базовых математических или научных методов. Вы можете воспользоваться пакетом матричных вычислений для нахождения собственных чисел матрицы, но, чтобы понять суть происходящего, вам все равно понадобится курс линейной алгебры. Теперь, когда вы знаете, что выбор быстрого алгоритма может означать выбор между бегом на месте и решением практических задач, вы будете обращать особое внимание на ситуации, зависящие от проектирования и анализа алгоритмов, в которых можно воспользоваться известными эффективными алгоритмами, такими как бинарный поиск и сортировка слиянием.

Вопросы и ответы

В. Почему необходимо прикладывать такие усилия, чтобы убедиться в правильности программы?

О. Чтобы избавиться от серьезных проблем. Важнейший пример такого рода — бинарный поиск. Например, вы уже понимаете суть бинарного поиска; классический пример упражнения из области программирования — написать версию, использующую цикл `while` вместо рекурсии. Попробуйте решить упражнение 4.2.2, не обращаясь к коду, приведенному в книге. В своем знаменитом эксперименте Джон Бентли однажды предложил эту задачу нескольким профессиональным программистам, но большинство их решений оказались *ошибочными*.

В. Включены ли реализации сортировки и поиска в библиотеку Java?

О. Да. Пакет Java `java.util` содержит статические методы `Arrays.sort()` и `Arrays.binarySearch()`, реализующие сортировку слиянием и бинарный поиск соответственно. Вообще говоря, каждый метод представляет семейство перегруженных методов: для типов, реализующих `Comparable`, и по одному для каждого примитивного типа.

В. Почему бы просто не использовать их?

О. Используйте, конечно. Как и во многих темах, которые вы изучали, понимание сути происходящего поможет вам использовать эти средства более эффективно.

В. Объясните, почему для вычисления индекса, находящегося на середине интервала между `lo` и `hi`, мы используем запись `lo+(hi-lo)/2` вместо `(lo+hi)/2`?

О. Чтобы избежать ошибки, если при вычислении `lo+hi` происходит переполнение `int`.

В. Почему я получаю предупреждение о непроверенной или небезопасной операции при компиляции программ Insertion.java и Merge.java?

О. Аргумент `sort()` представляет собой массив с объектами `Comparable`, но с технической точки зрения ничто не мешает его элементам относиться к иным типам. Чтобы избавиться от предупреждения, приведите сигнатуру к следующему виду:

```
public static <Key extends Comparable<Key>> void sort(Key[] a)
```

Запись `<Key>` будет описана в следующем разделе, когда мы будем рассматривать *обобщения*.

Упражнения

4.2.1. Разработайте реализацию `Questions` (листинг 4.2.1), который получает максимальное число `n` в качестве аргумента командной строки. Докажите, что ваша реализация работает правильно.

4.2.2. Разработайте нерекурсивную версию `BinarySearch` (листинг 4.2.3).

4.2.3. Измените программу `BinarySearch` (листинг 4.2.3) так, чтобы при нахождении ключа поиска в массиве возвращался наименьший индекс `i`, с которым элемент `a[i]` равен ключу, а в противном случае возвращалось значение `-i`, где `i` — наименьший индекс, для которого `a[i]` больше ключа.

4.2.4. Опишите, что произойдет при использовании бинарного поиска в неупорядоченном массиве. Почему не следует проверять массив на упорядоченность перед каждым вызовом бинарного поиска? Можно ли проследить за тем, что элементы, проверяемые в ходе бинарного поиска, следуют по возрастанию?

4.2.5. Опишите, почему при бинарном поиске желательно использовать неизменяемые ключи.

4.2.6. Добавьте в `Insertion` код для получения трассировки, приведенной в тексте.

4.2.7. Добавьте в `Merge` код для получения трассировки следующего вида:

```
% java Merge < tiny.txt
was had him and you his the but
had was
        and him
and had him was
                his you
                        but the
                but his the you
and but had him his the was you
```

4.2.8. Приведите трассировку для сортировки методом вставки и сортировки слиянием (в стиле трассировок, приведенных в тексте) для входных данных `it was the best of times it was`.

4.2.9. Реализуйте более общую версию программы 4.2.2, которая применяет поиск методом деления пополам к любой монотонно возрастающей функции. Используйте функциональное программирование в том же стиле, что и в примере с числовым интегрированием из раздела 3.3.

4.2.10. Напишите фильтр `DeDup`, который читает строки из стандартного ввода и выводит их в стандартный вывод с удалением дубликатов (и в порядке сортировки).

4.2.11. Измените программу `StockAccount` (листинг 3.2.8), чтобы она реализовала интерфейс `Comparable` (со сравнением портфелей акций по именам). *Подсказка*: для выполнения рутинных операций используйте метод `compareTo()` типа данных `String`.

4.2.12. Измените программу `Vector` (листинг 3.3.3), чтобы она реализовала интерфейс `Comparable` (с лексикографическим сравнением векторов по координатам).

4.2.13. Измените программу `Time` (листинг 3.3.21), чтобы она реализовала интерфейс `Comparable` (с хронологическим сравнением моментов времени).

4.2.14. Измените программу `Counter` (листинг 3.3.2), чтобы она реализовала интерфейс `Comparable` (со сравнением объектов по частоте вхождений).

4.2.15. Добавьте в программу `Insertion` (листинг 4.2.4) и `Merge` (листинг 4.2.6) методы для обеспечения сортировки подмассивов.

4.2.16. Разработайте нерекурсивную версию сортировки слиянием (листинг 4.2.6). Для простоты считайте, что количество элементов `n` равно степени 2. *Дополнительное задание*: позаботьтесь о том, чтобы ваша программа работала даже в том случае, если `n` не является степенью 2.

4.2.17. Определите распределение частот слов в своем любимом литературном произведении. Подчиняется ли оно закону Ципфа?

4.2.18. Проанализируйте математическими методами количество сравнений, выполняемых алгоритмом сортировки слиянием для сортировки массива длины `n`. Для простоты считайте, что количество элементов `n` равно степени 2.

Ответ. Пусть $M(n)$ — количество сравнений сортировки слиянием для массива длины n. Слияние двух подмассивов, общая длина которых равна n, требует от ½ n до $n - 1$ сравнений. Таким образом, $M(n)$ удовлетворяет следующему рекуррентному отношению:

$$M(n) \leq 2M(n/2) + n$$

с $M(1) = 0$. Заменяя n на 2^k, получаем:

$$M(2^k) \leq 2\,M(2^{k-1}) + 2^n.$$

Результат похож на рекуррентное отношение, которое приводилось для бинарного поиска, хотя и сложнее его. Но разделив обе стороны на 2^n, мы получим

$$M(2^k)/2^k \le M(2^{k-1})/2^{k-1} + 1,$$

что в точности соответствует рекуррентному отношению для бинарного поиска, а именно $M(2^k)/2^k \le T(2^k) = n$. Снова подставляя n вместо 2^k (и $\lg n$ вместо k), получаем $M(n) \le n \lg n$. Аналогичные рассуждения показывают, что $M(n) \ge \frac{1}{2} n \lg n$.

4.2.19. Проанализируйте сортировку слиянием для n, не являющегося степенью 2.

Частичное решение. Если число n нечетное, один подмассив содержит на один элемент больше другого. Следовательно, если n не является степенью 2, подмассивы на одном уровне не обязательно имеют одинаковый размер. Тем не менее каждый элемент входит в некоторый подмассив, а количество уровней остается логарифмическим, так что линейно-логарифмическая гипотеза выполняется для всех n.

Упражнения повышенной сложности

Следующие упражнения помогут вам приобрести опыт разработки быстрых решений типичных задач. Рассмотрите возможность применения бинарного поиска или сортировки слиянием или разработки собственного алгоритма «разделяй и властвуй». Реализуйте и протестируйте свой алгоритм.

4.2.20. *Медиана*. Добавьте в `StdStats` (листинг 2.2.4) метод `median()` для вычисления медианы массива из n целых чисел с линейно-логарифмическим временем. *Подсказка*: задача сводится к сортировке.

4.2.21. *Наиболее вероятное значение*. Добавьте в `StdStats` (листинг 2.2.4) метод `mode()` для вычисления наиболее вероятного значения (то есть значения, встречающегося чаще любого другого) в массиве из n целых чисел с линейно-логарифмическим временем. *Подсказка*: задача сводится к сортировке.

4.2.22. *Целочисленная сортировка*. Напишите фильтр с линейным временем, который читает из стандартного ввода последовательность целых чисел в диапазоне от 0 до 99 и выводит в стандартный вывод те же числа в отсортированном порядке. Например, для входной последовательности

```
98 2 3 1 0 0 0 3 98 98 2 2 2 0 0 0 2
```

программа должна вывести следующий результат:

```
0 0 0 0 0 0 1 2 2 2 2 2 3 3 98 98 98
```

4.2.23. *Наименьшее и наибольшее целое число*. Для заданного массива объектов `Comparable` напишите функции `floor()` и `ceiling()`, возвращающие индекс наибольшего (или наименьшего) элемента, который не больше (или не меньше) аргумента, за логарифмическое время.

4.2.24. *Битонический максимум.* Массив называется *битоническим*, если он состоит из возрастающей последовательности ключей, за которыми немедленно следует убывающая последовательность ключей. Для заданного битонического массива спроектируйте логарифмический алгоритм для нахождения индекса максимального ключа.

4.2.25. *Поиск в битоническом массиве.* Для заданного битонического массива из n разных целых чисел спроектируйте алгоритм с логарифмическим временем для проверки того, присутствует ли заданное целое число в массиве.

4.2.26. *Ближайшая пара.* Для заданного массива из n вещественных чисел спроектируйте алгоритм с линейно-логарифмическим временем для нахождения пары целых чисел с ближайшими значениями.

4.2.27. *Дальняя пара.* Для заданного массива из n вещественных чисел спроектируйте алгоритм с линейно-логарифмическим временем для нахождения пары целых чисел, по своим значениям удаленных друг от друга на максимальное расстояние.

4.2.28. *Сумма пар.* Для заданного массива из n целых чисел спроектируйте алгоритм с линейно-логарифмическим временем для определения того, найдутся ли в массиве два числа, сумма которых равна 0.

4.2.29. *Сумма троек.* Для заданного массива из n целых чисел спроектируйте алгоритм для определения того, найдутся ли в массиве два числа, сумма которых равна 0. Порядок роста времени выполнения вашей программы должен быть равен $n^2 \log n$. *Дополнительное задание*: разработайте программу, которая решает задачу за квадратичное время

4.2.30. *Доминирующее значение.* Значение в массиве длины n называется *доминирующим*, если оно встречается более $n/2$ раз. Для заданного массива строк спроектируйте алгоритм с линейным временем для нахождения доминирующего значения (если оно существует).

4.2.31. *Наибольший пустой интервал.* Заданы n временных меток для моментов запросов файлов с веб-сервера. Требуется найти наибольший интервал времени, в течение которого не запрашивался ни один файл. Напишите программу для решения этой задачи за линейно-логарифмическое время.

4.2.32. *Беспрефиксные коды.* В области сжатия данных множество строк называется беспрефиксным, если ни одна строка не является префиксом другой. Например, множество { `01, 10, 0010, 1111` } является беспрефиксным, тогда как множество { `01, 10, 0010, 1010` } беспрефиксным не является, потому что 10 — префикс 1010. Напишите программу, которая читает множество строк из стандартного ввода и определяет, является ли это множество беспрефиксным.

4.2.33. *Разбиение.* Разработайте алгоритм с линейным временем для сортировки массива объектов `Comparable`, заведомо принимающих не более двух разных значений. *Подсказка*: используйте два указателя; один начинает с левого конца и двига-

ется направо, а другой начинает с правого конца и двигается налево. Все элементы слева от левого указателя равны меньшему из двух значений, а все элементы справа от правого указателя равны большему из двух значений.

4.2.34. *Задача о голландском флаге.* Спроектируйте алгоритм с линейным временем для сортировки массива объектов `Comparable`, заведомо принимающих не более трех разных значений. (Эдсгер Дейкстра назвал эту задачу «задачей о голландском флаге», потому что результат из трех «полос» значений напоминает три полосы государственного флага Нидерландов.)

4.2.35. *Быстрая сортировка.* Напишите рекурсивную программу, которая сортирует массив объектов `Comparable`, используя (в виде подпрограммы) алгоритм разбиения из предыдущего упражнения: сначала случайный элемент `v` выбирается в качестве элемента разбиения, затем массив разбивается на левый подмассив, содержащий все элементы, меньшие `v`; средний подмассив со всеми элементами, равными `v`; и правый подмассив со всеми элементами, большими `v`. Далее выполняется рекурсивная сортировка левого и правого подмассивов.

4.2.36. *Обратное доменное имя.* Напишите фильтр, который читает последовательность доменных имен из стандартного ввода и выводит обратные доменные имена в порядке сортировки. Например, обратным доменным именем для *cs.princeton.edu* является *edu.princeton.cs*. Такие вычисления могут пригодиться, например, для анализа веб-журналов. Для этого создайте тип данных `Domain`, реализующий интерфейс `Comparable` (с использованием порядка обратных доменных имен).

4.2.37. *Локальный минимум в массиве.* Для заданного массива из n вещественных чисел разработайте алгоритм с логарифмическим временем для нахождения локального минимума (индекса `i`, для которого `a[i] < a[i-1] and a[i] < a[i+1]`).

4.2.38. *Дискретное распределение.* Разработайте быстрый алгоритм для многократного генерирования псевдослучайных чисел с дискретным распределением: для заданного массива `a[]` неотрицательных вещественных чисел, сумма которых равна 1, значение индекса `i` возвращается с вероятностью `a[i]`. Сформируйте массив `sum[]` накопленных сумм, для которых `sum[i]` содержит сумму первых `i` элементов `a[]`. Затем сгенерируйте случайное вещественное число `r` от 0 до 1 и используйте бинарный поиск для возвращения индекса `i`, для которого `sum[i]` ≤ `r` < `sum[i+1]`. Сравните быстродействие этого решения с решением из `RandomSurfer` (листинг 1.6.2).

4.2.39. *Вычисление волатильности.* Обычно волатильность δ является неизвестной величиной в формуле Блэка — Шоулза (см. упражнение 2.1.28). Напишите программу, которая читает из командной строки значения `s`, `x`, `r`, `t` и текущую цену европейских опционов и использует поиск методом половинного деления для вычисления δ.

4.2.40. *Порог протекания.* Напишите клиент `Percolation` (листинг 2.4.1), который использует поиск методом половинного деления для оценки величины порога протекания.

4.3. Стеки и очереди

В этом разделе будут рассмотрены две тесно связанные структуры (и соответствующие им типы данных) для работы с коллекциями объектов произвольного размера: *стек* и *очередь*. Стеки и очереди являются особыми случаями идеи коллекции. Объекты, входящие в коллекцию, называются *элементами*. Коллекция характеризуется четырьмя операциями: создание коллекции, вставка элемента, удаление (извлечение) элемента и проверка наличия элементов в коллекции.

Когда мы вставляем элемент в коллекцию, наши намерения ясны. Но какой элемент должен выбираться при удалении из коллекции? Каждый тип коллекции характеризуется правилом, используемым при удалении, и каждый тип может адаптироваться к различным реализациям с разными характеристиками быстродействия. Вы уже встречали разные правила удаления элементов в реальных ситуациях, даже если и не задумывались о них. Например, по правилу очереди всегда удаляется тот элемент, который находится в коллекции дольше всех. Эта дисциплина известна под названием FIFO (First-In, First-Out), или «первым вошел, первым вышел». Люди, стоящие в очереди в кассу, подчиняются этой дисциплине: очередь выстраивается в порядке прихода, и из очереди выходит тот человек, который находился в ней дольше всех остальных.

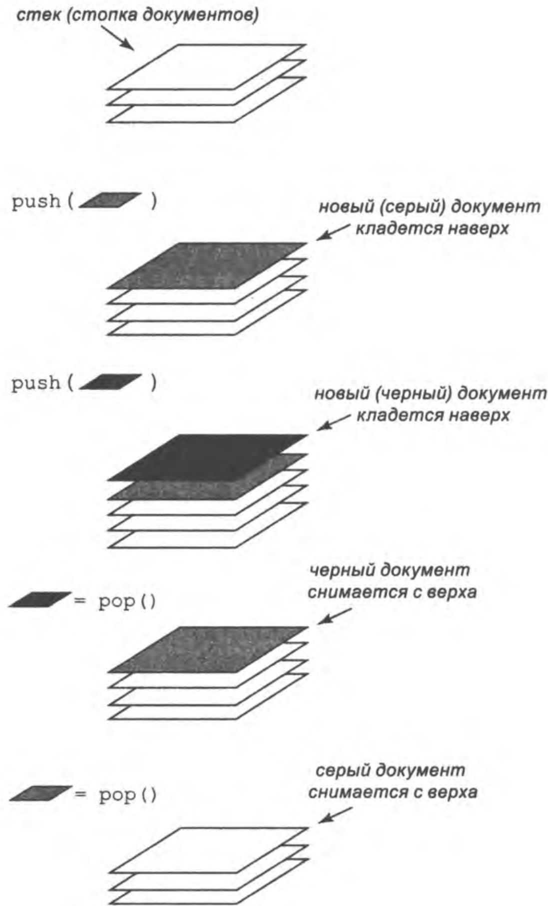
Для стека используется противоположная дисциплина: из коллекции всегда удаляется элемент, который провел в ней *меньше* всего времени. Эта дисциплина известна под названием LIFO (Last-In, First-Out), или «последним вошел, первым вышел». Например, похожий на LIFO порядок используется при входе и выходе в пассажирский салон самолета: люди, сидящие ближе ко входу, заходят последними и выходят раньше тех, кто сел до них.

Стеки и очереди находят широкое применение, поэтому вы должны знать их основные свойства и ситуации, для которых эти типы данных лучше подходят. Это отличные примеры фундаментальных структур данных, применяемых для решения высокоуровневых задач программирования. Они широко используются в системном и прикладном программировании, как вы убедитесь на примерах в этом разделе и разделе 4.5.

Стек

Стек — коллекция, работающая по принципу LIFO (Last-In, First-Out).

Дисциплина LIFO встречается в некоторых приложениях, которыми вы регулярно пользуетесь на своем компьютере. Например, многие пользователи упорядочивают свою электронную почту в стек: сообщения входят сверху (при получении) и извлекаются сверху (при этом сообщение, полученное последним, извлекается первым (LIFO). Преимущество такой стратегии заключается в том, что пользователь сразу видит новые сообщения при получении; недостаток — в том, что некоторые старые сообщения могут остаться непрочитанными, если стек никогда не опустошается.



Операции со стеком

Вероятно, другой распространенный пример стека уже встречался вам в ходе интернет-серфинга. Когда вы щелкаете по гиперссылке, ваш браузер выводит новую страницу (и добавляет ее в стек). Вы можете щелкать по ссылкам, чтобы посещать новые страницы, но всегда можете вернуться к предыдущей странице кнопкой Назад (извлечь страницу из стека). Дисциплина LIFO, обеспечиваемая стеком, демонстрирует как раз такое поведение, которого вы ожидаете.

Все эти примеры использования стеков интуитивно понятны, но, возможно, они вас еще не убедили. Стеки имеют фундаментальное значение для обработки данных, но мы вернемся к обсуждению практических применений позднее. А пока вы должны просто понять, как работают стеки и как они реализуются.

Стеки широко использовались с начала эпохи программирования. Краткая сводка операций со стеком приведена в следующей таблице.

public class *StackOfStrings	
*StackOfStrings()	создает пустой стек
boolean isEmpty()	стек пуст?
void push(String item)	вставляет строку в стек
String pop()	извлекает и возвращает строку, которая была вставлена в стек последней

API стека со строковыми элементами

Звездочка (*) означает, что мы будем рассматривать более одной реализации этого API (в этом разделе рассматриваются три такие реализации: `ArrayStackOfStrings`, `LinkedStackOfStrings` и `ResizingArrayStackOfStrings`). API также включает метод для проверки пустого стека; ответственность за вызов `isEmpty()`, предотвращающий вызов `pop()` при пустом стеке, остается обязанностью клиента.

В этом API действует важное ограничение, создающее неудобства в приложениях: нам хотелось бы иметь возможность создавать стеки для данных других типов, а не только для строк. Позднее в этом разделе мы покажем, как устранить это ограничение (и объясним, почему это важно делать).

Реализация в виде массива

Идея о представлении стека в формате массива естественна, но, прежде чем читать дальше, задумайтесь, как бы вы реализовали класс `ArrayStackOfStrings`.

Первая проблема, с которой вы можете столкнуться, — реализация конструктора `ArrayStackOfStrings()`. Очевидно, вам понадобится переменная экземпляра `items[]` для массива строк, содержащего элементы стека, но каким должен быть размер этого массива? Одно из возможных решений — начать с массива длины 0 и всегда приводить длину массива в соответствие с размером стека, но при таком решении придется выделять память для нового массива и копировать все элементы при каждом выполнении операции `push()` или `pop()`, а такое решение окажется излишне неэффективным и громоздким. Временно мы просто обойдем эту проблему, разрешив клиенту передать конструктору аргумент с максимальным размером стека.

Следующая проблема может возникнуть из-за естественного решения о хранении n элементов в массиве в порядке их вставки: элемент, который был вставлен последним, хранится в `items[0]`, а элемент, с момента вставки которого прошло больше всего времени, хранится в `items[n-1]`. Но тогда при каждой операции вставки или извлечения из стека все элементы придется перемещать в соответствии с новым состоянием стека. В более простом, более эффективном решении элементы хранятся в противоположном порядке: элемент, вставленный последним, хранится в `items[n-1]`, а самый давний элемент — в `items[0]`. Такая политика позволит добавлять и удалять элементы в конце массива без перемещения других элементов.

Трудно представить более простую реализацию API стека, чем в программе `ArrayStackOfStrings` (листинг 4.3.1) — все методы состоят из одной строки! Переменные экземпляра — массив `items[]`, содержащий элементы стека, и целочисленная переменная `n`, определяющая количество элементов в стеке. Чтобы удалить элемент из стека, мы уменьшаем `n` и возвращаем `items[n]`; чтобы вставить новый элемент, мы сохраняем в `items[n]` новый элемент, после чего увеличиваем `n`. Эти операции сохраняют следующие свойства.

- Количество элементов в стеке равно `n`.
- Стек пуст, когда значение `n` равно 0.
- Элементы стека хранятся в массиве в порядке их вставки.
- Последний вставленный элемент (если стек не пуст) хранится в `items[n-1]`.

StdIn	StdOut	n	items[]				
			0	1	2	3	4
		0					
to		1	to				
be		2	to	be			
or		3	to	be	or		
not		4	to	be	or	not	
to		5	to	be	or	not	to
-	to	4	to	be	or	not	to
be		5	to	be	or	not	be
-	be	4	to	be	or	not	be
-	not	3	to	be	or	that	be
that		4	to	be	or	that	be
-	that	3	to	be	or	that	be
-	or	2	to	be	or	that	be
-	be	1	to	be	or	that	be
is		2	to	is	or	not	to

Трассировка тестового клиента ArrayStackOfStrings

Как обычно, чтобы убедиться в правильности работы реализации, проще всего мыслить понятиями инвариантов. Убедитесь в том, что вы полностью понимаете эту реализацию. Возможно, для этого лучше всего тщательно изучить трассировку содержимого стека для серии операций `push()` и `pop()`. Тестовый клиент `ArrayStackOfStrings` позволяет провести тестирование с произвольной последовательностью операцией: операция `push()` выполняется для каждой строки

Листинг 4.3.1. Стек строк (реализация в виде массива)

```
public class ArrayStackOfStrings
{
   private String[] items;
   private int n = 0;

   public ArrayStackOfStrings(int capacity)
   {  items = new String[capacity];  }

   public boolean isEmpty()
   {  return (n == 0);  }

   public void push(String item)
   {  items[n++] = item;  }

   public String pop()
   {  return items[--n];  }

   public static void main(String[] args)
   {  // Создание стека заданной емкости; занесение и извлечение строк
      // в соответствии с  командами из стандартного потока ввода.
      int cap = Integer.parseInt(args[0]);
      ArrayStackOfStrings stack = new ArrayStackOfStrings(cap);
      while (!StdIn.isEmpty())
      {
         String item = StdIn.readString();
         if (!item.equals("-"))
            stack.push(item);
         else
            StdOut.print(stack.pop() + " ");
      }
   }
}
```

items[]	элементы стека
n	количество элементов
items[n-1]	элемент, вставленный последним

Методы стека представляют собой простые однострочные реализации. Клиент вставляет или извлекает строки из стека в соответствии с командами из стандартного потока ввода (знак «минус» означает извлечение, а любая другая строка — вставку). Код push() для проверки заполнения стека опущен (см. текст).

```
% more tobe.txt
to be or not to - be - - that - - - is
% java ArrayStackOfStrings 5 < tobe.txt
to be not that or be
```

стандартного ввода, кроме строк, состоящих из знака «минус» (–), для которых выполняется операция pop().

Главная особенность этой реализации заключается в том, что операции push() и pop() выполняются за фиксированное время. Недостаток — в том, что клиент должен заранее оценить максимальный размер стека и всегда использует память, пропорциональную этому максимуму, что может быть неприемлемо в некоторых

ситуациях. Мы опускаем код проверки заполнения стека в `push()`, но позднее будут рассмотрены реализации, которые решают эту проблему предотвращением возможного заполнения стека до отказа (не считая предельного случая полного отсутствия свободной памяти в Java).

Связные списки

Одна из важных целей при проектировании коллекций, таких как стеки и очереди, заключается в том, что объем используемой памяти пропорционален количеству элементов в коллекции. Использование массива фиксированной длины для реализации стека в `ArrayStackOfStrings` противоречит этой цели: создание стека с заданной емкостью приводит к потенциальным потерям огромного количества памяти, если стек пуст или почти пуст. Из-за этого свойства реализация на базе массива фиксированной длины не подходит для многих приложений. А теперь мы рассмотрим фундаментальную структуру данных, называемую *связным списком*, которая может обеспечить реализации коллекций (особенно стеков и очередей), соответствующие цели из начала этого абзаца.

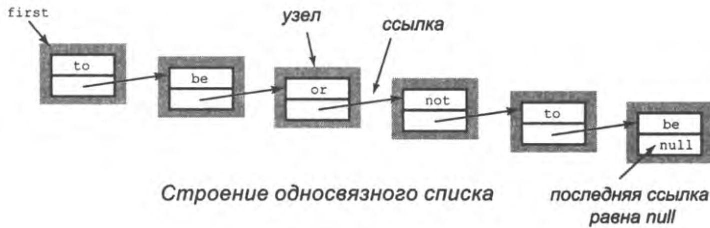
Односвязный список представляет собой последовательность *узлов*, в которой каждый узел содержит *ссылку* на следующий узел в цепочке. Обычно ссылка в последнем узле равна `null` — признак завершения списка. Узел — абстрактная сущность для хранения произвольных данных и ссылки, которая является отличительной особенностью связного списка. При трассировке кода, использующего связные списки и другие связные структуры, мы будем использовать следующие визуальные обозначения.

- Каждый узел связного списка представляется прямоугольником.
- Элемент данных и ссылка размещаются в прямоугольнике.
- Стрелки, указывающие на объекты, изображают ссылки.

Визуальное представление отражает важнейшие характеристики связных списков, а ссылки занимают в нем центральное место. Например, на диаграмме (с. 553) представлен односвязный список, содержащий серию элементов `to`, `be`, `or`, `not`, `to` и `be`.

При использовании объектно-ориентированного программирования связные списки реализуются просто. Мы определяем класс для абстракции узла, имеющий рекурсивную природу. Концепция рекурсивной структуры данных, как и концепция рекурсивной функции, на первых порах может показаться заумной.

```
class Node
{
   String item;
   Node next;
}
```



Строение односвязного списка

Объект `Node` содержит две переменные экземпляров: `String` и `Node`. Переменная экземпляра типа `String` используется для хранения любых данных, которые должны быть организованы в виде связного списка (переменных экземпляра может быть несколько) Переменная экземпляра `next` типа `Node` отражает ссылочную природу структуры данных: в ней хранится *ссылка* на следующий узел `Node` в связном списке (или `null`, если такого узла нет). Используя это рекурсивное определение, мы можем представить связный список переменной типа `Node`: эта переменная содержит либо `null`, либо ссылку на объект `Node`, поле `next` которого также содержит `null` или ссылку на следующий узел.

Чтобы подчеркнуть, что класс `Node` используется только для структурирования данных, мы не определяем никакие методы экземпляров. Как и с любым классом, объект типа `Node` может быть создан вызовом конструктора (без аргументов) `new Node()`. Результат представляет собой ссылку на новый объект `Node`, каждая из переменных экземпляров которого инициализируется значением по умолчанию `null`.

Например, чтобы построить связный список, содержащий серию значений `to`, `be` и `or`, мы создаем объект `Node` для каждого элемента:

```
Node first  = new Node();
Node second = new Node();
Node third  = new Node();
```

присваиваем переменной экземпляра `item` для каждого узла нужное значение:

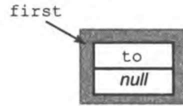
```
first.item  = "to";
second.item = "be";
third.item  = "or";
```

и задаем переменные экземпляров `next` для создания связного списка:

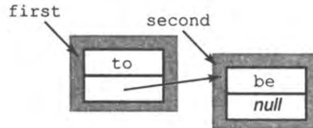
```
first.next  = second;
second.next = third;
```

В результате `first` содержит ссылку на первый элемент связного списка из трех узлов, `second` — ссылку на второй узел, а `third` — ссылку на последний узел. Код на диаграмме выполняет те же операции присваивания, но в другом порядке.

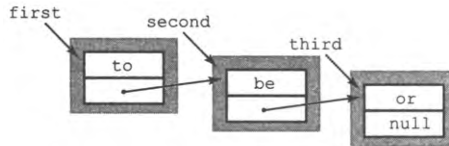
```
Node first  = new Node();
first.item  = "to";
```




```
Node second = new Node();
second.item = "be";
first.next  = second;
```




```
Node third  = new Node();
third.item  = "or";
second.next = third;
```



Построение связного списка

Связный список представляет последовательность элементов. В только что рассмотренном примере `first` представляет собой последовательность элементов `to`, `be` и `or`. Также для представления последовательности элементов можно воспользоваться массивом. Например, массив для представления той же последовательности элементов может выглядеть так:

```
String[] items = { "to", "be", "or" };
```

Эти два способа представления отличаются тем, что связаный список упрощает вставку элементов и их удаление из последовательности. Код выполнения этих двух операций будет приведен позднее.

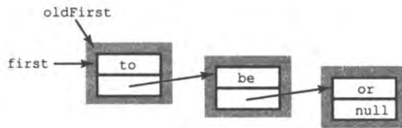
Допустим, вы хотите вставить новый узел в связный список. Проще всего выполнить вставку в начало списка. Например, чтобы вставить строку, не находящуюся в начале связного списка с первым узлом `first`, нужно сохранить `first` во временной переменной `oldFirst`, присвоить `first` новый объект `Node`, присвоить полю `item` этого объекта значение `not`, а его полю `next` — значение `oldFirst`.

Теперь предположим, что вы хотите удалить первый узел из связного списка. Эта операция выполняется еще проще: достаточно присвоить `first` значение `first.`

next. Обычно перед таким присваиванием необходимо прочитать значение элемента (присвоив его некоторой переменной), потому что после изменения значения first вы уже не сможете обратиться к данным узла, на который ссылалась эта переменная. Обычно объект Node становится потерянным, а занимаемая им память через некоторое время будет освобождена системой управления памятью Java.

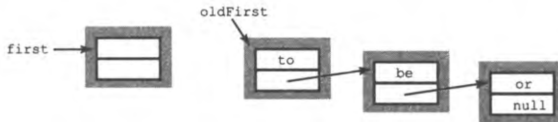
сохранение ссылки на первый узел связного списка

```
Node oldFirst = first;
```



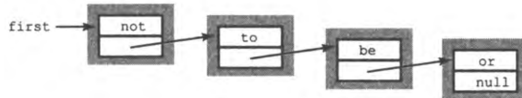
создание нового узла для начала списка

```
first = new Node();
```



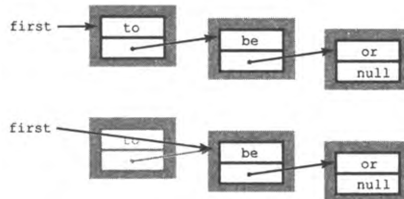
сохранение переменных экземпляров в новом узле

```
first.item = "not";
first.next = oldFirst;
```



Вставка нового узла в начало связного списка

```
first = first.next;
```



Удаление первого узла из связного списка

Код вставки и удаления узла в начале связного списка состоит из нескольких операций присваивания и, следовательно, выполняется за фиксированное время (независимо от длины списка). Если вы располагаете ссылкой на узел в произвольной

Листинг 4.3.2. Стек строк (связный список)

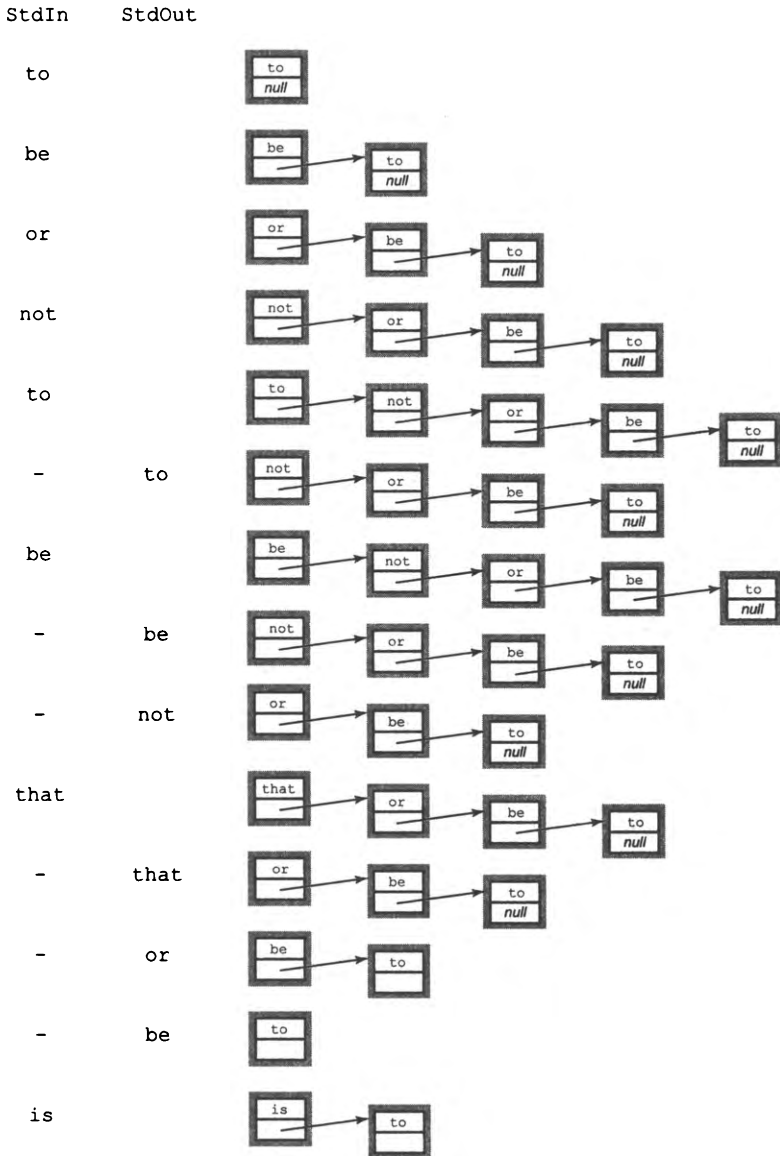
```
public class LinkedStackOfStrings
{
   private Node first;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return (first == null); }

   public void push(String item)
   {  // Вставка нового узла в начало списка.
      Node oldFirst = first;
      first = new Node();
      first.item = item;
      first.next = oldFirst;
   }

   public String pop()
   {  // Удаление первого узла из списка и возвращение элемента.
      String item = first.item;
      first = first.next;
      return item;
   }

   public static void main(String[] args)
   {
      LinkedStackOfStrings stack = new LinkedStackOfStrings();
      // Тестовый клиент приведен в листинге 4.3.1.
   }
}
```

`first`	первый узел списка

`item`	элемент стека
`next`	следующий узел списка

Эта реализация использует закрытый вложенный класс Node для представления стека в виде связного списка объектов Node. Переменная экземпляра first указывает на первый (вставленный последним) объект Node в связном списке. Переменная экземпляра next в каждом объекте Node указывает на следующий узел (значение next в последнем узле равно null). Отдельные конструкторы не нужны, потому что Java инициализирует переменные экземпляров значением null.

```
% java LinkedStackOfStrings < tobe.txt
to be not that or be
```



Трассировка тестового клиента `LinkedStackOfStrings`

позиции списка, аналогичный (но более сложный) код может использоваться для удаления узла, следующего за ним, или вставки узла после него — также за фиксированное время. Однако мы оставим эти реализации для упражнений (см. упражнение 4.3.24 и 4.3.25), потому что вставка и удаление в начале — единственные операции со связными списками, необходимые для реализации стеков.

Реализации стеков на базе связных списков

Программа `LinkedStackOfStrings` (листинг 4.3.2) использует связный список для реализации стека строк; это требует чуть большего объема кода, чем элементарное решение с массивом фиксированной длины.

В основе реализаций лежит *вложенный* класс `Node` наподобие того, который мы только что рассмотрели. Язык Java позволяет естественным образом определять и использовать внутри реализаций классов другие классы. Класс объявлен закрытым (`private`), потому что клиентам не нужно знать подробности реализации связных списков. Одна из особенностей закрытых вложенных классов заключается в том, что переменные экземпляров напрямую доступны из внешнего класса, но нигде более, поэтому объявлять переменные экземпляров `Node` открытыми или закрытыми не нужно (хотя это и не повредит).

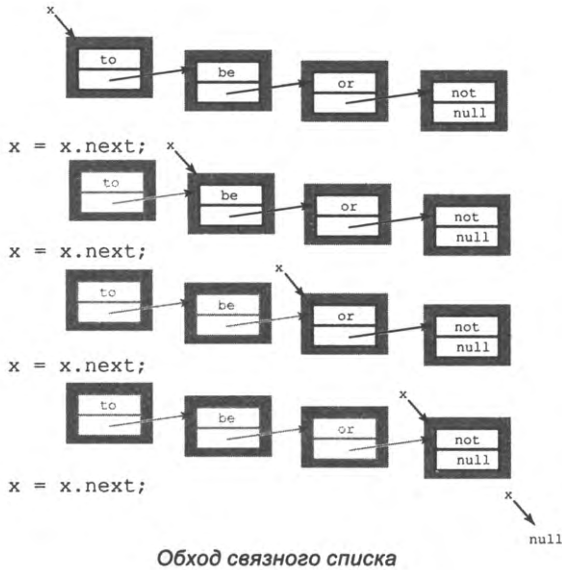
Сам класс `LinkedStackOfStrings` содержит всего одну переменную экземпляра: ссылку на связный список, представляющий стек. Этой ссылки достаточно для прямого обращения к элементу на вершине стека и косвенного обращения к остальным элементам в стеке для `push()` и `pop()`. И снова *обязательно разберитесь в реализации* — она станет прототипом для еще нескольких реализаций, использующих связные структуры, которые будут рассмотрены позднее в этой главе. Воспользуйтесь абстрактным визуальным представлением списка — это лучший способ проследить, как работает код.

Обход связного списка

Одной из самых частых операций с коллекциями является перебор элементов коллекции. Реализация метода `toString()`, неявно присутствующего в каждом API Java, упростит отладку кода стека с использованием трассировки. После класса `ArrayStackOfStrings` эта реализация выглядит уже знакомо.

```
public String toString()
{
   String s = "";
   for (int i = 0; i < n; i++)
      s += a[i] + " ";
   return s;
}
```

Это решение предназначено для использования только с малыми `n` — оно выполняется за квадратичное время, потому что каждая конкатенация строк требует линейного времени.



Обход связного списка

Сейчас нас интересует процесс просмотра каждого элемента. Существует соответствующая идиома («шаблонная» последовательность операций) для посещения всех элементов связного списка: мы инициализируем переменную цикла `x`, которая содержит ссылку на первый объект `Node` связного списка. Затем мы находим значение элемента, связного с `x`, обращаясь к `x.item`, обновляем `x` ссылкой на следующий объект `Node` связанного списка и повторяем этот процесс, пока `x` не станет равным `null` (это означает, что процесс достиг конца связного списка). Этот процесс, называемый *обходом* (или *перебором*) связного списка, компактно реализуется в методе `toString()` для `LinkedStackOfStrings`:

```
public String toString()
{
   String s = "";
   for (Node x = first; x != null; x = x.next)
      s += x.item + " ";
   return s;
}
```

При программировании с использованием связных списков эта идиома станет настолько же привычной, как и идиома сканирования элементов в массиве. В конце раздела рассматривается концепция *итератора*, которая позволяет писать клиентский код для перебора элементов коллекции, не опускаясь на более низкий уровень детализации.

Имея реализацию связных списков, мы можем писать программы, использующие большое количество стеков, не особенно беспокоясь о затратах памяти. Тот же принцип применим к любым разновидностям коллекций, поэтому связные списки

широко применяются в программировании. В самом деле, типичные реализации системы управления памятью Java основаны на ведении связных списков блоков памяти различных размеров. До широкого распространения высокоуровневых языков (таких, как Java) все подробности управления памятью и программирования с использованием связных списков были неотъемлемой частью арсенала любого программиста. В современных системах большая часть этих подробностей инкапсулируется в реализациях нескольких типов данных: стека, очереди, таблицы символов и множества (о них вы узнаете позднее в этой главе). В учебном курсе по алгоритмам и структурам данных вы изучите ряд других структур данных и приобретете опыт создания и отладки программ, работающих со связными списками. А пока вы можете сосредоточиться на понимании роли связных списков в реализации этих фундаментальных структур данных. Для стеков они исключительно важны, потому что позволяют реализовать методы `push()` и `pop()` с фиксированным временем исполнения и с небольшими дополнительными затратами памяти (для ссылок)[1].

Массивы с изменяющимся размером

Рассмотрим альтернативный подход к обеспечению произвольного изменения размеров в структуре данных, которая является достойным конкурентом связным спискам. Как и в случае со связными списками, мы представляем ее сейчас, потому что это решение легко понять в контексте реализации стека, и его важно знать при решении проблем реализации типов данных более сложных, чем стеки.

Идея заключается в том, чтобы модифицировать реализацию массива (листинг 4.3.1) для динамического изменения длины массива `items[]`; массив должен быть достаточно большим, чтобы в нем помещались все элементы, но не настолько большим, чтобы это приводило к излишним затратам памяти. Как выясняется, достичь этих целей на удивление несложно, и мы сделаем это в программе `ResizingArrayStackOfStrings` (листинг 4.3.3).

Сначала в методе `push()` мы проверяем, не пора ли расширять массив, а именно есть ли в массиве место для нового элемента `item`, сравнивая размер стека `n` с длиной массива `items.length`. Если место еще остается, мы просто вставляем новый элемент в массив командой `items[n++] = item`, как и прежде; если нет — длина массива *удваивается*, для чего мы создаем новый массив вдвое большей длины, копируем элементы стека в новый массив и изменяем переменную экземпляра `items[]`, чтобы она указывала на новый массив.

---

[1] Эти служебные затраты относительно невелики, только если полезные данные в элементах списка имеют намного больший размер, чем ссылка. — *Примеч. науч. ред.*

Листинг 4.3.3. Стек строк (изменение размера массива)

```
public class ResizingArrayStackOfStrings
{
   private String[] items = new String[1];
   private int n = 0;

   public boolean isEmpty()
   {  return (n == 0);  }

   private void resize(int capacity)
   {  // Перемещение стека в новый массив заданной емкости.
      String[] temp = new String[capacity];
      for (int i = 0; i < n; i++)
         temp[i] = items[i];
      items = temp;
   }

   public void push(String item)
   {  // Вставка элемента в стек.
      if (n == items.length) resize(2*items.length);
      items[n++] = item;
   }

   public String pop()
   {  // Удаление первого узла из списка и возвращение элемента.
      String item = items[--n];
      items[n] = null;  // Предотвращаем "холостое хранение" (см. далее
в тексте).
      if (n > 0 && n == items.length/4) resize(items.length/2);
      return item;
   }

   public static void main(String[] args)
   {
      // Тестовый клиент приведен в листинге 4.3.1.
   }
}
```

item	элемент стека
n	количество элементов в стеке

Реализация обеспечивает поддержку стеков произвольного размера без чрезмерных затрат памяти. Длина массива удваивается при заполнении и сокращается вдвое, чтобы массив всегда был заполнен по крайней мере на четверть. В среднем все операции выполняются за фиксированное время (см. текст).

```
% java ResizingArrayStackOfStrings < tobe.txt
to be not that or be
```

StdIn	StdOutn	n	items. length	items[] 0	1	2	3	4	5	6	7
		0	1	null							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	null				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2	2	to	is						

Трассировка тестового клиента ResizingArrayStackOfStrings

Аналогичным образом в `pop()` мы сначала проверяем, не слишком ли велик массив, и если условие выполняется, сокращаем его длину вдвое. Если задуматься, становится ясно, что размер стека нужно сравнивать с четвертью длины массива. Тогда после уменьшения массива вдвое он будет заполнен примерно наполовину и сможет выполнить достаточное количество операций `push()` и `pop()` без повторного изменения длины массива. Эта характеристика важна: например, если бы размер массива уменьшался вдвое, когда размер стека достигает половины длины массива, то полученный массив оказался бы полон и его пришлось бы увеличивать вдвое для `push()`. Так возникает риск высокозатратного цикла удвоений и сокращений.

Амортизационный анализ

Стратегия увеличения и сокращения в 2 раза представляет разумный компромисс между неэффективным расходованием памяти (если выбрать слишком большую длину массива, остается много пустых ячеек) и затратами времени (на изменение структуры массива после каждой вставки). Конкретная стратегия из `ResizingArrayStackOfStrings` гарантирует, что стек никогда не переполнится и никогда не будет заполнен менее чем на одну четверть (кроме пустого стека, для которого длина массива равна 1). Если вы хорошо разбираетесь в математике, вы можете проверить этот факт посредством математической индукции (см. упражнение 4.3.18). И что еще важнее, мы можем доказать, что затраты на расширение и сокращение массива вдвое всегда поглощаются (с точностью до постоянного множителя) в затратах на другие операции со стеком. И снова оставим подробности для математически одаренных читателей, но идея проста: когда `push()` удваивает длину

массива до n, обработка начинается с $n/2$ элементов в стеке, так что длина массива не может удвоиться снова, пока клиент не сделает минимум $n/2$ дополнительных вызовов `push()` (больше при промежуточных вызовах `pop()`). Если усреднить затраты операции `push()`, приводящей к удвоению, с затратами этих $n/2$ операций `push()`, мы получим константу. Иначе говоря, в `ResizingArrayStackOfStrings` *общие затраты всех стековых операций, разделенные на количество операций, ограничиваются константой.* Это утверждение слабее утверждения о том, что каждая операция выполняется за фиксированное время, но во многих приложениях оно приводит к тем же последствиям (например, если нас интересует прежде всего общее время выполнения). Анализ такого рода называется ***амортизационным анализом***, а изменение размера структуры данных массива — классический пример его применения.

Потерянные объекты

Политика уборки мусора в Java основана на освобождении памяти, связанной с любым объектом, который стал недоступным. В реализации `pop()` нашей исходной реализации `ArrayStackOfStrings` ссылка на извлеченный элемент остается в массиве. Этот элемент становится потерянным объектом — мы никогда не будем использовать его из класса либо из-за того, что стек уменьшится, либо из-за замены другой ссылкой при увеличении стека, но уборщик мусора Java об этом знать не может. Даже если клиент больше не будет работать с элементом, ссылка в массиве может поддерживать его в жизнеспособном состоянии. Это условие (хранение ссылки на элемент, который не будет использоваться) называется ***холостым хранением***; не путайте его с утечкой памяти (когда ссылка на элемент вообще отсутствует в системе управления памятью). В данном случае холостое хранение легко предотвращается. Реализация `pop()` из `ResizingArrayStackOfStrings` присваивает `null` элементу массива, соответствующему извлеченному элементу; при этом неиспользуемая ссылка заменяется, а система сможет освободить память, связанную с извлеченным элементом, когда клиент завершит работу с ним[1].

Реализация с изменяющимся размером массива (как и реализация на базе связного списка) позволяет писать программы, использующие стеки, не беспокоясь о затратах памяти. И снова один принцип распространяется на коллекции любого типа. Для некоторых типов данных, более сложных, чем стеки, массивы с изменяющимся размером оказываются предпочтительнее связных списков из-за возможности обращаться к любому элементу массива за фиксированное время (посредством индексирования), что может быть критично для реализации некоторых операций (например, см. программу `RandomQueue` в упражнении 4.3.37). Как и в случае со связными списками, реализацию массива переменной длины лучше локалиовать на уровне фундаментальных структур данных и не беспокоиться о его использовании в клиентском коде.

---

[1] Очевидно, это актуально, только если элемент массива содержит не сами данные (примитивного типа), а *ссылку* на объект; здесь это объект `String`. — *Примеч. науч. ред.*

Параметризованные типы данных

Итак, мы разработали реализации, которые позволяют создавать стеки с элементами одного конкретного типа (`String`). Но при разработке клиентских программ понадобятся реализации для коллекций с элементами других типов, не обязательно `String`. Коммерческая система обработки транзакций может поддерживать коллекции заказчиков, счетов, продавцов и транзакций; университетская система построения учебных расписаний может поддерживать коллекции классов, студентов и аудиторий; приложение-проигрыватель может поддерживать коллекции песен, исполнителей и альбомов; научная программа может поддерживать коллекции значений типа `double` или `int`. В любой написанной вами программе у вас вполне может возникнуть необходимость поддержания коллекций любого типа данных, которые вам потребуется создать. Как это сделать? После рассмотрения двух простых решений (и их недостатков), использующих рассмотренные ранее конструкции языка Java, будет представлена более функциональная конструкция, которая позволяет нормально решить эту проблему.

Создание новой коллекции для каждого типа данных элементов

В дополнение к `StackOfStrings` мы можем создать классы `StackOfInts`, `StackOfCustomers`, `StackOfStudents` и т. д. Такой подход потребует дублирования кода для каждого типа данных, а это нарушает один из канонов программирования, гласящий, что код следует по возможности использовать повторно (без копирования). Для каждого типа данных, который должен храниться в стеке, будет необходим отдельный класс, и сопровождение кода превращается в сущий кошмар: каждый раз, когда вам понадобится внести изменение, это придется делать в *каждой* версии кода. Тем не менее это решение применяется достаточно часто, потому что во многих языках программирования (включая ранние версии Java) нет более эффективного решения проблемы. Способность преодолеть этот барьер — признак опытного программиста и мощной среды программирования. Сможете ли вы реализовать стеки строк, стеки целых чисел, стеки данных вообще любого типа, используя всего один класс?

Использование коллекций с элементами Object

Можно разработать стек, все элементы которого относятся к типу `Object`. Благодаря наследованию в такой стек можно вставить объект любого типа (если вы захотите вставить объект типа `Apple`, это можно сделать, потому что `Apple` является субклассом `Object`, как и все остальные классы). Извлекая объект из стека, его необходимо преобразовать обратно к фактическому типу (в стеке хранятся только объекты `Object`, а наш код работает с объектами типа `Apple`). Итак, если мы создадим класс `StackOfObjects`, заменив все вхождения `String` на `Object` в одной из наших реализаций `*StackOfStrings`, мы сможем писать код следующего вида:

```
StackOfObjects stack = new StackOfObjects();
Apple a = new Apple();
stack.push(a);
...
a = (Apple) (stack.pop());
```

В результате достигается исходная цель: создать один класс для создания и обработки стеков объектов произвольного типа. Тем не менее такое решение нежелательно, потому что оно создает риск коварных ошибок в клиентском коде, которые невозможно обнаружить во время компиляции. Например, ничто не мешает программисту поместить в тот же стек объекты других типов, как в следующем примере:

```
ObjectStack stack = new ObjectStack();
Apple  a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b);
a = (Apple)  (stack.pop()); // Инициирует ClassCastException.
b = (Orange) (stack.pop());
```

Такое преобразование типов *предполагает*, что клиенты будут преобразовывать объекты, извлеченные из стека, к правильному типу, тем самым лишаясь защиты, предоставляемой системой типизации Java. В частности, программисты используют типизацию как раз для защиты от ошибок, возникающих из-за таких неявных предположений. Этот код невозможно проверить на правильность использования типов во время компиляции: в каком-нибудь сложном фрагменте кода может содержаться некорректное преобразование, которое останется незамеченным до возникновения особых обстоятельств уже на стадии выполнения. Таких ошибок необходимо избегать любой ценой, потому что они проявляются уже после того, как реализация будет передана пользователю, и исправить их уже не удастся.

Обобщения Java

Для решения этой задачи в Java существует специальный механизм: так называемые *обобщенные типы* (или *обобщения*). С обобщенными типами можно строить коллекции объектов, тип которых задается в клиентском коде. Главным преимуществом такого решения является возможность выявления ошибок несоответствия типов во время компиляции (в процессе разработки программного продукта), а не во время выполнения (когда программный продукт используется клиентом). С концептуальной точки зрения обобщения выглядят немного запутанно (они настолько глубоко влияют на язык программирования, что их поддержка не была включена в ранние версии Java), но для нашего применения в текущем контексте потребуется совсем небольшой объем дополнительного синтаксиса Java, и разобраться в них будет несложно. Мы присваиваем обобщенному классу имя `Stack` и выбираем обобщенное имя `Item` для типа объектов в стеке (можно использовать любое имя). Код `Stack` (листинг 4.3.4) идентичен коду `LinkedStackOfStrings` (модификатор `Linked` опускается, потому что у нас есть хорошая реализация для клиентов, которых представление не интересует), не считая того, что каждое вхождение `String` заменяется на `Item`, а объявление класса в первой строке кода выглядит так:

```
public class Stack<Item>
```

Листинг 4.3.4. Обобщенный стек

```
public class Stack<Item>
{
   private Node first;

   private class Node
   {
      Item item;
      Node next;
   }

   public boolean isEmpty()
   {  return (first == null); }

   public void push(Item item)
   {  // Вставка item в стек.
      Node oldFirst = first;
      first = new Node();
      first.item = item;
      first.next = oldFirst;
   }

   public Item pop()
   {  // Извлечение и возвращение последнего вставленного элемента.
      Item item = first.item;
      first = first.next;
      return item;
   }

   public static void main(String[] args)
   {
      Stack<String> stack = new Stack<String>();
      // Тестовый клиент приведен в листинге 4.3.1.
   }
}
```

`first`	**первый узел списка**
`item`	**элемент стека**
`next`	**следующий узел списка**

Этот код почти идентичен коду из листинга 4.3.2, но его стоит повторить, потому что он показывает, как легко использовать обобщения для того, чтобы клиенты могли создавать коллекции любого типа данных. Ключевое слово Item в этом коде — параметр-тип, заменитель для имени фактического типа, предоставленного клиентом.

```
% java Stack < tobe.txt
to be not that or be
```

Имя `Item` определяет *параметр-тип* — символическое обозначение некоторого фактического типа, который будет задан клиентским кодом, использующим стек. При реализации `Stack` фактический тип `Item` неизвестен, но клиент может использовать стек для любого типа данных, в том числе и для того, который был определен намного позднее разработки нашей реализации. В клиентском коде при создании стека указывается *аргумент-тип* `Apple`:

```
Stack<Apple> stack = new Stack<Apple>();
Apple a = new Apple();
...
stack.push(a);
```

При попытке занести в стек объект недопустимого типа:

```
Stack<Apple> stack = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b);     // Ошибка компиляции.
```

вы получите сообщение об *ошибке компиляции*:

```
push(Apple) in Stack<Apple> cannot be applied to (Orange)
```

Как видно, в нашей реализации `Stack` Java может использовать параметр-тип `Item` для проверки ошибок несоответствия типов: хотя фактический тип еще неизвестен, переменным типа `Item` должны присваиваться значения типа `Item` и т. д.

Автоупаковка

У обобщенного кода (наподобие того, что приведен в листинге 4.3.4) существует одна небольшое затруднение: параметр-тип обозначает ссылочный тип. Как использовать код для примитивных типов, таких как `int` и `double`? Механизм языка Java, называемый *автоупаковкой*, позволяет повторно использовать обобщенный код и с примитивными типами. В языке Java существуют встроенные объектные типы, называемые *обертками*, по одному для каждого из примитивных типов: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` и `Short` представляют `boolean`, `byte`, `char`, `double`, `float`, `int`, `long` и `short` соответственно. Java автоматически выполняет преобразование между этими ссылочными типами и соответствующими примитивными типами (в командах присваивания, аргументах методов и арифметических/логических выражениях), чтобы мы могли писать код следующего вида:

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);      // Автоупаковка (int -> Integer).
int a = stack.pop(); // Распаковка   (Integer -> int).
```

В этом примере Java автоматически преобразует (упаковывает) примитивное значение 17 к типу `Integer` при передаче методу `push()`. Метод `pop()` возвращает тип `Integer`, который Java преобразует (распаковывает) в значение `int`, прежде чем присваивать его переменной `a`. Данная возможность удобна при написании кода, но она сопряжена со значительными затратами ресурсов, которые могут повлиять на быстродействие. Поэтому, возможно, в каких-то ситуациях, критичных по быстродействию, без такого класса, как `StackOfInts`, все же не обойтись.

Обобщения предоставляют именно то, что нам нужно: возможность повторного использования кода и одновременно безопасность типов. Тщательное изучение кода `Stack` (листинг 4.3.4) и уверенность в том, что вы понимаете каждую строку

кода, непременно окупятся в будущем, так как параметризация типов — важный высокоуровневый программный инструмент, имеющий хорошую поддержку в Java. И вовсе не обязательно быть экспертом, чтобы пользоваться этим мощным инструментом.

Примеры использования стека

Стеки играют важную роль в обработке данных. Изучая операционные системы, языки программирования и другие нетривиальные темы вычислительных технологий, вы узнаете, что стеки не только явно используются во многих приложениях, но и служат основой для выполнения программ, написанных на многих языках высокого уровня, включая Java и Python.

Арифметические выражения

В ряде первых программ, рассмотренных в главе 1, вычислялись значения арифметических выражений следующего вида:

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

Если умножить 4 на 5, прибавить 3 к 2, перемножить результаты, а затем прибавить 1, вы получите значение 101. Но как Java выполняет эти вычисления? Даже не углубляясь в подробности внутреннего устройства Java, мы можем воспользоваться этим механизмом — просто написать программу, которая принимает на вход это выражение и выдает на выход результат вычислений. Для простоты начнем со следующего явного рекурсивного определения: *арифметическое выражение* представляет собой число или левую круглую скобку, за которыми следует арифметическое выражение, затем оператор, затем другое арифметическое выражение и далее правая круглая скобка. Для простоты это определение описывает арифметические выражения с полным набором круглых скобок, которые однозначно указывают, к каким операндам относятся те или иные операторы, — вероятно, вам чаще встречались выражения вида `1 + 2*3`, в которых используются правила приоритета вместо круглых скобок. В принципе, для определения правил приоритета можно использовать те же базовые механизмы, но пока мы обойдем эти сложности. Для простоты мы ограничимся поддержкой знакомых двухместных операторов `*`, `+` и `-`, а также оператора квадратного корня `sqrt`, получающего всего один аргумент. Мы можем легко обеспечить поддержку большего количества операторов для знакомых математических выражений, включая деление, тригонометрические и экспоненциальные функции. Сейчас нас интересует то, как интерпретировать строку с круглыми скобками, операторами и числами для выполнения в правильном порядке низкоуровневых арифметических операторов, поддерживаемых любым компьютером.

Листинг 4.3.5. Вычисление значения выражения

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> values = new Stack<Double>();
      while (!StdIn.isEmpty())
      {  // Прочитать лексему; занести в стек, если это оператор.
         String token = StdIn.readString();
         if      (token.equals("("))                ;
         else if (token.equals("+"))    ops.push(token);
         else if (token.equals("-"))    ops.push(token);
         else if (token.equals("*"))    ops.push(token);
         else if (token.equals("sqrt")) ops.push(token);
         else if (token.equals(")"))
         {  // Извлечь, вычислить и занести результат в стек,
            // для лексемы ")".
            String op = ops.pop();
            double v = values.pop();
            if      (op.equals("+"))    v = values.pop() + v;
            else if (op.equals("-"))    v = values.pop() - v;
            else if (op.equals("*"))    v = values.pop() * v;
            else if (op.equals("sqrt")) v = Math.sqrt(v);
            values.push(v);
         }  // Не оператор и не скобка: занести в стек значение double.
         else values.push(Double.parseDouble(token));
      }
      StdOut.println(values.pop());
   }
}
```

`ops`	стек операторов
`values`	стек операндов
`token`	текущая лексема
`v`	текущее значение

Программа, являющаяся клиентом для Stack, читает числовое выражение с полным набором круглых скобок из стандартного ввода, использует алгоритм Дейкстры для вычисления его результата и выводит полученное число в стандартный вывод. Эти вычисления демонстрируют важнейший вычислительный процесс: интерпретацию строки как программы и выполнение этой программы для вычисления результата. В сущности, выполнение программы Java представляет собой не что иное, как более сложную разновидность того же процесса.

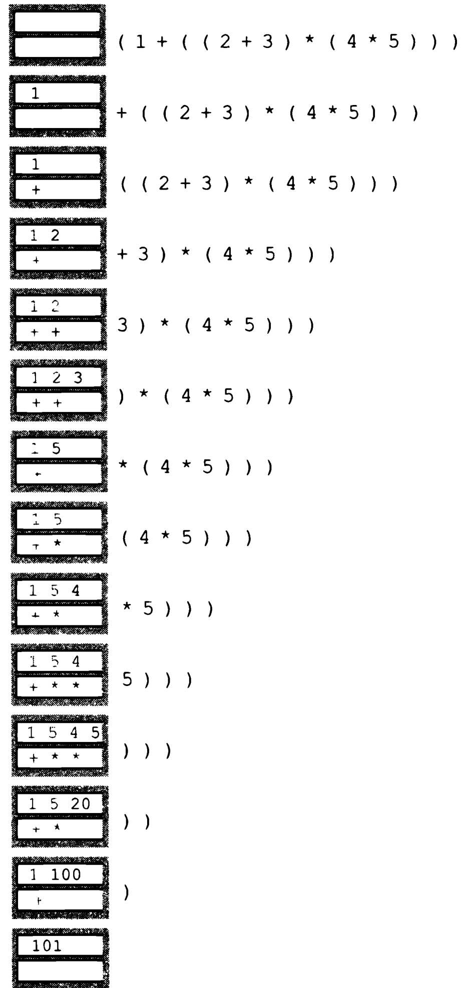
```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

```
% java Evaluate
( ( 1 + sqrt ( 5.0 ) ) * 0.5 )
1.618033988749895
```

Вычисление результатов арифметических выражений

Как именно преобразовать арифметическое выражение — строку символов — в представляемое ею значение? На удивление простой алгоритм, разработанный Эдсгером Дейкстрой в 1960-х годах, использует два стека: для операндов и для операторов. Выражение состоит из круглых скобок, операторов и операндов (чисел). Перемещаясь слева направо и последовательно рассматривая эти сущности, мы выполняем операции со стеками для одного из четырех возможных случаев.

- Операнды вставляются в стек операндов.
- Операторы вставляются в стек операторов.
- Левые круглые скобки игнорируются.
- При обнаружении правой круглой скобки оператор извлекается из стека, из стека операндов извлекается соответствующее число операндов, а затем в тот же стек операндов заносится результат применения этого оператора к операндам.



Трассировка процесса вычисления выражения (листинг 4.3.5)

После того как будет обработана последняя правая круглая скобка, в стеке остается одно значение, которое и является значением этого выражения. Алгоритм Дейкстры на первый взгляд выглядит таинственно, но вы можете легко убедиться в том, что он вычисляет правильное значение: каждый раз, когда алгоритм встречает подвыражение из двух операндов, разделенных оператором и заключенных в круглые скобки, в стеке операндов остается результат выполнения этой операции с этими операндами. Результат будет таким же, как если бы это значение стояло во входных данных вместо подвыражения, поэтому при замене подвыражения этим значением вы получите выражение, которое даст тот же результат. Этот прием можно применять снова и снова, пока мы не придем к одному значению. Например, этот алгоритм вычисляет один результат для всех следующих выражений:

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
( 1 + ( 5 * ( 4 * 5 ) ) )
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Программа `Evaluate` (листинг 4.3.5) реализует этот алгоритм. Код, по сути, является простым примером *интерпретатора*: программы, которая выполняет другую программу (в данном случае арифметическое выражение) по одному шагу или по одной строке. Напомним, что *компилятор* переводит программу с высокоуровневого языка на низкоуровневый язык, способный выполнить задачу. Преобразование компилятора — более сложный процесс, чем пошаговое преобразование, используемое интерпретатором, но оно основано на тех же принципах. Компилятор Java переводит код, написанный на языке программирования Java, в байт-код Java. Изначально язык Java был интерпретируемым. Однако теперь Java включает компилятор, который преобразует арифметические выражения (а на более общем уровне — программы Java) в низкоуровневый код *виртуальной машины* Java (воображаемая машина, легко моделируемая на реальном компьютере).

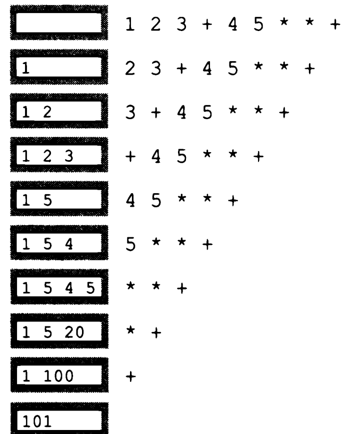
Стековые языки программирования

Интересно, что алгоритм Дейкстры также вычисляет такое же значение, как в нашем примере, для следующего выражения:

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Иначе говоря, каждый оператор может размещаться после двух операндов, а не между ними. В таком выражении каждая правая круглая скобка следует сразу же за оператором, поэтому скобки можем игнорировать и записать выражения в следующем виде:

```
1 2 3 + 4 5 * * +
```



Трассировка вычисления постфиксного выражения

Эта запись называется *обратной польской записью* (или *постфиксной* записью.) Для вычисления постфиксного выражения достаточно одного стека (см. упражнение 4.3.15). Перемещаясь слева направо, мы последовательно обрабатываем компоненты выражения и выполняем операции всего в двух возможных случаях.

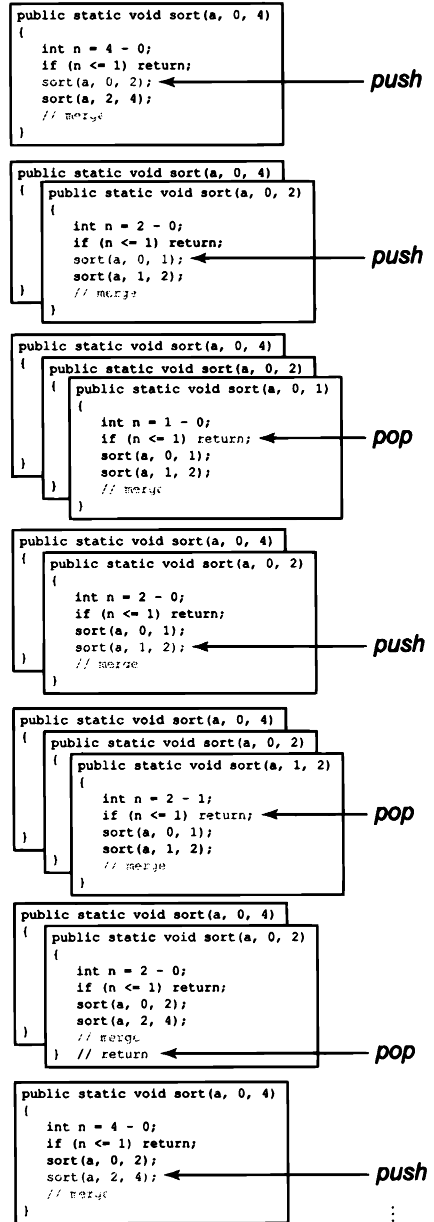
- Операнды заносятся в стек.
- При обнаружении оператора из стека извлекается соответствующее количество операндов, а в стек заносится результат применения оператора к этим операндам.

И снова этот процесс оставляет в стеке одно значение, которое является значением выражения. Это представление выглядит настолько просто, что некоторые языки

программирования, такие как Forth (научный язык программирования) и PostScript (язык описания страниц, используемый многими принтерами), используют стеки в качестве основных структур управления вычислениями. Например, строка 1 2 3 + 4 5 * * + является действительной программой и в Forth, и в PostScript и оставляет в стеке выполнения значение 101. Поклонники этих и других похожих языков программирования предпочитают их потому, что они упрощают многие типы вычислений. Более того, сама виртуальная машина Java работает на базе стека.

Абстракция вызова функции

Большинство программ неявно использует стеки, поскольку стеки предоставляют естественный способ реализации вызовов функций: в любой момент во время выполнения функции ее состояние полностью определяется значениями всех ее переменных и указателем на следующую выполняемую инструкцию. Одна из фундаментальных характеристик вычислительной среды заключается в том, что каждое вычисление полностью определяется его состоянием (и значениями его входных данных). В частности, система может приостановить вычисления, сохраняя их текущее состояние, а затем возобновить, восстановив сохраненное состояние. В учебном курсе по операционным системам вы изучите этот процесс во всех подробностях, потому что он критически важен для многих аспектов поведения компьютеров, которые мы воспринимаем как нечто само собой разумеющееся (например, переключение с одного приложения на другое сводится к простому сохранению и восстановлению состояния). Естественный способ реализовать абстракцию вызова функции основан на использовании стека. Чтобы вызвать функцию, занесите текущее состояние в стек. Чтобы вернуться из функции, извлеките состояние из стека для восстановления всех значений переменных на момент, предшествовавший вызову,



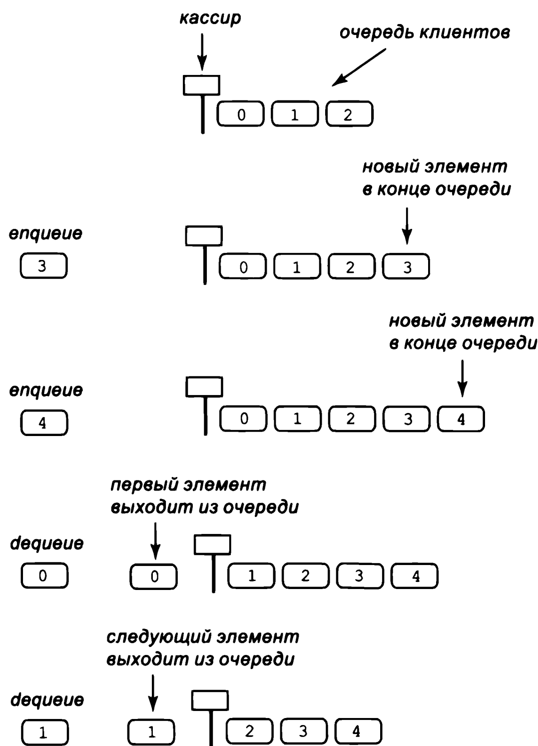
Использование стека для поддержки вызовов функций

замените возвращаемое значение функции (если оно есть) в выражении, содержащем вызов функции (если он есть), и продолжите выполнение со следующей выполняемой инструкции (местоположение которой было сохранено как часть состояния вычислений). Этот механизм работает каждый раз, когда функции вызывают друг друга, даже при рекурсивном вызове. В самом деле, если внимательно задуматься над этим процессом, вы увидите, что он, по сути, *не отличается* от процесса, подробно описанного для вычисления выражений. Таким образом, программу тоже можно рассматривать как сложное выражение.

Стек принадлежит к числу фундаментальных абстракций в теории вычислений. Стеки использовались для вычисления выражений, реализации абстракции вызова функции и для других целей с первых дней вычислительной техники. Еще одно применение стеков (обход дерева) будет рассмотрено в разделе 4.4. В явном виде стеки широко используются во многих областях информатики, включая проектирование алгоритмов, операционные системы, компиляторы и т. д.

Очереди FIFO

Очередь FIFO (или просто *очередь*) представляет собой коллекцию, работающую по принципу «первым вошел, первым вышел».



Типичная очередь FIFO

Политика выполнения задач в порядке их поступления часто встречается в повседневной жизни — от ожидания в очереди в театре или в очереди машин на дорожных пропускных пунктах до задач, ожидающих выполнения в приложении на вашем компьютере.

Базовым принципом любой стратегии обслуживания является субъективная справедливость. Что первым приходит в голову в отношении справедливости обслуживания? Вероятно, что клиент, который ожидал дольше остальных, должен обслуживаться в первую очередь. Собственно, именно в этом заключается суть дисциплины FIFO, поэтому очереди занимают центральное место во многих приложениях. Модель очереди естественным образом подходит для многих повседневных ситуаций, а ее свойства были подробно изучены еще до появления компьютеров.

Как обычно, начнем с формирования API. По традиции операции постановки в очередь присваивается имя `enqueue`, а операции извлечения из очереди — имя `dequeue`:

public class Queue<Item>	
`Queue()`	создает пустую очередь
`boolean isEmpty()`	очередь пуста?
`void enqueue(Item item)`	вставляет элемент в очередь
`Item dequeue()`	возвращает и извлекает элемент, который был вставлен последним
`int size()`	количество элементов в стеке

API обобщенной очереди FIFO

Как следует из API, в реализациях используются обобщения, чтобы мы могли писать клиентский код, позволяющий безопасно строить и использовать очереди со ссылками любого типа. Мы включили в API метод `size()`, хотя для стеков такого метода не было, потому что при использовании очередей часто требуется знать количество элементов в очереди, тогда как для стеков такой необходимости нет (см. листинг 4.3.8 и упражнение 4.3.11).

Применяя наши знания о стеках, мы можем использовать связные списки или массивы с изменяющимся размером для разработки реализаций, у которых операции выполняются за фиксированное время, а объем памяти, связанной с очередью, увеличивается и уменьшается вместе с количеством элементов в ней. Как и в случае со стеками, каждая из этих реализаций представляет классическое упражнение для программистов. Попробуйте подумать, как бы вы достигли каждую из этих целей в своей реализации, прежде чем читать дальше.

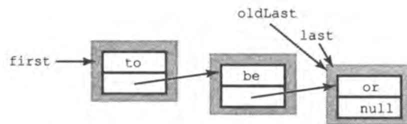
Реализация на базе связного списка

Чтобы реализовать очередь на базе связного списка, мы будем хранить элементы в порядке их поступления (обратном тому, который использовался в `Stack`). Реализация `dequeue()` не отличается от реализации `pop()` в `Stack` (сохранить элемент

из первого узла связного списка, удалить узел из очереди, вернуть сохраненный элемент). Однако с реализацией `enqueue()` дело обстоит сложнее: как добавить узел в конец связного списка? Для этого нужна ссылка на последний узел списка, потому что ссылку этого узла необходимо изменить — она должна указывать на новый узел, содержащий вставляемый элемент. В `Stack` единственная переменная экземпляра содержит ссылку на первый узел связного списка, и если у вас нет другой информации, остается только перебрать все узлы списка, чтобы добраться до его конца. Это решение неэффективно для длинных списков. Разумная альтернатива — введение второй переменной экземпляра, которая всегда содержит ссылку на последний узел связного списка. К добавлению новой переменной экземпляра, значением которой тоже нужно управлять, не стоит относиться легко — особенно в коде связного списка, потому что каждый метод, изменяющий список, должен проверить, нужно ли изменить эту переменную (и внести необходимые изменения). Например, удаление первого узла связного списка может потребовать изменения ссылки и на последний узел — ведь если в списке остается всего один узел, он одновременно является и первым, и последним! (Из-за таких тонкостей код, работающий со связными списками, имеет репутацию трудного для отладки.) Программа `Queue` (листинг 4.3.6) представляет собой реализацию очереди FIFO на базе связного списка, которая обладает теми же свойствами быстродействия, что и `Stack`: все методы выполняются за фиксированное время, а затраты памяти пропорциональны размеру очереди.

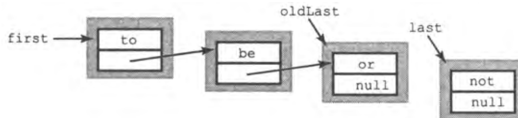
сохранение ссылки на последний узел

```
Node oldLast = last;
```



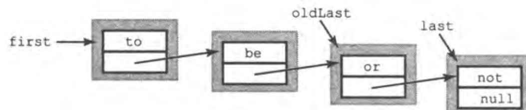
создание нового узла, который должен стать конечным

```
Node last = new Node();
last.item = "not";
```



связывание нового узла с концом списка

```
oldLast.next = last;
```



Вставка нового узла в конец связного списка

Листинг 4.3.6. Обобщенная очередь FIFO (связный список)

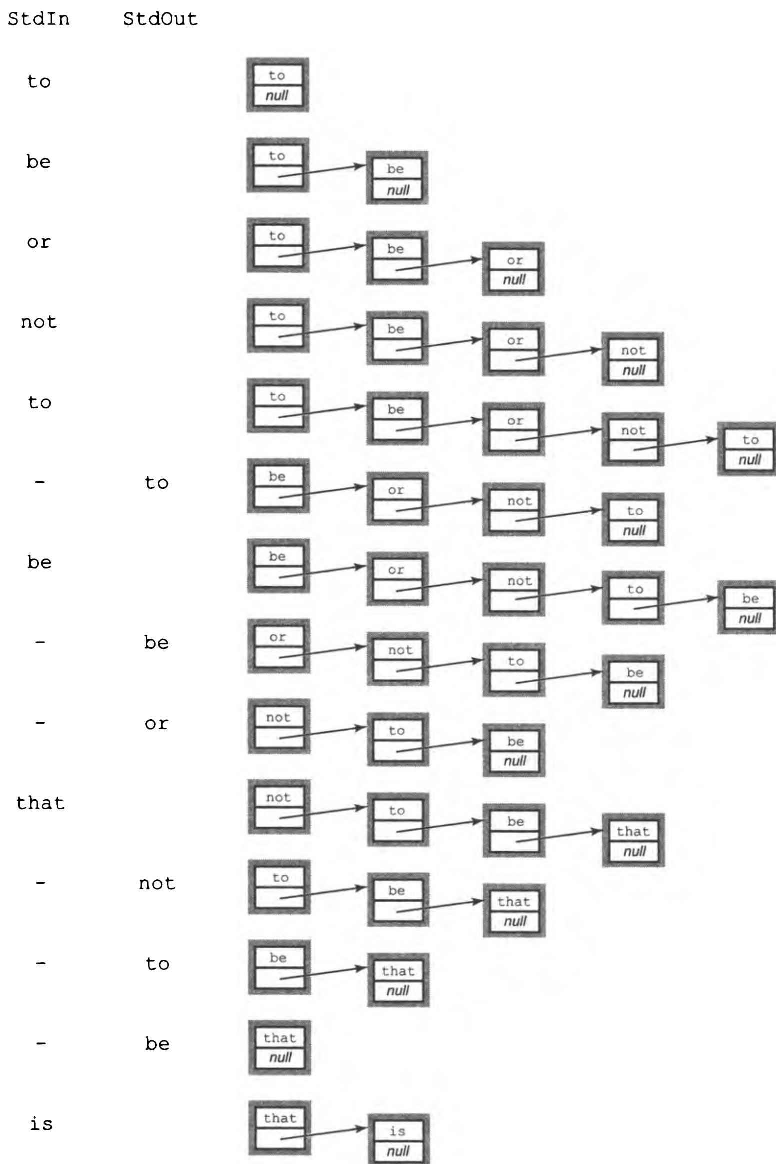
```
public class Queue<Item>
{
   private Node first;
   private Node last;

   private class Node
   {
      Item item;
      Node next;
   }

   public boolean isEmpty()
   {  return (first == null);  }

   public void enqueue(Item item)
   {  // Вставка нового узла в конец списка.
      Node oldLast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldLast.next = last;
   }

   public Item dequeue()
   {  // Метод извлекает первый узел из списка и возвращает элемент.
      Item item = first.item;
      first = first.next;
      if (isEmpty()) last = null;
      return item;
   }

   public static void main(String[] args)
   {  // Тестовый клиент похож на код листинга 4.3.2.
      Queue<String> queue = new Queue<String>();
   }
}
```

`first`	первый узел в списке
`last`	последний узел в списке

`item`	элемент очереди
`next`	следующий узел в списке

Реализация очень похожа на реализацию стека на базе связного списка (листинг 4.3.2): метод dequeue() почти идентичен pop(), но enqueue() присоединяет новый узел к концу списка, а не в его начало, как push(). Для этого создается переменная экземпляра last, которая содержит ссылку на последний узел списка. Реализация метода size() остается читателю для самостоятельной работы (см. упражнение 4.3.11).

```
% java Queue < tobe.txt
to be or not to be
```



Трассировка тестового клиента Queue (см. листинг 4.3.6)

Реализации на базе массива

Также возможно разработать реализации очередей FIFO на базе массивов, обладающие теми же характеристиками быстродействия, что и реализации стеков из `ArrayStackOfStrings` (листинг 4.3.1) и `ResizingArrayStackOfStrings` (листинг 4.3.3). Эти реализации представляют собой достойные упражнения для программиста, и мы рекомендуем вам заняться ими самостоятельно (см. упражнение 4.3.19).

Случайные очереди

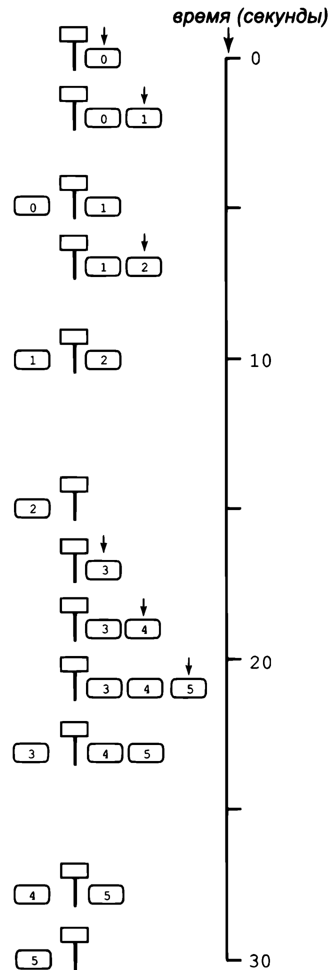
Несмотря на широкое применение дисциплин FIFO и LIFO, в них нет ничего священного. Ничто не мешает вам рассматривать другие правила извлечения элементов. Один из важнейших примеров — структура данных, в котором метод `dequeue()` извлекает и возвращает случайный элемент (выборка без возмещения), а метод `sample()` возвращает случайный элемент без его извлечения из очереди (выборка с возмещением). Мы будем называть эту структуру данных случайной очередью (см. упражнение 4.3.37).

API стека, очереди и случайной очереди почти идентичны — отличаются только имена классов и методов (которые выбираются произвольно, самим программистом). Истинные различия между этими структурами данных проявляются в семантике операции извлечения: какой именно элемент должен быть извлечен из коллекции? Различия между стеками и очередями отражены в описаниях того, что они делают, на естественном языке. Эти различия сродни различиям между `Math.sin(x)` и `Math.log(x)`, но, возможно, вы захотите сформулировать их в формальных описаниях стеков и очередей (по аналогии с математическими описаниями функций синуса и логарифма). Но точно описать, что же мы подразумеваем под дисциплинами FIFO, LIFO или случайной выборки, не так просто. Прежде всего, какой язык вы будете использовать для такого описания — английский, Java, язык математической логики? Задача описания поведения программы называется *задачей спецификации*, и она непосредственно ведет к некоторым нетривиальным аспектам теории программирования. В частности, мы уделяем столько внимания ясности и лаконичности кода еще и потому, что сам код может служить спецификацией для простых структур данных, таких как стеки, очереди и случайные очереди.

Практические применения очередей

В прошлом веке было доказано, что очереди FIFO оказываются адекватными моделями, применимыми в самых разных ситуациях, от производственных процессов до телефонных сетей или систем моделирования транспортных потоков. Область математики, называемая *теорией очередей*, или *теорией массового обслуживания*, с большим успехом использовалась для анализа и управления сложными системами разных типов. Очереди FIFO также играют важную роль в организации вычислений. Они часто встречаются при работе с компьютером: в очереди может храниться список песен для воспроизведения, документы, отправленные на печать, или события в компьютерной игре.

Возможно, самым масштабным практическим применением очередей является сам Интернет. Его работа основана на перемещении невероятного количества сообщений по огромному множеству очередей, обладающих разными свойствами и связанных разными запутанными отношениями. Понимание и управление такой сложной системой требует надежной реализации абстракции очереди, применения математических результатов теории очередей и имитационного моделирования. Следующий классический пример даст представление об этом процессе.



	поступление	выход	ожидание
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

Очередь M/M/1

Очередь M/M/1

Одна из важнейших моделей очередей — *очередь M/M/1* — хорошо моделирует многие ситуации из реального мира (например, очередь машин, ожидающих у пропускного пункта, или пациентов, ожидающих приема у врача). «M» здесь означает «марковский» (Markovian) или «без запоминания» (memoryless); запись указывает на то, что как поступление, так и обслуживание являются *пуассоновскими процессами*: как и интервалы между двумя последовательными поступлениями, так и время обслуживания подчиняются экспоненциальному распределению (см. упражнение 2.2.8). «1» означает, что в задаче только один сервер. Очередь M/M/1 параметризуется по *интенсивности поступления* λ (например, количеству машин, подъезжающих к пропускному пункту в минуту) и *интенсивности обслуживания* μ (например, количеству машин, проходящих через пропускной пункт в минуту) и характеризуется тремя свойствами.

- Сервер только один, используется дисциплина FIFO.
- Интервалы между последовательными поступлениями заявок (требующих обслуживания «клиентов») распределены по экспоненциальному закону с интенсивностью λ событий в минуту.
- Время обслуживания заявок в непустой очереди распределено по экспоненциальному закону с интенсивностью μ событий в минуту.

Среднее время между поступлениями заявок составляет $1/\lambda$ минут, а среднее время между обслуживаниями (при непустой очереди) составляет $1/\mu$ минут. Очередь будет неограниченно расти, если не выполняется условие $\mu > \lambda$; в противном случае поступление заявок в очередь и выход из нее будут сложным и интересным диамическим процессом.

Анализ

В практических ситуациях людей интересует влияние параметров λ и μ на различные свойства очереди. Клиента может интересовать среднее время ожидания в системе; проектировщику системы захочется узнать, сколько заявок в среднем будет ожидать обслуживания, или получить ответы на более сложные вопросы (например, вероятность того, что размер очереди превысит заданный максимум). Для простых моделей теория вероятностей предоставляет формулы, выражающие эти величины как функции λ и μ. Для очередей $M/M/1$ известно, что:

- среднее количество заявок в системе L равно $\lambda/(\mu - \lambda)$;
- среднее время, проведенное заявкой в системе, W равно $1/(\mu - \lambda)$.

Например, если машины поступают с интенсивностью $\lambda = 10$ в минуту, а интенсивность обслуживания равна $\mu = 15$ в минуту, среднее количество машин в системе будет равно 2, а среднее время, проводимое заявкой в системе, составит 1/5 минуты, или 12 секунд. Эти формулы подтверждают, что время ожидания (и длина очереди) неограниченно возрастает при приближении λ к μ. Они также подчиняются общему правилу, называемому *законом Литтла*: среднее количество заявок в системе в λ раз превышает среднее время, проводимое заявкой в системе ($L = \lambda W$), для многих типов очередей.

Моделирование

Программа `MM1Queue` (листинг 4.3.7) представляет собой клиента для `Queue`, который может использоваться для проверки подобных математических результатов. Это простой пример событийного моделирования: мы генерируем события, происходящие в определенные моменты времени, и изменяем структуры данных для этих событий, моделируя то, что происходит в моменты их возникновения. В очереди M/M/1 существуют события двух видов: поступление заявки и обслуживание заявки. Для этого программа поддерживает две переменные:

- `nextService` — время следующего обслуживания;
- `nextArrival` — время следующего поступления.

Чтобы смоделировать событие поступления, мы ставим в очередь `nextArrival` (время поступления); чтобы смоделировать обслуживание, мы извлекаем из очереди время поступления следующей заявки в очереди, вычисляем время ожидания `wait` (разность между временем завершения обслуживания и временем поступления заявки в очередь) и наносим время `wait` на диаграмму (см. листинг 3.2.3). Диаграмма, полученная после нанесения большого количества испытаний, характеризует

Листинг 4.3.7. Моделирование очереди M/M/1

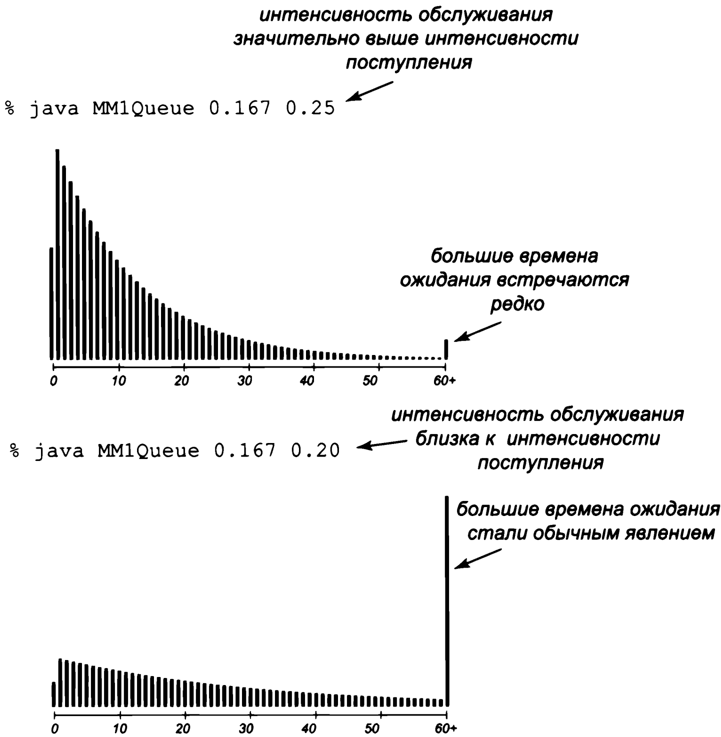
```
public class MM1Queue
{
   public static void main(String[] args)
   {
      double lambda = Double.parseDouble(args[0]);
      double mu     = Double.parseDouble(args[1]);
      Histogram hist = new Histogram(60 + 1);
      Queue<Double> queue = new Queue<Double>();
      double nextArrival = StdRandom.exp(lambda);
      double nextService = nextArrival + StdRandom.exp(mu);
      StdDraw.enableDoubleBuffering();

      while (true)
      {  // Моделирование поступлений перед очередным обслуживанием.
         while (nextArrival < nextService)
         {
            queue.enqueue(nextArrival);
            nextArrival += StdRandom.exp(lambda);
         }

         // Моделирование обслуживания.
         double wait = nextService - queue.dequeue();
         hist.addDataPoint(Math.min(60, (int) Math.round(wait)));
         StdDraw.clear();
         hist.draw();
         StdDraw.show();
         StdDraw.wait(20);
         if (queue.isEmpty())
            nextService = nextArrival + StdRandom.exp(mu);
         else
            nextService = nextService + StdRandom.exp(mu);
      }
   }
}
```

lambda	интенсивность поступления
mu	интенсивность обслуживания
hist	гистограмма
queue	очередь M/M/1
wait	время в очереди

Модель очереди M/M/1 отслеживает временные характеристики при помощи двух переменных nextArrival и nextService, а также очередь Queue с элементами double для вычисления времени ожидания. Значение каждого элемента в очереди представляет (смоделированное) время поступления в очередь. Время ожидания наносится на гистограмму при помощи класса Histogram (листинг 3.2.3).

систему массового обслуживания $M/M/1$. С практической точки зрения одна из важнейших характеристик процесса, которую вы можете выявить сами, запустив `MM1Queue` для различных значений параметров λ и μ, заключается в том, что среднее время пребывания заявки в системе (и среднее время пребывания заявок в системе) может быстро расти при приближении интенсивности обслуживания к интенсивности поступления. При высокой интенсивности обслуживания на гистограмме возникает хорошо заметный «хвост», в котором количество заявок с заданным временем ожидания сокращается до пренебрежимо малой величины

с ростом времени ожидания. Но если интенсивность обслуживания приближается к интенсивности поступления, «хвост» гистограммы растягивается настолько, что большинство значений находится именно в «хвосте», и начинают преобладать заявки с наивысшим временем ожидания.



Примерные результаты выполнения очереди MM1

Как и во многих других примерах, рассмотренных нами, экспериментальная проверка простой и понятной математической модели становится отправной точкой для изучения более сложных ситуаций. В реальных примерах может быть несколько очередей, несколько серверов, серверы с многофазным обслуживанием, конечная длина очередей и другие ограничения. Кроме того, распределение времени между поступлением заявок и времени обслуживания может не иметь математического выражения. В таких ситуациях выбора нет, и приходится прибегать к имитационному моделированию. Проектировщик системы строит компьютерную модель системы массового обслуживания (такую, как `MM1Queue`) и использует ее для экспериментов с параметрами системы (например, интенсивность обслуживания) для обеспечения должной реакции на внешние условия (например, интенсивность поступления заявок).

Коллекции с поддержкой перебора[1]

Как упоминалось ранее в этом разделе, одной из важнейших операций с массивами и связными списками является идиома цикла `for` для обработки каждого элемента. Эта распространенная парадигма программирования не ограничивается низкоуровневыми структурами данных, такими как массивы и связные списки. Для любой коллекции возможность обработки всех элементов (вероятно, в некотором определенном порядке) весьма полезна. Клиент хочет обработать каждый элемент некоторым способом или просто перебрать все элементы коллекции. Эта парадигма настолько важна, что она стала одной из полноправных конструкций в Java и многих других современных языках программирования (иначе говоря, механизмы ее поддержки встраиваются в сам язык, а не реализуются в библиотеках). Она позволяет писать понятный, компактный код, не зависящий от подробностей реализации коллекции.

Чтобы познакомить вас с этой концепцией, мы начнем с фрагмента клиентского кода, который выводит все элементы типа `String` из коллекции, по одному на строку:

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
   StdOut.println(s);
...
```

Эта конструкция иногда называется циклом-итератором или *циклом foreach*: тело цикла `for` выполняется для каждой строки `s` в коллекции. Клиентскому коду ничего не нужно знать о представлении реализации коллекции; он просто обрабатывает каждый элемент коллекции. Тот же цикл *foreach* будет работать с классом `Queue`, содержащим строки, или любой другой коллекцией строк с поддержкой перебора.

Трудно представить более ясный или компактный код. Тем не менее реализация коллекции с поддержкой перебора может потребовать дополнительной работы, которую мы рассмотрим более подробно. Во-первых, конструкция *foreach* является сокращенной формой записи `while`. Например, цикл *foreach*, приведенный ранее, эквивалентен следующей конструкции `while`:

```
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext())
{
   String s = iterator.next();
   StdOut.println(s);
}
```

[1] *Англ.* iterable. Понятие «итерирование» в значении «поэлементный перебор содержимого чего-либо», а также «итерация», «итерационный процесс» и т. п. обычно воспринимаются как термины и не переводятся. Но здесь следовало начать с элементарных разъяснений. — *Примеч. науч. ред.*

В этом коде хорошо видны три необходимые части, которые должны быть реализованы в любой коллекции с поддержкой перебора.

- Коллекция должна реализовать метод `iterator()`, возвращающий объект `Iterator`.
- Класс `Iterator` должен включать два метода: `hasNext()` (возвращает значение `boolean`) и `next()` (возвращает элемент из коллекции).

В Java для выражения идеи о том, что класс реализует некоторое множество методов, используется механизм наследования интерфейсов (см. раздел 3.3). Для коллекций с поддержкой перебора необходимые интерфейсы заранее определены в Java.

Чтобы включить поддержку перебора в класс, прежде всего следует добавить в его объявление конструкцию `implements Iterable<Item>`, соответствующую интерфейсу:

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

(определяемому в `java.lang.Iterable`), и добавить в класс метод, возвращающий `Iterator<Item>`. Итераторы являются обобщениями; вы можете использовать их для того, чтобы предоставить клиенту возможность перебора объектов некоторого типа (и только объектов этого конкретного типа).

Что такое итератор? Это объект класса, реализующего методы `hasNext()` и `next()`, как в следующем интерфейсе (определенном в `java.util.Iterator`):

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

И хотя интерфейс требует также метода `remove()`, в этой книге всегда используется пустой метод `remove()`, потому что чередования перебора с операциями, изменяющими структуру данных, лучше избегать.

Как видно из двух следующих примеров, реализация класса-итератора для реализаций на базе массивов и связных списков достаточно очевидна.

Итераторы в классе, использующем массив

В качестве первого примера мы рассмотрим все шаги, необходимые для поддержки перебора в `ArrayStackOfStrings` (листинг 4.3.1). Сначала объявление класса приводится к виду

```
public class ArrayStackOfStrings implements Iterable<String>
```

Другими словами, мы обещаем предоставить метод `iterator()`, чтобы клиент мог использовать цикл *foreach* для перебора строк в стеке. Сам метод `iterator()` выглядит просто:

```
public Iterator<String> iterator()
{  return new ReverseArrayIterator();  }
```

Он просто возвращает объект закрытого вложенного класса, реализующего интерфейс `Iterator` (который предоставляет методы `hasNext()`, `next()` и `remove()`):

```
private class ReverseArrayIterator implements Iterator<String>
{
   private int i = n-1;

   public boolean hasNext()
   {  return i >= 0;  }

   public String next()
   {  return items[i--];  }

   public void remove()
   { }
}
```

Обратите внимание: вложенный класс `ReverseArrayIterator` может обращаться к переменным экземпляров внешнего класса, в данном случае `items[]` и `n` (собственно, ради этой возможности мы и использовали вложенные классы для итераторов). Остается одна важная подробность: необходимо включить в начало `ArrayStackOfStrings` строку

```
import java.util.Iterator;
```

Теперь, поскольку клиентский код может использовать цикл *foreach* с объектами `ArrayStackOfStrings`, он может перебирать элементы, ничего не зная о низкоуровневом представлении массива. Эта схема исключительно важна для реализации фундаментальных типов коллекций. Например, она позволяет перейти на совершенно новое представление *без изменения клиентского кода*. Что еще важнее, с точки зрения клиента, она позволяет ему использовать механизм перебора, *ничего не зная о подробностях реализации*.

Итераторы в классе, использующем связный список

Те же действия (с другим кодом) позволяют обеспечить поддержку перебора в классе `Queue` (листинг 4.3.6), хотя он и является обобщенным. Сначала мы приводим объявление класса к виду

```
public class Queue<Item> implements Iterable<Item>
```

Другими словами, мы обещаем предоставить метод `iterator()`, чтобы клиент мог использовать цикл *foreach* для перебора элементов очереди независимо от типа. Как и в предыдущем случае, сам метод `iterator()` выглядит просто:

```
public Iterator<Item> iterator()
{  return new ListIterator();  }
```

И снова определяется закрытый вложенный класс, реализующий интерфейс `Iterator`:

```
private class ListIterator implements Iterator<Item>
{
   Node current = first;
   public boolean hasNext()
   {  return current != null; }
   public Item next()
   {
      Item item = current.item;
      current = current.next;
      return item;
   }
   public void remove()
   { }
}
```

И снова клиент может построить очередь элементов произвольного типа, а затем перебрать ее элементы, ничего не зная о низкоуровневом представлении связного списка:

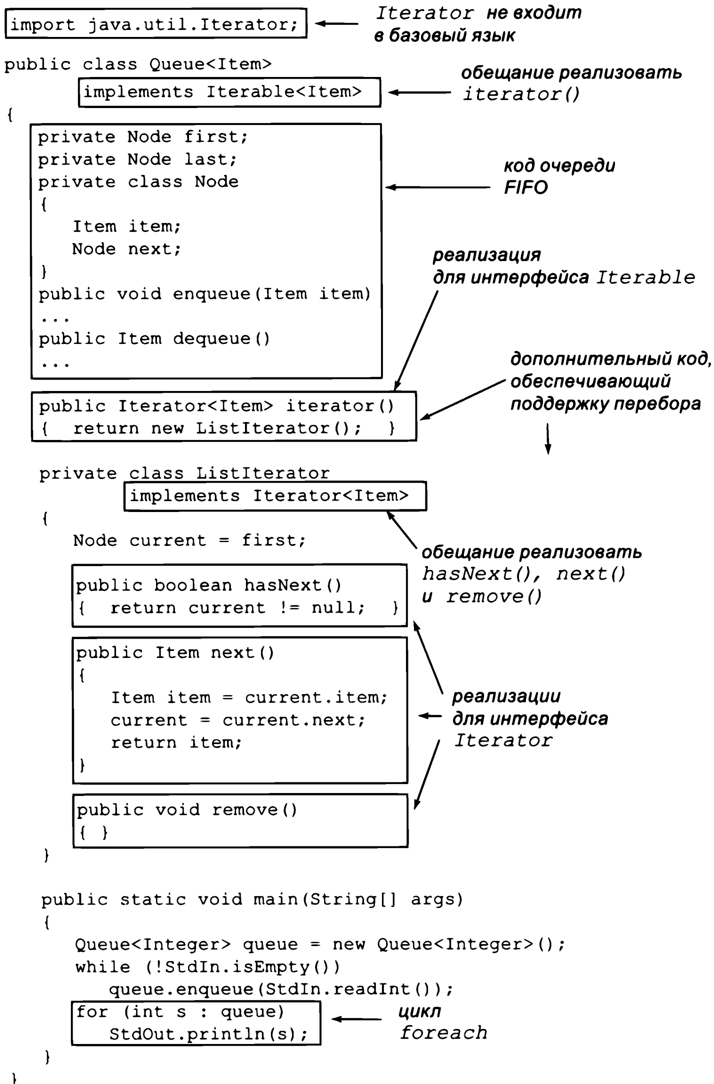
```
Queue<String> queue = new Queue<String>();
...
for (String s : queue)
   StdOut.println(s);
```

Этот клиентский код более ясно выражает суть вычислений, а значит, в ходе его разработки и сопровождения возникнет меньше проблем, чем с кодом, основанным на низкоуровневом представлении.

Наш итератор стека перебирает элементы в порядке LIFO, а итератор очереди перебирает их в порядке FIFO, хотя формально такого требования нет: элементы можно возвращать в любом порядке. Однако при разработке итератора желательно соблюдать простое правило: если спецификация типа данных подразумевает существование естественного порядка перебора — *используйте его*.

Реализации с поддержкой перебора на первый взгляд кажутся сложными, но они оправдывают затраченные усилия. Реализовывать их приходится относительно редко, а когда приходится, вы пользуетесь всеми преимуществами понятного и правильного клиентского кода и возможностями его повторного использования. Более того, как и с любой программной конструкцией, стоит вам оценить эти преимущества, как вы начнете пользоваться ими чаще.

Конечно, включение в класс поддержки перебора изменяет его API. Чтобы избежать излишнего усложнения описаний API, мы просто добавляем слова «с поддержкой перебора», показывая, что в класс был включен соответствующий код и клиент может использовать цикл *foreach* в клиентском коде. С этого момента мы будем использовать в клиентских программах (обобщенные) типы данных `Stack`, `Queue` и `RandomQueue` с поддержкой перебора.



Строение класса с поддержкой перебора

Распределение ресурсов

Рассмотрим другой пример, демонстрирующий использование структур данных и языковых возможностей Java, о которых мы говорили. Пусть имеется система, содержащая большое количество серверов с нежестким взаимодействием, которые должны совместно использовать некоторые ресурсы. Каждый сервер соглашается поддерживать собственную очередь элементов данных, а некий управляющий

центр распределяет элементы по серверам (и сообщает пользователям о том, где их найти). Например, такими элементами могут быть песни, фотографии или видеоролики, к которым необходимо предоставить доступ большому количеству пользователей. Для большей определенности, речь идет о миллионах элементов и тысячах серверов.

Рассмотрим программу, которая может использоваться управляющим центром для распределения элементов. Пока не будем обращать внимания на динамику процессов удаления элементов из системы, добавления и удаления серверов и т. д.

Если использовать *циклическую* стратегию распределения с последовательным перебором серверов, вы можете добиться сбалансированного распределения, но столь полный контроль за ситуацией обычно невозможен: например, в системе может существовать большое количество независимых распределителей, ни один из которых не обладает полной информацией о серверах. Соответственно такие системы чаще используют *случайную* стратегию, основанную на случайном выборе. Еще более эффективна стратегия, основанная на случайной *выборке* серверов и назначении нового элемента на наименее загруженный из них. Для малых очередей различия между стратегиями несущественны, но в системе с миллионами элементов на тысячах серверов выбор стратегии приводит к весьма существенным последствиям, потому что каждый сервер располагает фиксированным объемом ресурсов, которые могут быть выделены для этого процесса. Аналогичные системы используются в сетевом оборудовании Интернета, где некоторые очереди реализуются специализированными устройствами и длина очереди напрямую превращается в дополнительные расходы на приобретение аппаратуры. Но выборку какого размера следует использовать?

Программа `LoadBalance` (листинг 4.3.8) моделирует стратегию распределения ресурсов с использованием выборки, которая может использоваться для исследования этого вопроса. Рассматривавшиеся ранее структуры данных (очереди и случайные очереди) и высокоуровневые конструкции (обобщения и итераторы) эффективно используются для написания хорошо понятного кода, который может использоваться для экспериментов. В модели формируется случайная очередь из очередей (изображающих серверы), а внутренний цикл направляет каждый очередной запрос в минимальную очередь из выборки очередей, полученной при помощи метода `sample()` из `RandomQueue` (упражнение 4.3.36). Результат, к которому мы приходим, удивляет: выборка с размером 2 обеспечивает практически идеальный баланс, поэтому использовать выборки большего размера нет смысла.

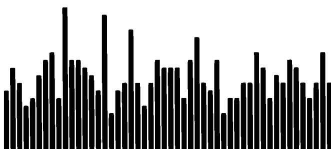
Мы подробно рассмотрели основные проблемы, связанные с затратами памяти и времени при использовании базовых реализаций API стека и очереди, — и не только из-за важности и полезности этих структур данных, но и потому, что вы с большой вероятностью столкнетесь с теми же проблемами при реализации ваших собственных конструкций.

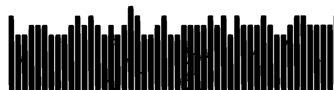
Листинг 4.3.8. Моделирование распределения нагрузки

```
public class LoadBalance
{
   public static void main(String[] args)
   {  // Распределение n элементов на m серверах с использованием
      // стратегии "кратчайшей очереди из выборки".
      int m = Integer.parseInt(args[0]);
      int n = Integer.parseInt(args[1]);
      int size = Integer.parseInt(args[2]);

     // Создание очередей на серверах.
      RandomQueue<Queue<Integer>> servers;
      servers = new RandomQueue<Queue<Integer>>();
      for (int i = 0; i < m; i++)
         servers.enqueue(new Queue<Integer>());

      for (int j = 0; j < n; j++)
      {  // Назначение элемента серверу.
         Queue<Integer> min = servers.sample();
         for (int k = 1; k < size; k++)
         {  // Выбор случайного сервера с обновлением
            // при появлении нового минимума.
            Queue<Integer> queue = servers.sample();
            if (queue.size() < min.size()) min = queue;
         }  // min - минимальная очередь на сервере.
         min.enqueue(j);
      }
      int i = 0;
      double[] lengths = new double[m];
      for (Queue<Integer> queue : servers)
         lengths[i++] = queue.size();
      StdDraw.setYscale(0, 2.0*n/m);
      StdStats.plotBars(lengths);
   }
}
```

`m`	количество серверов
`n`	количество элементов
`size`	размер выборки
`servers`	очереди
`min`	кратчайшая очередь в выборке
`queue`	текущий сервер

Приложение-клиент, использующее Queue и RandomQueue для моделирования процесса распределения n элементов в множестве из m серверов. Запросы помещаются в минимальную очередь из случайной выборки размера size.





Стоит ли в приложении, работающем с коллекцией данных, использовать стек, очередь FIFO или случайную очередь? Ответ на этот вопрос завист от высокоуровневого анализа клиента с целью определения того, какая из дисциплин — LIFO, FIFO или случайная выборка — лучше подходит в конкретной ситуации.

Какую структуру следует применить для организации ваших данных: массив, связный список или массив с изменяемым размером? Ответ на этот вопрос зависит от низкоуровневого анализа характеристик быстродействия. Основное преимущество *массива* — фиксированное время обращения к любому элементу; основной недостаток — необходимость заранее знать максимальную длину. У *связного списка* преимуществом является отсутствие ограничений на количество хранимых данных, а недостатком — невозможность обращения к произвольному элементу за фиксированное время. *Массив с изменяемым размером* сочетает преимущества массивов и связных списков (обращения к элементам выполняются за фиксированное время, но знать максимальную длину заранее не обязательно), но есть (небольшой) недостаток: фиксированным будет лишь «амортизированное» (усредненное) время выполнения. Каждая структура данных лучше подходит для определенных ситуаций, и в большинстве сред программирования вам, скорее всего, встретятся все три. Например, класс Java `java.util.ArrayList` использует массивы изменяемого размера, а класс Java `java.util.LinkedList` использует связный список.

Мощные высокоуровневые конструкции и новые возможности языка, которые были рассмотрены в этом разделе (обобщения и итераторы), не следует воспринимать как нечто само собой разумеющееся. Эти нетривиальные языковые возможности получили широкое распространение в основных языках только в начале XXI века, и они используются прежде всего профессиональными программистами. Тем не менее в последнее время они встречаются все чаще, потому что они хорошо поддерживаются в Java и C++, их поддержка была внедрена в более новых языках, таких как Python и Ruby, а многие разработчики начали понимать, какую пользу они могут принести в клиентском коде. К настоящему моменту вы уже знаете, что изучить новую возможность языка программирования — примерно то же самое, что научиться езде на велосипеде или написать `HelloWorld`: проблема выглядит совершенно загадочно, пока вы не сделаете это в первый раз, а потом решение быстро становится вашей второй натурой. Умение пользоваться обобщениями и итераторами определенно стоит потраченного времени.

Вопросы и ответы

В. Когда следует использовать `new` с `Node`?

О. Как и для многих других классов, ключевое слово `new` используется только тогда, когда вы хотите создать новый объект `Node` (новый узел в связном списке.) Не используйте `new` для создания новой ссылки на существующий объект `Node`. Например, код

```
Node oldFirst = new Node();
oldFirst = first;
```

создает новый объект `Node`, после чего немедленно перестает отслеживать единственную ссылку на него. Этот код не возвращает ошибку, но создавать потерянные объекты без необходимости свойственно неряшливому стилю программирования.

В. Почему мы объявляем `Node` как вложенный класс? Почему мы объявляем его закрытым?

О. Когда вложенный класс `Node` объявляется закрытым, методы внешнего класса могут обращаться к объектам `Node`, но доступ к ним из других классов запрещен. *Примечание для специалистов*: вложенный класс, не являющийся статическим, называется *внутренним* (inner) классом, так что формально наши классы `Node` являются внутренними классами — хотя другие, не являющиеся обобщенными, могут быть статическими.

В. Когда я ввожу команду `javac LinkedStackOfStrings.java` для запуска программы из листинга 4.3.2 и других похожих программ, в дополнение к файлу LinkedStackOfStrings.class появляется файл LinkedStackOfStrings$Node.class. Для чего он нужен?

О. Это файл вложенного класса `Node`. В соглашениях об именах языка Java символ `$` используется для отделения имени внешнего класса от имени внутреннего класса.

В. Следует ли разрешить клиенту вставлять `null`-элементы в стек или в очередь?

О. Этот вопрос часто возникает при реализации коллекций в Java. Наша реализация (и библиотеки стеков и очередей языка Java) разрешает вставку значений `null`.

В. Существуют ли в Java библиотеки для работы со стеками и очередями?

О. И да, и нет. В Java есть встроенная библиотека с именем `java.util.Stack`, но вам не стоит использовать ее для работы со стеком. Она содержит дополнительные операции, которые обычно не поддерживаются для стеков, — например, чтение `i`-го элемента. Также существует возможность добавления элементов на дно стека (а не на вершину), так что библиотека может использоваться для реализации очереди! И хотя лишние операции могут показаться приятным дополнением, на самом деле это скорее проклятие. Мы используем типы данных не потому, что они поддерживают все мыслимые операции, а потому, что они позволяют точно определить те операции, которые нам нужны. Это делается прежде всего для того, чтобы система могла помешать предотвращению операций, которые вам не нужны. API `java.util.Stack` — пример *широкого интерфейса*, которого мы обычно стараемся избегать.

В. Я хочу использовать представление массива для обобщенного стека, но следующий код не компилируется. В чем дело?

```
private Item[] item = new Item[capacity];
```

О. Хорошая попытка. К сожалению, Java не разрешает создавать массивы обобщений. Специалисты продолжают ожесточенные споры на эту тему. Впрочем, существует обходное решение: преобразование типа. Вы можете использовать запись:

```
private Item[] item = (Item[]) new Object[capacity];
```

В. Почему нужно импортировать `java.util.Iterator`, но не `java.lang.Iterable`?

О. По историческим причинам интерфейс `Iterator` является частью пакета `java.util`, который не импортируется по умолчанию. Интерфейс `Iterable` появился относительно недавно, и он включен в пакет `java.lang`, который импортируется по умолчанию.

В. Могу ли я использовать конструкцию *foreach* с массивами?

О. Да (хотя формально массивы не реализуют интерфейс `Iterable`).

Следующий код выводит аргументы командной строки в стандартный вывод:

```
public static void main(String[] args)
{
   for (String s : args)
      StdOut.println(s);
}
```

В. Что произойдет, если при использовании обобщений опустить аргумент-тип в объявлении или вызове конструктора?

```
Stack<String> stack = new Stack();          // Небезопасно
Stack         stack = new Stack<String>();  // Небезопасно
Stack<String> stack = new Stack<String>();  // Правильно
```

О. Первая строка выдает предупреждение на стадии компиляции. Вторая строка выдает предупреждение на стадии компиляции, если вызвать `stack.push()` с аргументом `String`, и ошибку компиляции, если присвоить результат `stack.pop()` переменной типа `String`. В третьей строке можно использовать оператор `<>`, позволяющий Java вычислить аргумент-тип для вызова конструктора по контексту:

```
Stack<String> stack = new Stack<>();   // Оператор <>
```

В. Почему бы не создать единый тип данных `Collection`, реализующий методы для добавления элементов, извлечения элемента, вставленного позже всех остальных, извлечения элемента, вставленного ранее всех остальных, извлечения случайного элемента, перебора элементов, возвращения количества элементов в коллекции и любых других операций, которые нам могут понадобиться? Тогда все эти операции будут реализованы в одном классе, который будет использоваться многими клиентами.

О. Это типичный пример широкого интерфейса, которого, как упоминалось в разделе 3.3, следует избегать. Одна из причин нежелательности широких интерфейсов — трудность построения единой реализации, эффективной для всех операций.

Более важная причина заключается в том, что узкие интерфейсы формируют в программах определенную дисциплину, благодаря которой клиентский код становится более понятным. Если один клиент использует `Stack<String>`, а другой использует `Queue<Customer>`, можно понять, что для первого важна дисциплина LIFO, а для второго — дисциплина FIFO. Другой подход основан на применении наследования с целью инкапсуляции операций, общих для всех коллекций. Впрочем, такие реализации создаются специалистами, тогда как научиться строить обобщенные реализации, подобные `Stack` и `Queue`, способен любой программист.

Упражнения

4.3.1. Добавьте в программу `ArrayStackOfStrings` (листинг 4.3.1) метод `isFull()`, который возвращает `true`, если размер стека равен емкости массива. Измените операцию `push()`, чтобы она генерировала исключение при вызове с заполненным стеком.

4.3.2. Приведите результататы выполнения `java ArrayStackOfStrings 5` для следующих входных данных:

```
it was - the best - of times - - - it was - the - -
```

4.3.3. Предположим, клиент выполняет со стеком серию чередующихся операций вставки и извлечения. Операции вставки заносят в стек целые числа от 0 до 9 (по порядку); операции извлечения выводят возвращаемые значения. Какая из следующих выходных последовательностей *невозможна*?

```
a.  4 3 2 1 0 9 8 7 6 5
b.  4 6 8 7 5 3 2 9 0 1
c.  2 5 6 7 4 8 9 3 1 0
d.  4 3 2 1 0 5 6 7 8 9
e.  1 2 3 4 5 6 9 8 7 0
f.  0 4 6 5 3 8 1 7 2 9
g.  1 4 7 9 8 6 5 3 0 2
h.  2 1 4 3 6 5 8 7 9 0
```

4.3.4. Напишите фильтр `Reverse`, который читает строки из стандартного ввода и направляет их в стандартный вывод в обратном порядке. Используйте в реализации стек или очередь.

4.3.5. Напишите статический метод, который читает числа с плавающей точкой из стандартного ввода (по одному) и возвращает массив с этими числами в том порядке, в каком они следовали в стандартном вводе. *Подсказка*: используйте стек или очередь.

4.3.6. Напишите клиент стека `Parentheses`, который читает из стандартного ввода последовательность круглых скобок, квадратных скобок и фигурных скобок и использует стек для проверки их сбалансированности. Например, для серии `[()]{}{[()()]()}` программа должна выводить `true`, а для `[(])` — `false`. *Подсказка*: используйте стек.

4.3.7. Что выведет следующий фрагмент кода для n = 50? Приведите высокоуровневое описание работы этого фрагмента для положительного целого n.

```
Stack<Integer> stack = new Stack<Integer>();
while (n > 0)
{
   stack.push(n % 2);
   n /= 2;
}
while (!stack.isEmpty())
    StdOut.print(stack.pop());
StdOut.println();
```

Ответ: он выводит двоичное представление n (`110010` для n = 50).

4.3.8. Что делает следующий фрагмент кода с очередью `queue`?

```
Stack<String> stack = new Stack<String>();
while (!queue.isEmpty())
   stack.push(queue.dequeue());
while (!stack.isEmpty())
   queue.enqueue(stack.pop());
```

4.3.9. Добавьте в класс `Stack` (листинг 4.3.4) метод `peek()`, который возвращает последний вставленный элемент стека (без его извлечения.)

4.3.10. Приведите содержимое и длину массива для `ResizingArrayStackOfStrings` со следующими входными данными:

```
it was - the best - of times - - - it was - the - -
```

4.3.11. Добавьте в классы `Stack` (листинг 4.3.4) и `Queue` (листинг 4.3.6) метод `size()`, возвращающий количество элементов в коллекции. *Подсказка*: проследите за тем, чтобы ваш метод выполнялся за фиксированное время. Для этого определите переменную экземпляра n, которая инициализируется 0, увеличивается в `push()` и `enqueue()`, уменьшается в `pop()` и `dequeue()` и возвращается в `size()`.

4.3.12. Постройте диаграмму использования памяти в стиле диаграмм из раздела 4.1 для примера с тремя узлами, использованного для демонстрации связных списков в этом разделе.

4.3.13. Напишите программу, которая читает из стандартного ввода выражение без левых круглых скобок и выводит эквивалентное инфиксное выражение со вставленными скобками. Например, для ввода

```
1 + 2 ) * 3 - 4 ) * 5 - 6 ) ) )
```

программа должна выводить

```
( ( 1 + 2 ) * ( ( 3 - 4 ) * ( 5 - 6 ) ) )
```

4.3.14. Напишите фильтр `InfixToPostfix`, который преобразует арифметическое выражение из инфиксной формы в постфиксную.

4.3.15. Напишите программу `EvaluatePostfix`, которая читает из стандартного ввода постфиксное выражение, вычисляет его результат и выводит полученное значение. (Перенаправление вывода программы из предыдущего упражнения этой программе дает поведение, эквивалентное программе `Evaluate` из листинга 4.3.5.)

4.3.16. Предположим, клиент выполняет чередующуюся последовательность операций вставки и извлечения из очереди FIFO. Операции вставки заносят в стек целые числа от 0 до 9 (по порядку); операции извлечения выводят возвращаемые значения. Какая из следующих последовательностей *невозможна*?

```
a.  0 1 2 3 4 5 6 7 8 9
b.  4 6 8 7 5 3 2 9 0 1
c.  2 5 6 7 4 8 9 3 1 0
d.  4 3 2 1 0 5 6 7 8 9
```

4.3.17. Напишите для класса `Stack` с поддержкой перебора клиентское приложение со статическим методом `copy()`, который получает в качестве аргумента стек со строковыми элементами и возвращает копию этого стека. Альтернативное решение представлено в упражнении 4.3.48.

4.3.18. Напишите клиентское приложение для класса `Queue`, которое получает в командной строке целочисленный аргумент `k` и выводит `k`-ю с конца строку из стандартного потока ввода.

4.3.19. Разработайте тип данных `ResizingArrayQueueOfStrings`, реализующий очередь на базе массива фиксированной длины так, чтобы все операции выполнялись за фиксированное время. Затем расширьте свою реализацию, чтобы она использовала массив с изменяемым размером для отмены ограничений на длину. *Подсказка*: проблема связана с тем, что элементы будут «ползать» по массиву в результате добавления и извлечения из очереди. Используйте арифметические вычисления по модулю для индексов элементов, находящихся в начале и в конце очереди.

StdIn	StdOut	n	lo	hi	items[]							
					0	1	2	3	4	5	6	7
		0	0	0	null							
to		1	0	1	to	null						
be		2	0	2	to	be						
or		3	0	3	to	be	or	null				
not		4	0	4	to	be	or	not				
to		5	0	5	to	be	or	not	to	null	null	null
-	to	4	1	4	null	be	or	not	to	null	null	null
be		5	1	6	null	be	or	not	to	be	null	null
-	be	4	2	6	null	null	or	not	to	be	null	null
-	or	3	3	6	null	null	null	not	to	not	null	null
that		4	3	7	null	null	null	not	to	not	that	null

4.3.20. (Для любителей математики.) Докажите, что массив из ResizingArrayStackOfStrings никогда не заполняется менее чем на четверть. Затем докажите, что для любого клиента ResizingArrayStackOfStrings общие затраты на все операции со стеком, отнесенные к количеству операций, ограничены константой.

4.3.21. Измените программу MM1Queue (листинг 4.3.7) и создайте программу MD1Queue, которая моделирует очередь с фиксированным (детерминированным) временем обслуживания с интенсивностью μ. Проверьте закон Литтла для этой модели.

4.3.22. Разработайте класс StackOfInts, использующий для реализации стека целых чисел представление связного списка (но без обобщений). Напишите клиентское приложение, которое сравнивает быстродействие вашей реализации с Stack<Integer> для оценки потерь от использования автоупаковки в системе.

Упражнения для связных списков

Эти упражнения помогут вам приобрести опыт работы со связными списками. Чтобы вам было проще выполнить их, используйте визуальное представление того, что описано в тексте.

4.3.23. Допустим, x — узел Node связного списка. Что произойдет при выполнении следующего фрагмента кода?

```
x.next = x.next.next;
```

Ответ: из списка будет удален узел, следующий за x.

4.3.24. Напишите метод find(), который получает в аргументах первый узел Node связного списка и строку key и возвращает true, если в списке присутствует узел, поле item которого содержит key; в противном случае возвращается false.

4.3.25. Напишите метод delete(), который получает первый узел Node связного списка и аргумент k типа int и удаляет k-й узел из связного списка (если он существует).

4.3.26. Допустим, x — узел Node связного списка. Что произойдет при выполнении следующего фрагмента кода?

```
t.next = x.next;
x.next = t;
```

Ответ: узел t будет вставлен непосредственно после узла x.

4.3.27. Почему следующий фрагмент кода не приводит к тому же результату, что и фрагмент кода из предыдущего упражнения?

```
x.next = t;
t.next = x.next;
```

Ответ: когда приходит время обновления `t.next`, `x.next` содержит уже не исходный узел, следующий за `x`, а сам узел `t`!

4.3.28. Напишите метод `removeAfter()`, который получает в качестве аргумента узел связного списка `Node` и удаляет узел, следующий за указанным узлом (и не делает ничего, если аргумент равен `null` или поле `next` аргумента содержит `null`).

4.3.29. Напишите метод `copy()`, который получает в качестве аргумента узел связного списка `Node` и создает новый связный список с такой же последовательностью элементов без уничтожения исходного списка.

4.3.30. Напишите метод `remove()`, который получает в в качестве аргумента узел связного списка `Node` и строку `key` и удаляет из списка все узлы с полем `item`, равным `key`.

4.3.31. Напишите метод `max()`, который получает в аргументе первый узел `Node` связного списка и возвращает значение максимального элемента в списке. Предполагается, что все элементы содержат целые числа, а если связный список пуст, метод возвращает 0.

4.3.32. Разработайте рекурсивное решение для предыдущего упражнения.

4.3.33. Напишите метод, который получает в качестве аргумента первый узел `Node` связного списка, переставляет элементы в обратном порядке и возвращает первый узел `Node` результата.

4.3.34. Напишите рекурсивный метод для вывода элементов связного списка в обратном порядке. Не изменяйте связи в списке. *Простое решение*: квадратичное время, фиксированные дополнительные затраты памяти. *Другое простое решение*: линейное время, линейные дополнительные затраты памяти. *Более сложное решение*: разработайте алгоритм «разделяй и властвуй», выполняемый за линейно-логарифмическое время с логарифмическими дополнительными затратами памяти.

4.3.35. Напишите рекурсивный метод для случайной перестановки узлов связного списка посредством изменения ссылок. *Простое решение*: квадратичное время, фиксированные дополнительные затраты памяти. *Более сложное решение*: разработайте алгоритм «разделяй и властвуй», выполняемый за линейно-логарифмическое время с логарифмическими дополнительными затратами памяти. Шаг «слияния» описан в упражнении 1.4.40.

Упражнения повышенной сложности

4.3.36. *Дек*. Двусторонняя очередь, или *дек* (*deque*), — коллекция, сочетающая характеристики стека и очереди. Напишите класс `Deque`, который использует связный список для реализации следующего API:

public class Deque<Item>	
Deque()	создает пустой дек
boolean isEmpty()	дек пуст?
void enqueue(Item item)	добавляет item в конец
void push(Item item)	добавляет item в начало
Item pop()	извлекает и возвращает элемент в начале
Item dequeue()	извлекает и возвращает элемент в конце

API обобщенной двусторонней очереди

4.3.37. *Случайная очередь.* Случайная очередь представляет собой коллекцию, которая поддерживает следующий API:

public class RandomQueue<Item>	
RandomQueue()	создает пустую случайную очередь
boolean isEmpty()	случайная очередь пуста?
void enqueue(Item item)	добавляет item в случайную очередь
Item dequeue()	извлекает и возвращает случайный элемент (выборка без возмещения)
Item sample()	возвращает случайный элемент без извлечения его из очереди

API обобщенной случайной очереди

Напишите класс `RandomQueue`, который реализует этот API. *Подсказка*: используйте массив с изменяемым размером. Чтобы удалить элемент из очереди, поменяйте местами элемент в случайной позиции (индексы от 0 до `n-1`) с последним элементом (индекс `n-1`). Затем извлеките и верните последний элемент, как в `ResizingArrayStack`[1]. Напишите приложение-клиент, которое выводит содержимое колоды карт в случайном порядке, используя `RandomQueue<Card>`.

4.3.38. *Случайный итератор.* Напишите для класса `RandomQueue<Item>` из предыдущего упражнения итератор, который возвращает элементы в случайном порядке. Разные итераторы должны возвращать элементы в разных случайных порядках. *Примечание*: это упражнение сложнее, чем кажется на первый взгляд.

4.3.39. *Задача Иосифа Флавия.* В древней задаче Иосифа Флавия n людей, оказавшихся в безвыходном положении, решают убить друг друга. Они располагаются по кругу (в позициях от 0 до $n - 1$) и начинают убивать каждого m-го человека, пока не останется только один. Легенда гласит, что Иосиф рассчитал, где следует сесть, чтобы сохранить себе жизнь. Напишите приложение `Josephus` — клиент `Queue`, которое получает как аргументы командной строки два целых числа m и n

---

[1] Это необратимо меняет порядок элементов *внутри* очереди, что делает возможной *только* случайную выборку. — *Примеч. науч. ред.*

и выводит порядок исключения (а следовательно, показывает, какое место должен занять Иосиф в круге).

```
% java Josephus 2 7
1 3 5 0 4 2 6
```

4.3.40. *Расширенная очередь.* Реализуйте класс со следующим API, который сочетает свойства очереди и стека, поддерживая удаление i-го из элементов, вставленных последними:

public class GeneralizedQueue<Item>	
GeneralizedQueue()	создает пустую расширенную очередь
boolean isEmpty()	расширенная очередь пуста?
void add(Item item)	вставляет item в расширенную очередь
Item remove(int i)	извлекает и возвращает i-й из последних вставленных элементов
int size()	количество элементов в очереди

API обобщенной расширенной очереди

Сначала разработайте реализацию, использующую массив с изменяемым размером, потом реализацию на основе связного списка. (В упражнении 4.4.57 представлена более эффективная реализация на основе дерева бинарного поиска.)

4.3.41. *Циклический буфер. Циклический буфер* (или *циклическая очередь*) представляет собой коллекцию FIFO для хранения последовательности элементов вплоть до предварительно заданного предела. При вставке элемента в заполненный циклический буфер новый элемент заменяет тот, который был вставлен ранее всех остальных. Циклические буферы хорошо подходят для передачи данных между асинхронными процессами и для хранения файлов журналов. Когда буфер пуст, потребитель ждет появления данных; когда буфер заполнен, производитель ждет отправки данных (чтобы добавить новые на освободившееся место). Разработайте API циклического буфера и реализацию, использующую массив фиксированной длины.

4.3.42. *Слияние двух отсортированных очередей.* Даны две очереди со строками, упорядоченными по возрастанию. Переместите все строки в третью очередь так, чтобы строки в третьей очереди были упорядочены по возрастанию.

4.3.43. *Нерекурсивная сортировка слиянием.* Для заданных n строк создайте n очередей, каждая из которых содержит одну строку. Создайте очередь из n очередей. Затем многократно применяйте операцию сортировки слиянием из предыдущего упражнения к первым двум очередям в очереди и ставьте в очередь результат слияния. Повторяйте, пока очередь очередей не будет содержать только одну очередь.

4.3.44. *Очередь из двух стеков.* Покажите, как реализовать очередь с использованием двух стеков. *Подсказка*: если занести элементы в стек, а затем извлечь их, они будут следовать в обратном порядке. Повторение процесса вернет их в порядок FIFO.

4.3.45. *Перемещение вперед.* Прочитайте последовательность символов из стандартного потока ввода и сформируйте из символов связный список символов без дублирования. Когда вы читаете символ, не встречавшийся ранее, вставьте его в начало списка. Когда вы читаете символ-дубликат, удалите его из списка и вставьте снова в начало. Так реализуется хорошо известная стратегия перемещения вперед, применяемая при кэшировании, сжатии данных и во многих других ситуациях, в которых более вероятны повторные обращения к элементам, к которым были недавние обращения.

4.3.46. *Топологическая сортировка.* Требуется упорядочить n заданий, пронумерованных от 0 до $n - 1$, для выполнения их на сервере. Некоторые задания могут быть запущены только после завершения других заданий. Напишите программу `TopologicalSorter`, которая получает аргумент командной строки n и читает из стандартного потока ввода последовательность упорядоченных пар заданий $i\ j$, после чего выводит такую последовательность целых чисел, что для каждой введенной пары $i\ j$ задание i предшествует заданию j. Используйте следующий алгоритм: сначала постройте для каждого задания из ввода (1) очередь заданий, которые должны выполняться после него, и (2) степень входов (количество заданий, которые должны предшествовать ему). Затем постройте очередь всех узлов, у которых степень входов равна 0, и последовательно удаляйте все задания со степенью входов 0, сохраняя целостность всех структур данных. Этот процесс находит много практических применений. Например, с его помощью можно смоделировать предусловия для прохождения учебных курсов и найти последовательность курсов, необходимую для получения ученой степени.

4.3.47. *Буфер текстового редактора.* Разработайте тип данных для буфера текстового редактора, который реализует следующий API:

`public class Buffer`	
`Buffer()`	создает пустой буфер
`void insert(char c)`	вставляет символ c в позиции курсора
`char delete()`	возвращает символ в позиции курсора, удаляя его из буфера
`void left(int k)`	перемещает курсор на k позиций влево
`void right(int k)`	перемещает курсор на k позиций вправо
`int size()`	количество символов в буфере

API текстового буфера

Подсказка: используйте два стека.

4.3.48. *Копирующий конструктор для стека.* Создайте новый конструктор для реализации `Stack` на базе связного списка, чтобы строка кода

```
Stack<Item> t = new Stack<Item>(s);
```

сохраняла в `t` ссылку на новую, независимую копию стека `s`. Операции вставки и извлечения с `s` или `t` должны выполняться независимо друг от друга.

4.3.49. *Копирующий конструктор для очереди.* Создайте новый конструктор, чтобы строка кода

```
Queue<Item> r = new Queue<Item>(q);
```

сохраняла в `r` ссылку на новую, независимую копию очереди `q`.

4.3.50. *Цитата.* Разработайте тип данных `Quote`, реализующий следующий API для работы с цитатами:

public class Quote	
Quote()	создает пустую цитату
void add(String word)	присоединяет слово word в конец цитаты
void add(int i, String word)	вставляет слово word в элемент с индексом i
String get(int i)	слово с индексом i
int count()	количество слов в цитате
String toString()	все слова цитаты

API для работы с цитатами

Определите вложенный класс `Card`, который содержит одно слово цитаты, и ссылку на следующее слово:

```
private class Card
{
   private String word;
   private Card next;
   public Card(String word)
   {
      this.word = word;
      this.next = null;
   }
}
```

4.3.51. *Циклическая цитата.* Повторите предыдущее упражнение, но используйте циклический связный список (так называемое *кольцо*). В циклическом связном списке каждый узел содержит ссылку на следующий узел, а последний узел в списке содержит ссылку на первый узел (вместо `null`, как в стандартном связном списке, завершенном `null`).

4.3.52. *Обратная перестановка связного списка (итерационная).* Напишите нерекурсивную функцию, которая получает в аргументе первый узел `Node` связного списка и переставляет элементы в обратном порядке, возвращая первый узел `Node` результата.

4.3.53. *Обратная перестановка связного списка (рекурсивная).* Напишите рекурсивную функцию, которая получает в аргументе первый узел `Node` связного списка и переставляет элементы в обратном порядке, возвращая первый узел `Node` результата.

4.3.54. *Моделирование очереди.* Выясните, что произойдет, если изменить `MM1Queue` для использования стека вместо очереди. Выполняется ли закон Литтла? Ответьте на тот же вопрос для произвольной очереди. Постройте гистограммы и сравните среднеквадратичные отклонения времен ожидания.

4.3.55. *Моделирование распределения нагрузки.* Измените программу `LoadBalance`, чтобы она выводила среднюю и максимальную длину очереди вместо построения гистограммы. Используйте ее для проведения моделирования для 1 миллиона элементов и 100 000 очередей. Выведите среднее значение максимальной длины очереди для 100 испытаний с размерами выборки 1, 2, 3 и 4. Подтверждают ли ваши эксперименты заключения, приведенные в тексте, относительно использования размера выборки 2?

4.3.56. *Список файлов.* Папка представляет собой список файлов и папок. Напишите программу, которая получает в аргументе командной строки имя папки и выводит все файлы, содержащиеся в этой папке, с рекурсивным выводом содержимого каждой папки (с отступом) под ее именем. *Подсказка*: используйте очередь, см. описание `java.io.File`.

4.4. Таблицы символов

Таблица символов[1] (*таблица идентификаторов, ассоциативный массив, словарь*) представляет собой структуру данных, связывающую *ключи* со *значениями*. Клиент может поместить элемент в таблицу символов, указав пару «ключ — значение», а потом прочитать значение, связанное с заданным ключом. Например, университет может связать с номером социального страхования (ключ) такую информацию, как имя студента, домашний адрес и оценки (значение), чтобы к записи любого студента можно было обратиться по номеру социального страхования. Это решение также может пригодиться ученому, которому нужно структурировать данные; предприятию, желающему хранить информацию о клиентских операциях; поисковой системе, связывающей ключевые слова с веб-страницами, и в бесчисленном множестве других ситуаций[2].

В этом разделе рассматривается базовый API типа данных таблицы символов. Кроме операций чтения и записи, характерных для таблиц символов, наш API включает операции для проверки того, связано ли с заданным ключом какое-либо

---

[1] Традиционный термин. Имеются в виду «символы» в смысле, характерном для структуры программ (символическое имя, связанное с некоторым значением или адресом) и в теории трансляции (см. гл. 5). Другие варианты названий относятся к аналогичным структурам, но могут предполагать различие в контексте употребления и в деталях реализации. — *Примеч. науч. ред.*

[2] Продолжением этой концепции является табличное представление информации в *реляционных базах данных*. — *Примеч. науч. ред.*

значение (`contains`), для удаления ключа (и связанного с ним значения), определения количества пар «ключ — значение» в таблице символов (`size`) и перебора ключей в ней. Также будут рассмотрены другие порядковые операции с таблицами символов, естественным образом возникающие в различных ситуациях.

Мы рассмотрим два классических приложения — поиск в словаре и индексирование — и кратко обсудим возможности применения каждого из них на практике. Такие приложения превратились в фундаментальные инструменты, встречающиеся в той или иной форме в каждой вычислительной среде; их легко принять за нечто само собой разумеющееся и легко использовать неправильно. Очень важно, чтобы любой пользователь, использующий словарь или индекс, понимал, как он построен и как им пользоваться эффективно. Именно по этой причине мы подробно рассмотрим таблицы символов в этом разделе.

Из-за своей фундаментальной важности таблицы символов интенсивно использовались и становились предметом изучения с первых дней вычислительной теории. Мы рассмотрим две классические реализации. Первая реализация использует операцию *хеширования*, которая преобразует ключи в индексы массива, которые могут использоваться для обращения к значениям. Вторая реализация основана на структуре данных, называемой *деревом бинарного поиска* (*BST*, Binary Search Tree). Несмотря на свою чрезвычайную простоту, эти реализации лежат в основе промышленных реализаций таблиц символов, включаемых в современные среды программирования. Код, который мы рассматриваем для хеш-таблиц и деревьев бинарного поиска, лишь немногим сложнее кода со связным списком, который мы рассматривали для стеков и очередей; тем не менее он покажет вам новое измерение структурирования данных, имеющее далеко идущие последствия.

API

Таблица символов представляет собой коллекцию пар «ключ — значение». Обобщенный тип `Key` используется для ключей, а обобщенный тип `Value` — для значений: каждая запись в таблице символов связывает значение `Value` с ключом `Key`. Эти предположения приводят к следующему базовому API.

public class *ST<Key, Value>	
*ST()	создает пустую таблицу символов
void put(Key key, Value val)	связывает val с key
Value get(Key key)	значение, связанное с ключом key
void remove(Key key)	удаление ключа key (и связанного с ним значения)
boolean contains(Key key)	связано ли с ключом key какое-либо значение?
int size()	количество пар «ключ — значение»
Iterable<Key> keys()	все ключи в таблице символов

API обобщенной таблицы символов

Как обычно, звездочка (*) обозначает наличие нескольких реализаций. В этом разделе будут представлены две классические реализации: HashST и BST. (В тексте также будут кратко описаны некоторые элементарные реализации.) В этом API отражены некоторые решения из области проектирования, кратко перечисленные ниже.

Неизменяемые ключи

Предполагается, что ключи, находящиеся в таблице символов, не изменяют свои значения. Самые простые и наиболее часто используемые типы ключей, String и встроенные типы-обертки (такие, как Integer и Double), являются неизменяемыми.

Политика замены старого значения

Если пара «ключ — значение», вставляемая в таблицу символов, уже связывает с заданным ключом другое значение, то считается, что новое значение должно заменять старое (как при присваивании нового значения элементу массива). Применяя метод contains(), клиент может избежать этого, если он захочет.

Значение не найдено

Метод get() возвращает null, если с заданным ключом не связано никакое значение. У этого решения есть два следствия, указанные ниже.

Ключи и значения null

Клиент не может использовать null в качестве ключа или значения. Это правило позволяет реализовать contains() следующим образом:

```
public boolean contains(Key key)
{  return get(key) != null;  }
```

Удаление

Мы также включили в API метод для удаления ключа (и связанного с ним значения) из таблицы символов, потому что такой метод необходим во многих ситуациях. Однако реализация функции удаления будет для краткости отложена для упражнений или для курса алгоритмов и структур данных более высокого уровня.

Перебор пар «ключ — значение»

Метод keys() предоставляет клиентам возможность перебора пар «ключ — значение» в структуре данных. Для простоты он возвращает только ключи; клиент при желании может использовать get() для получения значения, связанного с ключом. Это позволяет использовать клиентский код следующего вида:

```
ST<String, Double> st = new ST<String, Double>();
...
for (String key : st.keys())
   StdOut.println(key + " " + st.get(key));
```

Хешируемые ключи

Java, как и многие другие языки, включает прямую языковую и системную поддержку реализаций таблиц символов. В частности, каждый объектный тип включает метод `equals()` (который может использоваться для проверки равенства двух ключей по правилам, определяемым типом данных ключа) и метод `hashCode()` (поддерживающий конкретную разновидность реализации таблиц символов, которая будет рассмотрена позднее в этом разделе).

Для стандартных типов данных, которые чаще всего используются для ключей, можно воспользоваться системными реализациями этих методов. С другой стороны, для типов данных, которые вы определяете сами, реализацию необходимо тщательно продумать (см. раздел 3.3). Многие программисты просто считают, что система автоматически предоставляет нужную реализацию, но при работе с нестандартными типами ключей необходимо действовать внимательно.

`public class *ST<Key extends Comparable<Key>, Value>`

	`*ST()`	создает пустую таблицу символов
`void`	`put(Key key, Value val)`	связывает val с key
`Value`	`get(Key key)`	значение, связанное с ключом key
`void`	`remove(Key key)`	удаление ключа key (и связанного с ним значения)
`boolean`	`contains(Key key)`	связано ли с ключом key какое-либо значение?
`int`	`size()`	количество пар «ключ — значение»
`Iterable<Key>`	`keys()`	все ключи в таблице символов
`Key`	`min()`	наименьший ключ
`Key`	`max()`	наибольший ключ
`int`	`rank(Key key)`	количество ключей, меньших заданного
`Key`	`select(int k)`	k-й по значению ключ (в порядке возрастания)
`Key`	`floor(Key key)`	наибольший ключ, меньший или равный `key`
`Key`	`ceiling(Key key)`	наименьший ключ, больший или равный `key`

API для упорядоченной таблицы символов

Ключи, реализующие Comparable

Во многих ситуациях ключи могут быть строками или другими типами данных с естественным порядком. В Java, как упоминалось в разделе 3.3, предполагается, что такие ключи реализуют интерфейс `Comparable`. Таблицы символов с ключами, поддерживающими `Comparable`, важны по двум причинам. Во-первых, мы можем воспользоваться упорядочением ключей для разработки реализаций `put` и `get`, что дает определенные гарантии быстродействия. Во-вторых, при реализации ключами интерфейса `Comparable` можно представить себе целое семейство новых операций (и обеспечить их поддержку). Например, клиент может запросить наи-

меньший ключ, наибольший ключ, медианный ключ или же перебрать все ключи в порядке сортировки. Полное изложение этой темы больше подходит для книги, посвященной алгоритмам и структурам данных, но в этом разделе вы познакомитесь с простой структурой данных, которая может легко поддерживать операции из неполного API, приведенного выше.

Таблицы символов принадлежат к числу наиболее изученных структур данных, поэтому влияние этих (и многих других альтернативных) решений было давно и подробно проанализировано, как вы узнаете в будущих учебных курсах информатики. В этом разделе для ознакомления с важнейшими свойствами таблиц символов мы рассмотрим две типичные клиентские программы, разработаем эффективные реализации двух классических решений и проанализируем характеристики быстродействия этих реализаций. Это убедит вас в том, что эти решения эффективно справляются с потребностями типичных клиентов, даже если им потребуется обработать очень большое количество ключей и значений.

Клиенты таблиц символов

Поближе познакомившись с концепцией таблиц символов, вы поймете, что они находят очень широкое применение. Чтобы убедить вас в этом, мы начнем с двух типичных примеров, которые встречаются во многих важных и хорошо знакомых практических ситуациях.

Поиск по словарю

В простейшем случае клиент последовательностью операций записи `put` строит таблицу символов, к которой будут обращаться с запросами чтения `get`. Следовательно, мы будем поддерживать коллекцию так, чтобы иметь возможность быстрого обращения к необходимым данным. Во многих случаях полезно также, чтобы словарь был *динамическим*, то есть предоставлял средства не только для быстрого поиска, но и для *изменения* содержимого таблицы. Следующий список хорошо знакомых примеров демонстрирует полезность этого подхода.

	ключ	**значение**
телефонная книга	имя	телефон
словарь	слово	определение
банковский счет	номер счета	баланс
геномика	кодон	аминокислота
данные	дата/время	результаты
компилятор Java	имя переменной	адрес в памяти
файлообменник	название песни	машина
система DNS в Интернете	сайт	IP-адрес

Типичные применения словарей

- *Телефонная книга.* Если ключами являются имена людей, а значениями — номера телефонов, таблица символов моделирует телефонную книгу. Принципиальное отличие от печатной телефонной книги заключается в том, что мы можем добавлять новые имена или изменять существующие номера телефонов. Также можно использовать номер телефона в качестве ключа, а имя в качестве значения.
- *Словарь.* Идея связывания слова с его определением хорошо знакома каждому пользователю. Веками люди держали у себя дома и на рабочем месте печатные словари, чтобы иметь возможность проверять определения и правильность написания (значения) слов (ключи). Сейчас же появилось много качественных реализаций таблиц символов, а встроенная проверка орфографии и немедленный доступ к определениям слов на компьютерах стали привычным явлением.
- *Информация о банковских счетах.* Владельцы акций регулярно проверяют их текущую цену в Интернете. Многие веб-службы связывают биржевое сокращение акций (ключ) с текущей ценой (значение), обычно предоставляя много другой информации (вспомните листинг 3.1.8). Существует немало коммерческих приложений такого рода; в частности, финансовые организации связывают информацию о счете с именем или номером счета, а образовательные организации предоставляют информацию об оценках по имени или идентификатору студента.
- *Геномика.* Таблицы символов играют важнейшую роль в современной геномике. Простейший пример — использование букв A, C, T и G для представления нуклеотидов, входящих в ДНК живых организмов. Пожалуй, следующим по простоте является соответствие между кодонами (триплетами нуклеотидов) и аминокислотами (TTA соответствует лейцину, TCT — серину и т. д.), затем соответствие между последовательностями аминокислот и протеинами и т. д. Ученые, ведущие исследования в области геномики, постоянно используют различные типы таблиц символов для структурирования этой информации.
- *Экспериментальные данные.* Современные ученые во всех областях, от астрофизики до ядерного синтеза, работают с огромными объемами экспериментальных данных. Организация этих данных и эффективное обращение к ним крайне важны для понимания их смысла. Таблицы символов становятся отправной точкой для работы с информацией, а нетривиальные структуры данных и алгоритмы, основанные на таблицах символов, стали важной частью научных исследований.
- *Языки программирования.* Одним из первых практических применений таблиц символов стала организация информации для программирования. Поначалу программы представляли собой простые последовательности чисел, но программисты очень быстро поняли, что использовать символические имена для обозначения операций и адресов памяти (имена переменных) намного удобнее. Для связывания имен с числами применялись таблицы символов. С ростом размера программмм затраты на работу с такими таблицами стали заметно влиять на время разработки программ. Это привело к разработке структур данных и алгоритмов вроде тех, которые будут рассмотрены в этом разделе.

- *Файлы.* Таблицы символов повсеместно используются для организации данных в компьютерных системах. Возможно, самым очевидным примером является *файловая система*: имя файла (ключ) связывается с информацией о его расположении (значение). Ваш музыкальный проигрыватель использует аналогичную систему для связывания названий песен (ключей) с местонахождением самих музыкальных дорожек (значения).
- *Система DNS в Интернете.* Система доменных имен (DNS), лежащая в основе организации информации в Интернете, связывает URL-адреса (ключи) в форме, удобной для людей (такие, как *www.princeton.edu* или *www.wikipedia.org*), с IP-адресами, понятными для сетевых маршрутизаторов (например, 208.216.181.15 или 207.142.131.206). По сути, эта система представляет собой «телефонную книгу» следующего поколения: люди используют легко запоминающиеся имена, а машины эффективно работают с числовыми данными. Количество операций поиска по таблицам символов, ежесекундно выполняемых маршрутизаторами Интернета по всему миру, огромно, поэтому необходимость эффективного выполнения операций сомнений не вызывает. Каждый год в Интернете появляются миллионы новых компьютеров и других устройств, поэтому таблицы символов в маршрутизаторах Интернета должны быть динамическими.

При всем разнообразии этот список — всего лишь типичные примеры, дающие представление о масштабах применения абстракции таблиц символов. Каждое ваше обращение к некоторым данным по имени — повод для применения таблицы символов. Вы видите работу файловой системы вашего компьютера или Всемирной паутины, но где-то «за кулисами» при этом трудится таблица символов.

Например, для построения таблицы символов, связывающей названия аминокислот с кодонами, пишется код следующего вида:

```
ST<String, String> amino;
amino = new ST<String, String>();
amino.put("TTA", "leucine");
...
```

Идея связывания информации с ключами настолько фундаментальна, что во многих высокоуровневых языках существует встроенная поддержка *ассоциативных массивов*: разработчик использует стандартный синтаксис операций с массивами, но вместо целочисленных индексов в квадратных скобках указываются ключи. В таких языках можно использовать запись `amino["TTA"] = "leucine"` вместо `amino.put("TTA", "leucine")`. И хотя в Java такой синтаксис (пока) не поддерживается, умение мыслить понятиями ассоциативных массивов помогает понять основное назначение таблиц символов.

Программа `Lookup` (листинг 4.4.1) строит множество пар «ключ — значение» на основании файла данных, разделенных запятыми (см. раздел 3.1), имя которого указано в командной строке; затем программа выводит значения, соответствующие ключам, прочитанным из стандартного потока ввода. В аргументах командной строки передаются имя файла и два целых числа, которые задают два поля: ключ и значение.

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
...
GCA,Ala,A,Alanine
GCG,Ala,A,Alanine
GAT,Asp,D,Aspartic Acid
GAC,Asp,D,Aspartic Acid
GAA,Gly,G,Glutamic Acid
GAG,Gly,G,Glutamic Acid
GGT,Gly,G,Glycine
GGC,Gly,G,Glycine
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine

% more DJIA.csv
...
20-Oct-87,1738.74,608099968,1841.01
19-Oct-87,2164.16,604300032,1738.74
16-Oct-87,2355.09,338500000,2246.73
15-Oct-87,2412.70,263200000,2355.09
...
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
...

% more ip.csv
...
www.ebay.com,66.135.192.87
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.cnn.com,64.236.16.20
www.google.com,216.239.41.99
www.nytimes.com,199.239.136.200
www.apple.com,17.112.152.32
www.slashdot.org,66.35.250.151
www.espn.com,199.181.135.201
www.weather.com,63.111.66.11
www.yahoo.com,216.109.118.65
...
```

Типичные файлы с данными, разделенными запятыми (CSV)

Листинг 4.4.1. Поиск по словарю

```
public class Lookup
{
   public static void main(String[] args)
   {  // Построение словаря и вывод значений для ключей из StdIn
      In in = new In(args[0]);
      int keyField = Integer.parseInt(args[1]);
      int valField = Integer.parseInt(args[2]);
      String[] database = in.readAllLines();
      StdRandom.shuffle(database);
      ST<String, String> st = new ST<String, String>();
      for (int i = 0; i < database.length; i++)
      {  // Чтение пары "ключ — значение" и добавление в st.
         String[] tokens = database[i].split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
      }
      while (!StdIn.isEmpty())
      {  // Чтение ключа и получение значения.
         String s = StdIn.readString();
         StdOut.println(st.get(s));
      }
   }
}
```

in	входной поток (.csv)
keyField	позиция ключа
valField	позиция значения
database[]	строки входных данных
st	таблица символов
tokens	значения в строке
key	ключ
val	значение
s	запрос

Клиент таблицы символов читает пары «ключ — значение» из файла с данными, разделенными запятыми, после чего выводит значения, соответствующие ключам из стандартного ввода. И ключи, и значения имеют тип String.

```
% java Lookup amino.csv 0 3
TTA
Leucine
ABC
null
TCT
Serine
% java Lookup amino.csv 3 0
Glycine
GGG
```

```
% java Lookup ip.csv 0 1
www.google.com
216.239.41.99
% java Lookup ip.csv 1 0
216.239.41.99
www.google.com
% java Lookup DJIA.csv 0 1
29-Oct-29
252.38
```

Загрузите с сайта книги файлы Lookup.java и ST.java (реализация таблицы символов коммерческого уровня, которая будет рассмотрена в конце раздела) для проведения поиска по таблицам символов. Также на сайте книги доступны многочисленные файлы с данными, разделенными запятыми (.csv), для различных ситуаций, описанных выше: amino.csv (соответствия кодонов и аминокислот), DJIA.csv (верхнее значение, объем сделок и нижнее значение биржевого индекса Доу-Джонса за каждый день в истории наблюдений) и ip.csv (выборка значений из базы данных DNS). Выбирая, какое поле должно использоваться в качестве ключа, помните, что *каждый ключ должен однозначно определять значение*. Если с таблицей было

выполнено несколько операций записи, связывающих разные значения с одним ключом, таблица символов сохраняет только последнее значение (аналогично ассоциативным массивам). Возможность связывания нескольких значений с одним ключом будет рассмотрена позднее.

Как будет показано позднее в этом разделе, затраты на выполнение операций чтения и записи в `Lookup` связаны логарифмической зависимостью с размером таблицы. Из этого факта следует, что получение ответа на первый запрос (и выполнение всех операций записи при построении таблицы) может сопровождаться небольшой задержкой, но на остальные запросы вы будете получать ответ немедленно.

Индексирование

Программа `Index` (листинг 4.4.2) — классический пример клиента таблицы символов, который чередует вызовы `get()` и `put()`: она читает последовательность строк из стандартного потока ввода и выводит отсортированный список различающихся строк вместе со списком целых чисел, указывающих позиции вхождения каждой строки во входных данных. Вы работаете с большим объемом данных и хотите знать, где встречается та или иная строка. На первый взгляд с каждым ключом связываются несколько значений, но на самом деле значение всего одно: очередь. Программа `Index` получает в качестве аргументов командной строки два целых числа, управляющих выводом: первое определяет минимальную длину «слова» — подстроки, включаемой в таблицу символов, а второе — минимальное количество вхождений (среди слов в тексте), при котором слово включается в выводимый указатель слов (индекс). Следующая подборка практических применений индексирования демонстрирует широту и многообразие его возможностей.

- *Алфавитный указатель в книге.* Во многих учебниках имеется алфавитный указатель, в котором по слову можно найти номера страниц, на которых встречается это слово. Поскольку ни один читатель не захочет видеть в алфавитном указателе все без исключения слова, такая программа, как `Index`, может стать хорошей отправной точкой для построения хорошего указателя.
- *Языки программирования.* Если в большой программе используется большое количество идентификаторов, полезно знать, где используется каждое имя. Такая программа, как `Index`, помогает программисту следить за тем, где в программе встречается тот или иной идентификатор. В современных средах программирования таблицы символов лежат в основе программных инструментов, которые используются программистами для управления идентификаторами в программах.
- *Геномика.* В типичном (разве что предельно упрощенном) сценарии исследований в области геномики ученый хочет знать позиции заданной генетической последовательности в существующем геноме или наборе геномов. Существование или близкое расположение некоторых последовательностей может иметь научную значимость. Отправной точкой для таких исследований становится индекс вроде того, который строится программой `Index`, измененный с учетом того обстоятельства, что геномы не разбиваются на слова.

Листинг 4.4.2. Индексирование

```
public class Index
{
   public static void main(String[] args)
   {
      int minlen = Integer.parseInt(args[0]);
      int minocc = Integer.parseInt(args[1]);

      // Создание и инициализация таблицы символов.
      ST<String, Queue<Integer>> st;
      st = new ST<String, Queue<Integer>>();
      for (int i = 0; !StdIn.isEmpty(); i++)
      {
         String word = StdIn.readString();
         if (word.length() < minlen) continue;
         if (!st.contains(word))
            st.put(word, new Queue<Integer>());
         Queue<Integer> queue = st.get(word);
         queue.enqueue(i);
      }

      // Вывод слов, у которых счетчик вхождений превышает порог.
      for (String s : st)
      {
         Queue<Integer> queue = st.get(s);
         if (queue.size() >= minocc)
            StdOut.println(s + ": " + queue);
      }
   }
}
```

`minlen`	минимальная длина
`minocc`	порог вхождений
`st`	таблица символов
`word`	текущее слово
`queue`	очередь позиций для текущего слова

Приложение-клиент таблицы символов индексирует текстовый файл по позициям слов. Ключами являются слова, а значениями — очереди позиций, в которых слово встречается в файле.

```
% java Index 9 30 < TaleOfTwoCities.txt
confidence: 2794 23064 25031 34249 47907 48268 48577 ...
courtyard: 11885 12062 17303 17451 32404 32522 38663 ...
evremonde: 86211 90791 90798 90802 90814 90822 90856 ...
...
something: 3406 3765 9283 13234 13239 15245 20257 ...
sometimes: 4514 4530 4548 6082 20731 33883 34239 ...
vengeance: 56041 63943 67705 79351 79941 79945 80225 ...
```

	ключ	**значение**
книга	термин	номера страниц
геномика	часть цепочки ДНК	позиции
веб-поиск	ключевое слово	веб-сайты
бизнес	имя клиента	операции

Типичные примеры индексирования

- *Веб-поиск.* Когда вы вводите ключевое слово и получаете список сайтов, содержащих это слово, вы используете индекс, построенный поисковой системой. Одно значение (список страниц) связывается с каждым ключом (запросом), хотя реальная ситуация чуть динамичнее и сложнее, потому что пользователь часто указывает сразу несколько ключей, а информация о страницах распределяется в Интернете (в отличие от хранения таблицы на одном компьютере).
- *Информация о клиентах.* В брокерских компаниях, управляющих портфелями акций своих клиентов, один из способов отслеживания ежедневных операций основан на ведении индекса списка операций. Ключом является номер портфеля, а значением — список вхождений этого номера в список операций.

Загрузите программу `Index` с сайта книги и выполните ее для различных входных файлов; она поможет вам лучше понять практичность таблиц символов. Вы увидите, что построение больших индексов для огромных файлов обходится без особых задержек, потому что каждая операция записи и запрос на чтение обрабатываются немедленно. Обеспечение немедленного отклика для огромных таблиц символов — один из классических примеров использования достижений в проектировании алгоритмов.

Элементарные реализации таблиц символов

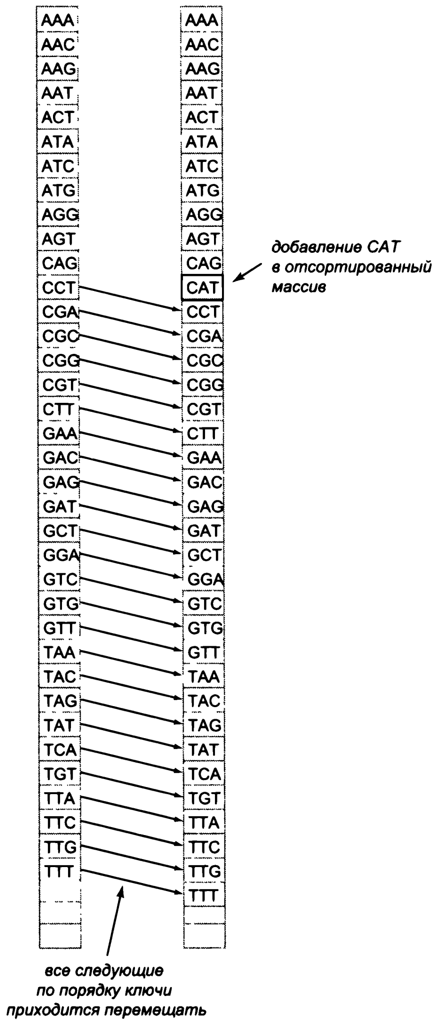
Все эти примеры убедительно демонстрируют важность таблиц символов. Реализации таблиц символов интенсивно изучались, для них были изобретены многие алгоритмы и структуры данных, и современные среды программирования (такие, как Java) включают одну или несколько их реализаций. Как обычно, понимание базовой реализации поможет вам оценить, выбрать и более эффективно использовать одну из более сложных или реализовать вашу собственную версию для какой-нибудь специальной ситуации, с которой вы можете столкнуться.

Для начала рассмотрим две элементарные реализации, основанные на двух базовых структурах данных: массивах с изменяемым размером и связном списке. Это делается только для того, чтобы убедить вас в необходимости более сложной структуры данных, поскольку реализации выполняют чтение и запись за линейное время, из-за чего они плохо подходят для реальных больших приложений.

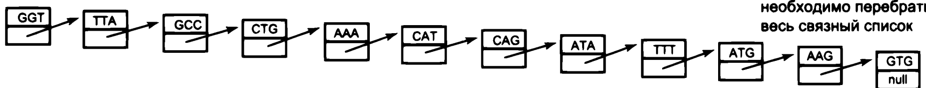
Пожалуй, простейшая реализация основана на хранении пар «ключ — значение» в неупорядоченном связном списке (или массиве) и применении *последовательного поиска* (см. упражнение 4.4.6). Последовательный поиск проверяет один за другим каждый узел (или элемент) последовательности, пока не будет найден заданный ключ или не кончатся все элементы списка (или массива). Для типичных приложений такая реализация неприемлема, потому что, например, при отсутствии ключа в таблице символов запрос на чтение элемента будет выполняться за линейное время[1].

---

[1] Строго говоря, при наличии ключа время выполнения запроса тоже будет линейным, но просматривать придется не весь список, а только его часть. Обычно принимается, что это в среднем половина длины списка. — *Примеч. науч. ред.*



Вставка в отсортированный массив выполняется за линейное время



Последовательный поиск в связном списке выполняется за линейное время

Также возможно другое решение: использовать отсортированный массив (с изменяемым размером) для ключей и параллельный массив для значений. Так как ключи хранятся в отсортированном порядке, вы можете провести поиск ключа (и связанного с ним значения) с использованием *бинарного поиска*, как в разделе 4.2. Реализовать таблицу символов на основе этого решения несложно (см. упражнение 4.4.5). В такой реализации поиск выполняется быстро (логарифмическое время), но вставка обычно оказывается медленной (линейное время) из-за необходимости поддерживать упорядоченность в массиве с изменяемым размером. Каждый раз, когда в массив добавляется новый ключ, все следующие по порядку ключи должны сдвигаться в массиве на одну позицию вперед; из этого следует, что в худшем случае запись занимает линейное время[1].

Чтобы реализовать таблицу символов, которая обеспечивала бы приемлемое быстродействие для таких клиентов, как `Lookup` и `Index`, нам понадобится структура данных более гибкая, чем просто связный список или массив с изменяемым размером. Рассмотрим два примера таких структур данных: *хеш-таблица* и *бинарное дерево поиска*.

Хеш-таблица

Хеш-таблица — структура данных, в которой ключи делятся на небольшие группы для ускорения поиска. Мы выбираем параметр m и делим ключи на m групп примерно одинакового размера. Для каждой группы ключи хранятся в неупорядоченном связном списке, в котором выполняется последовательный поиск, как в только что рассмотренной нами элементарной реализации.

Для деления ключей на m групп используется хеш-функция, которая связывает каждый возможный ключ с *хеш-значением* — целым числом от 0 до $m - 1$. Это позволяет представить таблицу символов в виде массива связных списков и использовать хеш-значение как индекс массива для обращения к нужному списку.

Хеширование чрезвычайно полезно, поэтому во многих языках программирования предусмотрена его прямая поддержка. Как было показано в разделе 3.3, для этого каждый класс Java должен содержать метод `hashCode()`. Если вы используете нестандартный тип, проверьте реализацию `hashCode()`, потому что реализация по умолчанию не всегда справляется с разбиением ключей на группы равного размера. Для преобразования хеш-кода в хеш-значение от 0 до $m - 1$ используется выражение `Math.abs(x.hashCode() % m)`.

Вспомните, о чем говорилось ранее: если два объекта равны (как сообщает метод `equals()`), они имеют одинаковые хеш-коды. Если объекты не равны, они тоже могут иметь одинаковые хеш-коды. Хеш-функции проектируются таким образом,

[1] Аналогично предыдущему замечанию — затраты времени линейны, но сдвиг затрагивает в среднем только половину массива. На практике часто бывает выгоднее допустить более медленную запись, поскольку типичные применения предполагают более частые запросы на чтение. — *Примеч. науч. ред.*

чтобы вызов `Math.abs(x.hashCode() % m)` с приблизительно равной вероятностью возвращал хеш-значения от 0 до $m - 1$.

ключ	хеш-код	хеш-значение
GGT	70516	1
TTA	83393	3
GCC	70375	0
CTG	67062	2
AAA	64545	0
CAT	66486	1
CAG	66473	2
ATA	65134	4
TTT	83412	2
ATG	65140	0
AAG	64551	1
GTG	70906	1

Хеш-коды и хеш-значения для n = 12 строк (m = 5)

В таблице приведены хеш-коды и хеш-значения для 12 типичных ключей `String` при $m = 5$. *Примечание*: обычно хеш-коды представляют собой целые числа от -2^{31} до $2^{31} - 1$, но для коротких алфавитно-цифровых строк они представляют собой короткие положительные целые числа.

После всей подготовки реализация эффективной таблицы символов с хешированием становится тривиальным расширением кода связных списков из раздела 4.3. Мы определяем массив с m связными списками, в котором элемент i содержит связный список всех ключей с хеш-значением i (вместе со значениями, ассоциированными с ключами). Алгоритм поиска ключа выглядит так.

- Вычислить хеш-значение для нахождения связного списка.
- Перебрать узлы связного списка, проверяя их на присутствие ключа.
- Если искомый ключ входит в связный список, вернуть связанное с ним значение; в противном случае вернуть `null`.

Чтобы вставить пару «ключ — значение»:

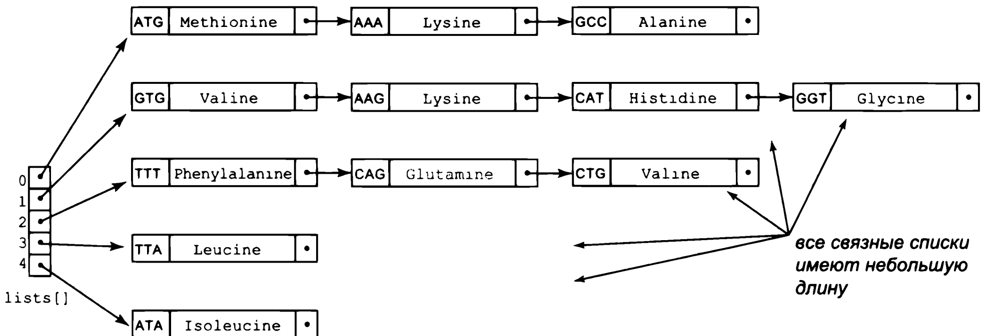
- вычислить хеш-значение для нахождения связного списка;
- перебрать узлы связного списка, проверяя их на присутствие ключа;
- если искомый ключ входит в связный список, заменить текущее связанное с ним значение новым; в противном случае создать новый узел с заданным ключом и значением и вставить его в начало связного списка.

Программа `HashST` (листинг 4.4.3) представляет собой полноценную реализацию для фиксированного числа связных списков $m = 1024$. Ее работа зависит от следующего вложенного класса, представляющего каждый узел связного списка:

```
private static class Node
{
   private Object key;
   private Object val;
   private Node next;

   public Node(Object key, Object val, Node next)
   {
      this.key  = key;
      this.val  = val;
      this.next = next;
   }
}
```

Эффективность `HashST` зависит от значения m и от качества хеш-функции. Если предположить, что хеш-функция обеспечивает хорошее распределение ключей, быстродействие примерно в m раз выше, чем у последовательного поиска в связном списке, при дополнительных затратах памяти на m ссылок и связных списков. Перед вами типичный компромисс между затратами памяти и времени: чем больше значение m, тем больше затраты памяти, но тем быстрее выполняется операция.

На следующей диаграмме изображена хеш-таблица, построенная для ключей из нашего примера, вставляемых в порядке на с. 616. Сначала CGT вставляется в связный список 1, затем TTA вставляется в связный список 3, затем GCC вставляется в список 0 и т. д. После того как хеш-таблица будет построена, поиск CAG начинается с вычисления хеш-значения (2) и последовательного поиска по связному списку 2. Ключ CAG обнаруживается во втором узле связного списка 2, и метод `get()` возвращает значение `Glutamine`.



Хеш-таблица (m=5)

Листинг 4.4.3. Таблица с хешированием

```
public class HashST<Key, Value>
{
   private int m = 1024;
   private Node[] lists = new Node[m];

   private class Node
   {  /* См. в тексте. */  }

   private int hash(Key key)
   {  return Math.abs(key.hashCode() % m); }

   public Value get(Key key)
   {
      int i = hash(key);
      for (Node x = lists[i]; x != null; x = x.next)
         if (key.equals(x.key))
            return (Value) x.val;
      return null;
   }

   public void put(Key key, Value val)
   {
      int i = hash(key);
      for (Node x = lists[i]; x != null; x = x.next)
      {
         if (key.equals(x.key))
         {
            x.val = val;
            return;
         }
      }
      lists[i] = new Node(key, val, lists[i]);
   }
}
```

m	количество связных списков
lists[i]	связный список для хеш-значения i

Программа использует массив связных списков для реализации таблицы символов с хешированием (хеш-таблицы). Хеш-функция выбирает один из m списков. Если таблица содержит n ключей, средние затраты на выполнение операции put() или get() составляют n/m (при соответствующей реализации hashCode()). Затраты на операцию будут фиксированными, если вы используете массив с изменяемым размером и примете меры к тому, чтобы среднее количество ключей на список составляло от 1 до 8 (см. упражнение 4.4.12). Реализации contains(), keys(), size() и remove() откладываются до упражнений 4.4.8–4.4.11.

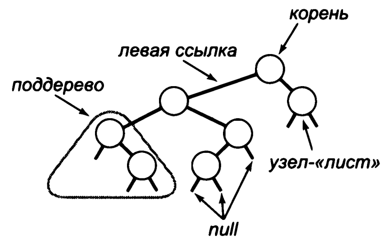
Часто программисты выбирают большое фиксированное значение m на основании приблизительной оценки количества ключей (как значение 1024, которое мы выбрали). Если приложить больше усилий, можно гарантировать, что среднее количество ключей на список является постоянной величиной; для этого для размещения `lists[]` используется массив с изменяемым размером. Например,

упражнение 4.4.12 показывает, как гарантировать, что среднее количество ключей в списке лежит в диапазоне от 1 до 8, что обеспечивает фиксированное (амортизированное) время выполнения операций чтения и записи. Конечно, в конкретной практической ситуации эти параметры всегда можно отрегулировать.

Главным преимуществом хеш-таблиц является эффективность выполнения операций чтения и записи. Недостаток заключается в том, что хеш-таблицы не упорядочивают ключи и, следовательно, не могут выдавать их отсортированными (или поддерживать другие операции, зависящие от порядка). Например, если заменить `ST` на `HashST` в программе `Index`, ключи будут выводиться в произвольном порядке вместо порядка сортировки. А если вы захотите получить наименьший или наибольший ключ, придется проводить полный поиск по всем ключам. Далее будет рассмотрена реализация таблицы, которая поддерживает порядковые операции при условии реализации ключами `Comparable` без особого ущерба для быстродействия `put()` и `get()`.

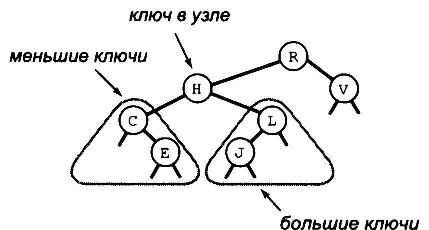
Бинарные деревья поиска

Бинарное дерево — математическая абстракция, играющая центральную роль в эффективном структурировании информации. Бинарное дерево определяется рекурсивно: оно представляет собой либо пустое дерево (`null`), либо узел, содержащий ссылки на два непересекающихся бинарных дерева. Бинарные деревья в программировании играют важную роль, потому что они обеспечивают эффективный баланс между гибкостью и простотой реализации. Бинарные деревья находят множество применений в науке, математике и информатике, и эта модель наверняка встретится вам во многих ситуациях.



Строение бинарного дерева

При описании бинарных деревьев часто используется терминология, знакомая нам по любым другим деревьям. Узел на вершине дерева называется *корнем*; узел, на который указывает левая ссылка, — *левым поддеревом*; а узел, на который указывает правая ссылка, — *правым поддеревом*. Традиционно деревья на диаграммах изображаются в перевернутом виде, то есть корень располагается наверху. Узлы, у которых обе ссылки равны `null`, называются *листовыми* узлами (или просто *листьями*). *Высота* дерева равна максимальному количеству ссылок среди всех путей от корня к листу.



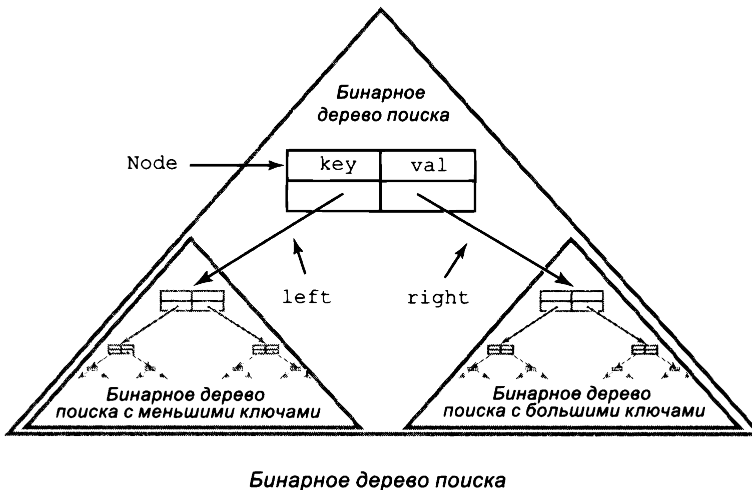
Симметричный порядок

Как и в случае с массивами, связными списками и хеш-таблицами, бинарные деревья используются для хранения коллекций данных. Для реализаций таблиц символов используется особая разновидность бинарного дерева, называемая *бинарным деревом поиска* (BST, Binary Search Tree). Бинарное дерево поиска представляет собой бинарное дерево, каждый узел которого содержит пару «ключ — значение», а ключи располагаются в симметричном порядке: ключ узла больше ключа любого узла своего левого поддерева и меньше ключа любого узла своего правого поддерева. Как вы вскоре увидите, симметричное упорядочение обеспечивает эффективные реализации операций чтения и записи.

Разработку реализации бинарного дерева поиска мы начнем с класса для абстракции узла. Этот класс содержит ссылки на ключ, значение, левое и правое поддерево. Тип ключа должен реализовать интерфейс `Comparable` (чтобы задать порядок ключей), но тип значения произволен.

```
class Node
{
   Key key;
   Value val;
   Node left, right;
}
```

Это определение напоминает наше определение узлов связных списков, кроме того, что класс содержит две ссылки вместо одной. Как и в случае со связными списками, идея рекурсивной структуры данных кажется довольно головоломной, но, по сути, мы просто добавляем вторую ссылку в свое определение связного списка (и задаем ограничения упорядоченности).



Бинарное дерево поиска

Чтобы (немного) упростить код, мы добавим конструктор `Node`, который инициализирует переменные экземпляров `key` и `val`:

```
Node(Key key, Value val)
{
   this.key = key;
   this.val = val;
}
```

Результатом выполнения `new Node(key, val)` является ссылка на объект `Node` (которая может быть присвоена любой переменной типа `Node`), переменным экземпляра которой присваиваются заданные значения, а переменные экземпляра `left` и `right` инициализируются `null`.

Как и в случае со связными списками, при трассировке кода с BST мы используем визуальное представление изменений.

- Каждый объект представляется прямоугольником.
- Значения переменных экземпляров размещаются в прямоугольниках.
- Ссылки обозначаются стрелками, указывающими на объекты.

Чаще всего используется еще более простое абстрактное представление, в котором узлы изображаются прямоугольниками (или кругами), внутри которых размещаются ключи (без значений), а ссылки изображаются стрелками. Это абстрактное представление позволяет сосредоточиться на структуре связей.

Для примера рассмотрим бинарное дерево поиска со строковыми ключами и целочисленными значениями. Чтобы построить бинарное дерево поиска из одного узла, связывающее значение 0 с ключом `it`, мы создаем объект `Node`:

```
Node first = new Node("it", 0);
```

Так как обе ссылки `left` и `right` содержат `null`, этот узел представляет бинарное дерево поиска из одного узла. Чтобы добавить узел, связывающий значение 1 с ключом `was`, следует создать другой объект `Node`:

```
Node second = new Node("was", 1);
```

(который сам по себе является бинарным деревом поиска) и связать его с полем `right` первого объекта `Node`:

```
first.right = second;
```

Второй узел размещается справа от первого, потому что в алфавитном порядке `was` следует после `it`. (Также можно было бы присвоить `second.left` ссылку `first`.) Теперь можно добавить третий узел, который связывает значение 2 с ключом `the`:

```
Node third = new Node("the", 2);
second.left = third;
```

и четвертый узел, который связывает значение 3 с ключом `best`:

```
Node fourth = new Node("best", 3);
first.left = fourth;
```

Обратите внимание: каждая из ссылок — `first`, `second`, `third` и `fourth` — по определению является бинарным деревом поиска (каждая равна `null` или содержит ссылку на бинарное дерево поиска, и в каждом узле выполняется условие упорядочения).

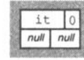
В текущем контексте мы следим за тем, чтобы при установлении связей между узлами каждый создаваемый объект `Node` являлся корнем бинарного дерева поиска (содержал ключ, значение, ссылку на левое бинарное дерево поиска с меньшими значениями и ссылку на правое бинарное дерево поиска с большими значениями). С точки зрения структуры данных бинарного дерева поиска значение несущественно, поэтому на диаграммах оно часто игнорируется, но мы включаем его в определение, потому что оно играет важнейшую роль в концепции таблицы символов.

Бинарное дерево поиска представляет упорядоченную последовательность элементов. В только что рассмотренном примере `first` представляет последовательность `best it the was`. Для представления последовательности элементов можно использовать массив; например, строка

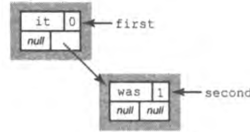
```
String[] a = { "best", "it", "the", "was" };
```

определяет ту же упорядоченную последовательность строк. Для заданного множества различных ключей существует только один способ представления их в виде упорядоченного массива, но много способов представления их в виде бинарного дерева поиска (см. упражнение 4.4.7). Эта гибкость позволяет разрабатывать эффективные реализации таблиц символов. Например, в нашем примере для вставки новой пары «ключ — значение» было достаточно создать новый узел и изменить всего одну ссылку. Как выясняется, это всегда можно сделать. Что не менее важно, вы всегда можете легко найти в бинарном дереве поиска узел с заданным ключом или узел, ссылку которого необходимо изменить при вставке новой пары «ключ — значение». Ниже будет рассмотрен код реализации таблицы символов, который решает обе задачи.

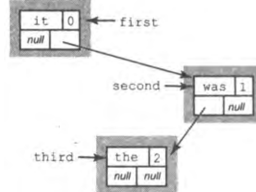
```
Node first  = new Node("it", 0);
```



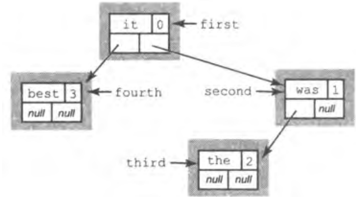

```
Node second = new Node("was", 1);
first.right = second;
```




```
Node third= new Node("the", 2);
second.left = third;
```




```
Node fourth = new Node("best", 2);
first.left = fourth;
```

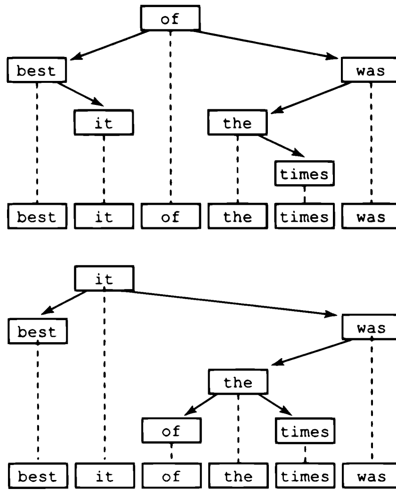


Связывание узлов бинарного дерева поиска

Поиск

Допустим, вы хотите найти в бинарном дереве узел с заданным ключом (или прочитать значение, связанное с заданным ключом, из таблицы символов). Возможны

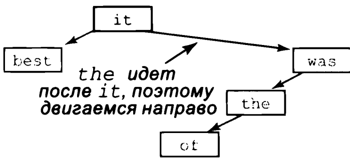
два результата: поиск может быть успешным (ключ найден в бинарном дереве поиска; возвращается связанное с ним значение) или неуспешным (в бинарном дереве поиска нет узла с заданным ключом; возвращается `null`).

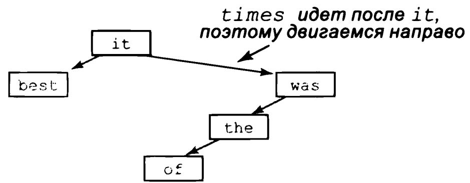


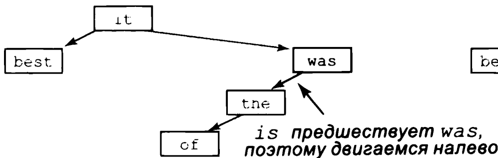
Два бинарных дерева поиска, представляющих одну последовательность

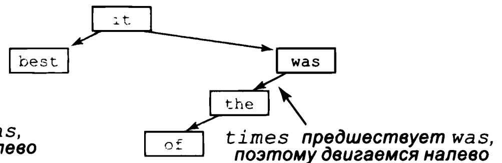
успешный поиск узла с ключом `the`

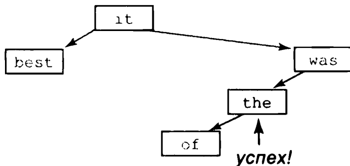
неуспешный поиск узла с ключом `times`

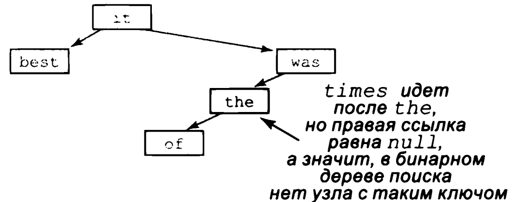








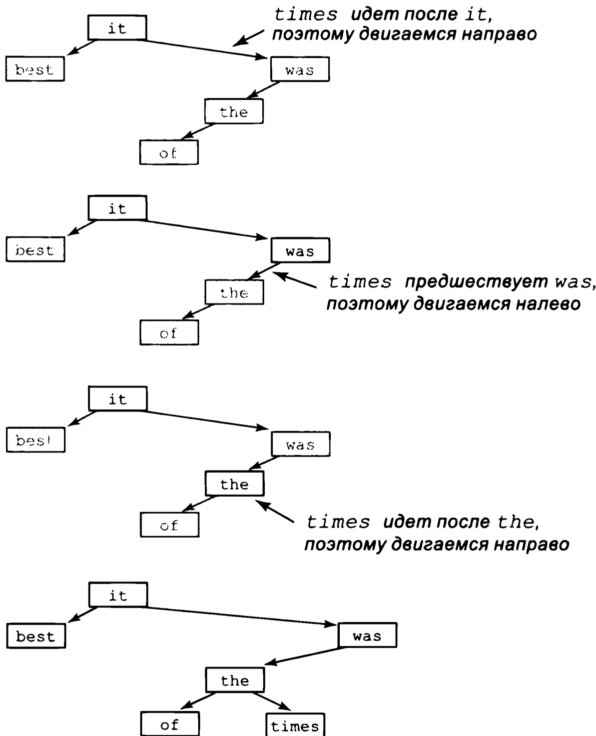




Поиск в бинарном дереве поиска

Рекурсивный алгоритм поиска очевиден: для заданного бинарного дерева поиска (ссылка на Node) нужно сначала проверить, является ли дерево пустым (ссылка равна null). Если дерево пустое, то поиск завершается как безуспешный (возвращается null). Если дерево не пустое, следует проверить, равен ли ключ в узле ключу поиска. Если ключи равны, то поиск завершается как успешный (возвращается значение, связанное с ключом). В противном случае ключ поиска сравнивается с ключом в узле. Если ключ поиска меньше, он ищется (рекурсивно) в левом поддереве; если он больше, то поиск (рекурсивно) продолжается в правом поддереве.

Если рассматривать происходящее с точки зрения рекурсии, нетрудно убедиться в том, что этот алгоритм работает так, как было задумано; в основе доказательства лежит инвариант — ключ находится в бинарном дереве поиска в том и только в том случае, если он находится в текущем поддереве. Важнейшее свойство рекурсивного метода заключается в том, что для принятия решения о том, что делать дальше, достаточно всего одного узла. Более того, обычно при поиске проверяется лишь небольшое количество узлов дерева: каждый раз, когда мы переходим к одному из поддеревьев узла, мы никогда не просматриваем никакие узлы другого поддерева.

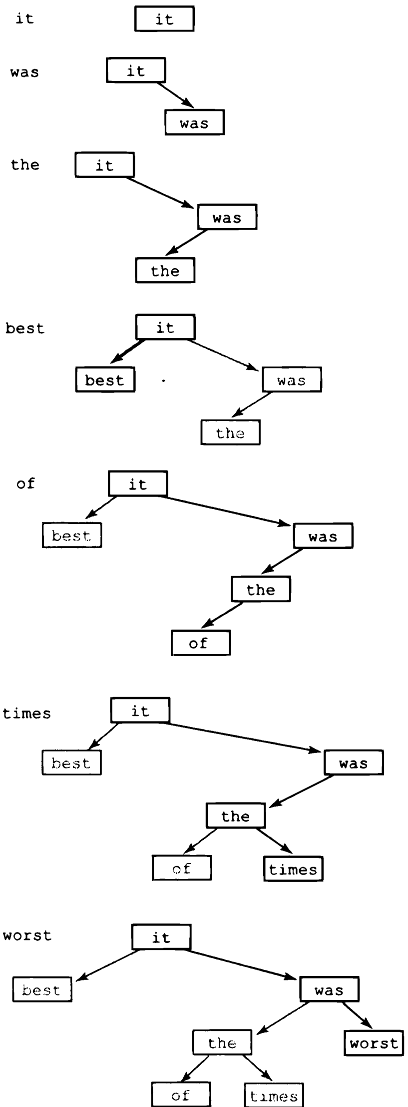


Вставка нового узла в бинарное дерево поиска

Вставка

Допустим, в бинарное дерево поиска вставляется новый узел (в реализации таблицы символов в структуру данных включается новая пара «ключ – значение»). Логика вставки напоминает логику поиска ключа, но реализация несколько сложнее. Чтобы понять ее, следует осознать, что для вставки достаточно изменить только одну ссылку, причем это именно та `null`-ссылка, которая будет найдена при неуспешном поиске этого ключа.

Если дерево пустое, мы создаем и возвращаем новый объект `Node`, содержащий пару «ключ – значение»; если ключ поиска меньше, чем ключ корневого узла, то левой ссылке присваивается результат вставки пары «ключ – значение» в левое поддерево; если ключ поиска больше, то правой ссылке присваивается результат вставки пары «ключ – значение» в правое поддерево; наконец, если ключи равны, то текущее значение заменяется новым. Сброс значения левой или правой ссылки после рекурсивного вызова обычно оказывается излишним, потому что ссылка изменяется только при пустом поддереве, но проще присвоить значение, чтобы избежать лишних проверок.



Построение бинарного дерева поиска

Реализация

Программа `BST` (листинг 4.4.4) содержит реализацию таблицы символов на базе этих двух рекурсивных алгоритмов. Если вы сравните этот код с реализацией бинарного поиска `BinarySearch` (листинг 4.2.3), а также реализациями стека и очереди `Stack` (листинг 4.3.4) и `Queue` (листинг 4.3.6), вы оцените элегантность и простоту этого кода. Постарайтесь мыслить с рекурсивной точки зрения и убедитесь в том, что этот код работает так, как задумано. Пожалуй, для этого проще всего отследить процесс заполнения изначально пустого бинарного дерева поиска некоторым множеством ключей. Если вы понимаете, как это происходит, значит, вы понимаете эту фундаментальную структуру данных.

Листинг 4.4.4. Бинарное дерево поиска

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;

   private class Node
   {
      Key key;
      Value val;
      Node left, right;
      Node(Key key, Value val)
      {  this.key = key; this.val = val;  }
   }

   public Value get(Key key)
   {  return get(root, key);  }

   private Value get(Node x, Key key)
   {
      if (x == null) return null;
      int cmp = key.compareTo(x.key);
      if      (cmp < 0) return get(x.left,  key);
      else if (cmp > 0) return get(x.right, key);
      else              return x.val;
   }

   public void put(Key key, Value val)
   {  root = put(root, key, val);  }

   private Node put(Node x, Key key, Value val)
   {
      if (x == null) return new Node(key, val);
      int cmp = key.compareTo(x.key);
      if      (cmp < 0) x.left  = put(x.left,  key, val);
      else if (cmp > 0) x.right = put(x.right, key, val);
      else              x.val = val;
      return x;
   }
}
```

root	корень бинарного дерева поиска
key	ключ
val	значение
left	левое поддерево
right	правое поддерево

Реализация типа данных таблицы символов основана на рекурсивной структуре данных BST и рекурсивных методах для ее обхода. Реализации contains(), size() и remove() рассматриваются в упражнениях 4.4.18–4.4.20. Реализация keys() приведена в конце этого раздела.

Кроме того, методы `put()` и `get()` бинарного дерева поиска чрезвычайно эффективны: обычно каждый из них обращается к небольшому количеству узлов бинарного дерева поиска (к узлам на пути от корня до искомого узла или `null`-ссылки, заменяемой ссылкой на новый узел). Далее, видно, что операции записи и чтения завершаются за логарифмическое время (при выполнении некоторых условий).

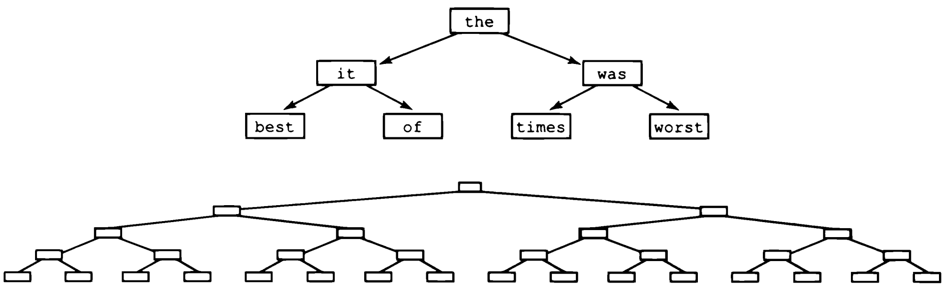
Кроме того, реализация `put()` создает только один новый объект `Node` и добавляет одну ссылку. Вы легко убедитесь в этом, построив диаграмму создания бинарного дерева поиска со вставкой нескольких ключей в изначально пустое дерево — каждый узел просто добавляется где-то у нижнего края дерева.

Характеристики быстродействия бинарных деревьев поиска

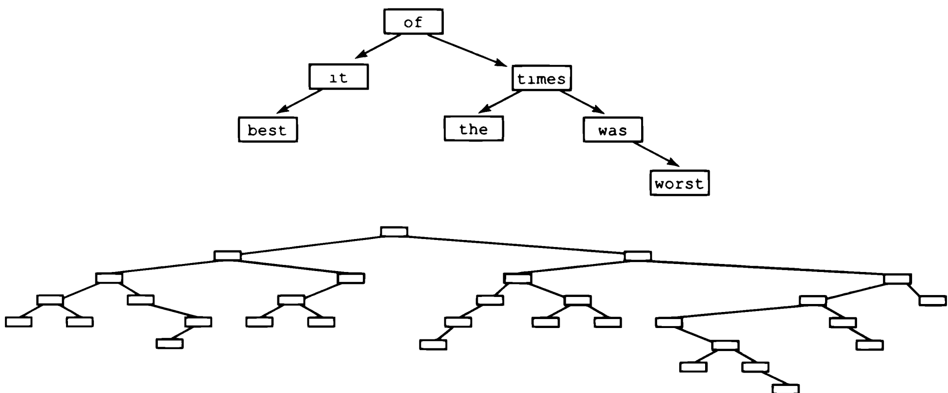
Время выполнения алгоритмов бинарных деревьев поиска в конечном итоге зависит от формы дерева, а форма дерева зависит от порядка вставки ключей. Понимание этой зависимости — критический фактор возможности эффективного использования бинарных деревьев поиска в реальных ситуациях.

Лучший случай

В лучшем случае дерево идеально сбалансировано (у каждого объекта `Node` ровно два потомка, отличных от `null`), а расстояние от корня до каждого листового узла



Идеально сбалансированное бинарное дерево поиска (лучший случай)

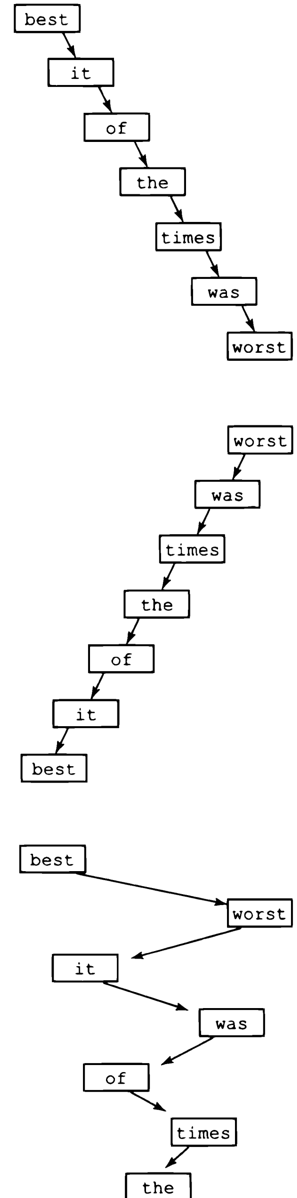


Типичные бинарные деревья поиска, построенные для ключей со случайным порядком

составляет ровно lg n ссылок. В таких деревьях сразу понятно, что время безуспешного поиска является логарифмическим, потому что эти затраты удовлетворяют тому же рекуррентному отношению, что и затраты на бинарный поиск (см. раздел 4.2), так что затраты на каждую операцию чтения и записи пропорциональны lg n или ниже. На практике такие идеально сбалансированные деревья при последовательной вставке ключей встречаются разве что при большом везении, но характеристики быстродействия для лучшего случая знать полезно.

Средний случай

Если в дерево вставляются случайные ключи, можно ожидать, что время поиска также будет логарифмическим, потому что первый ключ становится корнем дерева, а ключи будут разделены приблизительно пополам. Применяя тот же аргумент к поддеревьям, можно рассчитывать на получение того же результата, что и для лучшего случая. Эти интуитивные представления подтверждаются тщательным анализом: классический математический анализ показывает, что время, необходимое для выполнения операций чтения и записи в дереве, построенном для ключей со случайным порядком, будет логарифмическим (за дополнительной информацией обращайтесь на сайт книги). А если говорить точнее, предполагаемое количество сравнений ключей равно примерно ~2 ln n для случайной записи или чтения в дереве, построенном из n случайно упорядоченных ключей. В практическом примере (таком, как `Lookup`), когда мы можем явно рандомизировать порядок ключей, этого результата достаточно для (вероятностных) гарантий логарифмического быстродействия. В самом деле, поскольку значение 2 ln n равно приблизительно 1,39 lg n, время среднего случая всего на 39% больше лучшего случая. В таких примерах, как `Index`, где управлять порядком вставки невозможно, таких гарантий нет, но типичные данные обеспечивают логарифмическое быстродействие (см. упражнение 4.4.26). Как и при бинарном поиске, этот факт чрезвычайно важен из-за огромного разрыва между линейным и логарифмическим временем: при реализации таблиц символов на базе бинарного дерева поиска мы можем выполнять миллионы операций в секунду (и более) даже в очень больших таблицах.



Бинарные деревья поиска для худшего случая

Худший случай

В худшем случае каждый узел (кроме одного) содержит ровно одну ссылку `null`, так что бинарное дерево поиска фактически представляет собой связный список с одной неиспользуемой ссылкой, когда операции чтения и записи выполняются за линейное время. К сожалению, этот худший случай нередко встречается на практике — например, он встречается при вставке ключей по порядку.

Таким образом, для хорошего быстродействия базовой реализации BST требуется, чтобы ключи были достаточно близки к случайным, тогда дерево не содержит много длинных путей. Если вы не уверены в том, что это предположение выполняется, *не используйте простое бинарное дерево поиска*. О возникших проблемах будет свидетельствовать только одно: замедление отклика при росте размера задачи (и программы, страдающие этим, вовсе не редкость). (*Примечание*: некоторые разновидности бинарных деревьев поиска исключают этот худший случай и гарантируют логарифмическое быстродействие на операцию, для чего все деревья приводятся к практически идеально сбалансированному состоянию. Одна из популярных разновидностей такого рода — *красно-черное дерево*.)

Обход бинарного дерева поиска

Вероятно, простейшей функцией обработки дерева является *обход дерева*: для дерева, заданного ссылкой на его корень, нужно обработать последовательно все его узлы. Для связных списков эта задача решается переходом по единственной ссылке от узла к узлу. Однако обход бинарного дерева требует принятия решений, потому что узел содержит *две* ссылки. На помощь приходит рекурсия. Обработка каждого ключа в бинарном дереве поиска выполняется по следующей схеме.

- Обработать каждый узел левого поддерева.
- Обработать корневой узел.
- Обработать каждый узел правого поддерева.

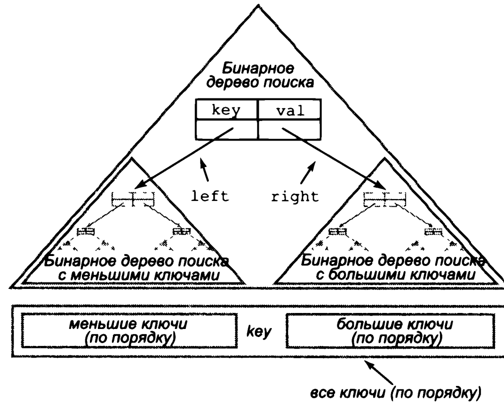
Этот подход называется *симметричным* (inorder) обходом дерева, и его не стоит путать с префиксным (начиная с корня дерева) или постфиксным (завершая корнем дерева) обходом, встречающимся в других приложениях.

Для заданного бинарного дерева поиска вы можете легко убедиться посредством математической индукции, что этот процесс не только обрабатывает каждый узел бинарного дерева поиска, но и делает это в порядке сортировки ключей. Например, следующий метод выводит ключи бинарного дерева поиска с корнем, переданным в качестве аргумента, в порядке возрастания ключей узлов:

```
private void traverse(Node x)
{
   if (x == null) return;
   traverse(x.left);
```

```
    StdOut.println(x.key);
    traverse(x.right);
}
```

Сначала выводятся все ключи левого поддерева в порядке сортировки ключей. Затем выводится корень, следующий за ними в порядке сортировки, после чего выводятся все ключи правого поддерева, также в порядке сортировки ключей.



Рекурсивный симметричный обход бинарного дерева поиска

Этот на удивление простой метод заслуживает более тщательного изучения. Его можно взять за основу реализации `toString()` для бинарных деревьев поиска (см. упражнение 4.4.21). Он также служит основой для реализации метода `keys()`, позволяющего клиентам использовать цикл Java *foreach* для перебора ключей бинарного дерева поиска в порядке сортировки (вспомните, что эта функциональность недоступна в хеш-таблицах, в которых порядок отсутствует). Это фундаментальное применение симметричного порядка обхода будет рассмотрено ниже.

Перебор ключей

Только что рассмотренный рекурсивный метод `traverse()` позволяет обработать все пары «ключ — значение» в нашем типе данных `BST`. Для простоты можно ограничиться только ключами, потому что значения всегда можно получить при необходимости. Теперь наша цель — реализация метода `keys()`, с которым станет возможным клиентский код следующего вида:

```
BST<String, Double> st = new BST<String, Double>();
...
for (String key : st.keys())
    StdOut.println(key + " " + st.get(key));
...
```

Программа `Index` (листинг 4.4.2) служит другим примером клиентского кода, использующего цикл *foreach* для перебора пар «ключ — значение».

В простейшей реализации `keys()` все ключи собираются в коллекцию с поддержкой перебора (например, `Stack` или `Queue`), которая и возвращается клиенту.

```
public Iterable<Key> keys()
{
   Queue<Key> queue = new Queue<Key>();
   inorder(root, queue);
   return queue;
}

private void inorder(Node x, Queue<Key> queue)
{
   if (x == null) return;
   inorder(x.left, queue);
   queue.enqueue(x.key);
   inorder(x.right, queue);
}
```

Когда вы впервые видите обход дерева, он кажется чем-то сверхъестественным. Упорядоченный перебор фактически становится бесплатным приложением к структуре данных, спроектированной для быстрого поиска и вставки. Заметим, что аналогичный прием (то есть сбор ключей в коллекции с поддержкой перебора) может использоваться и для реализации метода `keys()` класса `HashST` (см. упражнение 4.4.10), но при этом порядок ключей окажется произвольным, так как хеш-таблицы не предполагают упорядочивания ключей.

Порядковые операции с таблицами символов

Гибкость бинарных деревьев поиска и возможность сравнения ключей позволяют реализовать многие полезные операции помимо тех, которые могут эффективно поддерживаться для хеш-таблиц. Ниже перечислены лишь наиболее типичные действия; для бинарных деревьев поиска было создано много других важных операций, широко применяемых в приложениях. Мы оставим реализации этих операций для упражнений, а дальнейшее изучение характеристик быстродействия и практического применения можно продолжить в учебном курсе алгоритмов и структур данных.

Минимум и максимум

Чтобы найти наименьший ключ в бинарном дереве поиска, переходите от корня по левым ссылкам, пока не будет достигнута ссылка `null`. Последний ключ на этом пути и является наименьшим во всем бинарном дереве поиска. Аналогичная процедура с переходом по правым ссылкам приведет к наибольшему ключу в бинарном дереве поиска (см. упражнение 4.4.27).

Размер дерева и размеры поддеревьев

Чтобы отслеживать количество узлов в бинарном дереве поиска, включите в `BST` дополнительную переменную экземпляра `n`, в которой хранится количество узлов в дереве. Инициализируйте ее 0 и увеличивайте при каждом создании нового объ-

екта `Node`. Также возможно другое решение: включить в каждый объект `Node` дополнительную переменную экземпляра для подсчета количества узлов в поддереве, корнем которого является данный узел (см. упражнение 4.4.29).

Диапазонный поиск и подсчет элементов в диапазоне

С таким рекурсивным методом, как `inorder()`, можно вернуть объект `Iterable` для ключей, находящихся в диапазоне между двумя заданными значениями, за время, пропорциональное высоте бинарного дерева поиска и количества ключей в диапазоне (см. упражнение 4.4.31). Если хранить в каждом узле переменную экземпляра с размером поддерева с корнем в этом узле, подсчет ключей между двумя заданными значениями также можно выполнить за время, пропорциональное высоте дерева (см. упражнение 4.4.31).

Статистика и ранги

Если хранить в каждом узле переменную экземпляра с размером поддерева с корнем в этом узле, можно реализовать рекурсивный метод, который возвращает k-й наименьший ключ за время, пропорциональное высоте дерева (см. упражнение 4.4.55). Также можно вычислить *ранг* ключа — количество ключей в бинарном дереве поиска, строго меньших заданного (см. упражнение 4.4.56).

В дальнейшем мы будем использовать эталонную реализацию `ST`, которая реализует API упорядоченной таблицы символов на базе класса Java `java.util.TreeMap` — реализации таблицы символов на базе красно-черных деревьев. Дополнительную информацию о красно-черных деревьях можно получить на курсах структур данных и алгоритмов повышенного уровня. В частности, они предоставляют гарантии логарифмического времени для `get()`, `put()` и многих других из описанных операций.

Множества

В завершение рассмотрим тип данных, более простой по сравнению с таблицей символов, широко применяемый на практике и легко реализуемый на базе хеш-таблиц или бинарных деревьев поиска. *Множество* представляет собой коллекцию уникальных ключей, фактически таблицу символов без значений. Конечно, можно использовать `ST` и игнорировать значения, но клиентский код, использующий следующий API, получается более простым и ясным.

Как и в случае с таблицами символов, никакие внутренние аспекты реализации не требуют поддержки `Comparable`. Тем не менее обработка ключей с поддержкой сравнения вполне типична, и она позволяет выполнять различные порядковые операции, поэтому мы включили `Comparable` в API. Реализация `SET` посредством удаления всех упоминаний `val` в коде `BST` достаточно тривиальна (см. упражнение 4.4.23). Также легко разрабатывается реализация `SET` на базе хеш-таблицы.

```
public class SET<Key extends Comparable<Key>>
```

`SET()`	создает пустое множество
`boolean isEmpty()`	множество пусто?
`void add(Key key)`	добавляет ключ key в множество
`void remove(Key key)`	удаляет ключ key из множества
`boolean contains(Key key)`	ключ key встречается в множестве?
`int size()`	количество элементов в множестве

Примечание: реализации также должны реализовать интерфейс Iterable<Key>, чтобы клиенты могли обращаться к ключам в цикле foreach.

API обобщенного множества

Программа `DeDup` (листинг 4.4.5) — клиент `SET`, который читает последовательность строк из стандартного ввода и выводит только первое вхождение каждой строки (то есть из потока удаляются повторы). Другие примеры клиентов `SET` встречаются в упражнениях в конце этого раздела.

В следующем разделе важность этой фундаментальной абстракции будет продемонстрирована в контексте практического примера.

Перспектива

Реализации таблиц символов принадлежат к числу первоочередных тем для дальнейшего изучения алгоритмов и структур данных. Примеров хватает: сбалансированные бинарные деревья поиска, хеширование, префиксные деревья... Реализации многих алгоритмов и структур данных встречаются в Java и многих других вычислительных средах. Разные API и разные предположения о ключах требуют разных реализаций. Исследователи алгоритмов и структур данных продолжают изучать реализации всевозможных таблиц символов.

Какая реализация таблицы символов лучше — хеширование или бинарные деревья поиска? Прежде всего следует выяснить, поддерживают ли клиентские ключи сравнение и нужны ли клиенту порядковые операции с таблицами символов, такие как выбор и ранжирование. Если ответ «да», значит, нужно использовать бинарные деревья поиска. Если «нет», то большинство программистов остановится на хешировании, потому что таблицы на основе хеш-таблиц обычно работают быстрее, чем на основе бинарных деревьев поиска (если у вас есть хорошая хеш-функция для ключа).

Применение бинарных деревьев поиска для реализации таблиц символов и множеств — превосходный пример использования абстракции дерева, повсеместно распространенной и знакомой. Мы привыкли к древовидным структурам в повседневной жизни: генеалогические деревья, спортивные турниры, структурные схемы организаций, деревья синтаксического анализа в грамматике. Примеры деревьев также часто встречаются в организации вычислений, включая деревья вызовов функций, деревья разбора языков программирования или файловые системы.

Листинг 4.4.5. Фильтр Dedup

```
public class DeDup
{
   public static void main(String[] args)
   {  // Устранение повторяющихся строк
      SET<String> distinct = new SET<String>();
      while (!StdIn.isEmpty())
      {  // Чтение строки; если это дубликат - строка игнорируется.
         String key = StdIn.readString();
         if (!distinct.contains(key))
         {  // Сохранение и вывод новой строки.
            distinct.add(key);
            StdOut.print(key);
         }
         StdOut.println();
      }
   }
}
```

distinct	левый конец
key	правый конец

Клиент SET — фильтр, который читает строки из стандартного ввода и записывает строки в стандартный вывод, игнорируя повторяющиеся строки. Для обеспечения эффективности используется коллекция SET с уникальными строками, встречавшимися до настоящего момента.

```
% java DeDup < TaleOfTwoCities.txt
it was the best of times worst age wisdom foolishness...
```

Многие важные применения деревьев происходят из научных и инженерных областей; среди них можно выделить филогенетические деревья в вычислительной биологии, многомерные деревья в компьютерной графике, минимаксные деревья в экономике, деревья квадрантов в моделях молекулярной динамики. Также встречаются другие, более сложные связные структуры, как будет показано в разделе 4.5.

Словари, алфавитные указатели и другие разновидности таблиц символов используются людьми ежедневно. За непродолжительное время приложения, основанные на таблицах символов, заменили телефонные книги, энциклопедии и многие другие физические артефакты, которые верно служили нам в прошлом тысячелетии. Без реализаций таблиц символов, основанных на таких структурах данных, как хеш-таблицы и бинарные деревья поиска, подобные приложения были бы невозможны; с ними у пользователя возникает ощущение, что любую необходимую информацию можно найти в Сети мгновенно.

Вопросы и ответы

В. Почему нужно использовать неизменяемые ключи таблиц символов?

О. Изменение ключа, находящегося в хеш-таблице или бинарном дереве поиска, может привести к нарушению инвариантности структуры данных.

В. Почему переменная экземпляра `val` во вложенном классе `Node` из `HashST` объявляется с типом `Object` вместо `Value`?

О. Хороший вопрос. К сожалению, как было показано в разделе «Вопросы и ответы» в конце раздела 3.1, Java не позволяет создавать массивы обобщений. Одно из последствий этого ограничения — то, что в методе `get()` приходится выполнять преобразование типа, которое порождает предупреждение на стадии компиляции (несмотря на то что преобразование заведомо пройдет успешно во время выполнения). При этом переменную экземпляра `val` во вложенном классе `Node` из `BST` можно объявить с типом `Value`, потому что она не использует массивы.

В. Почему бы не использовать библиотеки Java для работы с таблицами символов?

О. Теперь, когда вы понимаете, как работают таблицы символов, ничто не мешает вам использовать коммерческие реализации `java.util.TreeMap` и `java.util.HashMap`. Они имеют такой же базовый API, как у `ST`, но позволяют использовать `null`-ключи и используют имена `containsKey()` и `keySet()` вместо `contains()` и `iterator()` соответственно. Кроме того, они содержат ряд вспомогательных методов, но при этом не поддерживают некоторые упоминаемые здесь методы (например, требующие упорядочивания сведений о содержимом таблицы). Также можно использовать реализации `java.util.TreeSet` и `java.util.HashSet` с API, как у нашей реализации `SET`.

Упражнения

4.4.1. Модифицируя программу `Lookup`, создайте программу `LookupAndPut`, которая позволяет задавать операции через стандартный потока ввода. Знак «+» означает, что следующие две строки определяют пару «ключ — значение» для вставки.

4.4.2. Модифицируя программу `Lookup`, создайте программу `LookupMultiple`, которая позволяет задать несколько значений с одним ключом. Для этого все значения сохраняются в очереди, как в `Index`, а затем выводятся по запросу:

```
% java LookupMultiple amino.csv 3 0
Leucine
TTA TTG CTT CTC CTA CTG
```

4.4.3. Модифицируя программу `Index`, создайте программу `IndexByKeyword`, которая получает имя файла из командной строки и строит индекс для содержимого стандартного ввода, используя только ключевые слова из файла. *Примечание*: при использовании одного и того же файла и для индексирования, и для списка ключевых слов результат должен быть таким же, как в программе `Index`.

4.4.4. Модифицируя программу `Index`, создайте программу `IndexLines`, которая считает ключами только непрерывные последовательности букв (без знаков препинания или цифр) и использует в качестве значения номер строки вместо позиции слова. Эта функциональность может пригодиться при программировании:

```
% java IndexLines 6 0 < Index.java
continue 12
enqueue 15
Integer 4 5 7 8 14 .
parseInt 4 5
println 22
```

4.4.5. Разработайте для API таблицы символов реализацию `BinarySearchST`, которая поддерживает параллельные массивы ключей и значений, храня их элементы в порядке сортировки ключей. Используйте бинарный поиск для чтения, перемещайте большие пары «ключ — значение» на одну позицию вправо при записи (используйте массив с изменяемым размером, чтобы длина массива была пропорциональна количеству пар «ключ — значение» в таблице). Протестируйте свою реализацию с `Index` и проверьте гипотезу о том, что с такой реализацией выполнение `Index` занимает время, пропорциональное произведению количества строк и количества уникальных строк во входных данных.

4.4.6. Разработайте для API таблицы символов реализацию `SequentialSearchST`, которая поддерживает связный список узлов, содержащих ключи и значения; узлы следуют в произвольном порядке. Протестируйте свою реализацию с `Index` и проверьте гипотезу о том, что с такой реализацией выполнение `Index` занимает время, пропорциональное произведению количества строк и количеству уникальных строк во входных данных.

4.4.7. Вычислите выражение `x.hashCode() % 5` для односимвольных строк

```
E A S Y Q U E S T I O N
```

Возьмите за образец диаграмму, приведенную в тексте, и постройте хеш-таблицу, созданную при связывании `i`-го ключа этой последовательности со значением `i`, для `i` от 0 до 11.

4.4.8. Реализуйте метод `contains()` для `HashST`.

4.4.9. Реализуйте метод `size()` для `HashST`.

4.4.10. Реализуйте метод `keys()` для `HashST`.

4.4.11. Модифицируйте программу `HashST`, добавив метод `remove()`, который получает аргумент `Key` и удаляет заданный ключ (с соответствующим значением) из таблицы символов, если этот ключ существует.

4.4.12. Доработайте программу `HashST`, чтобы в ней использовался массив с изменяющимся размером, а средняя длина списка, связанного с каждым хеш-значением, лежала в диапазоне от 1 до 8.

4.4.13. Изобразите бинарное дерево поиска, создаваемое при вставке ключей

```
E A S Y Q U E S T I O N
```

в указанном порядке в изначально пустое дерево. Какова высота полученного бинарного дерева поиска?

4.4.14. Допустим, бинарное дерево поиска содержит целочисленные ключи от 1 до 1000 и вы ищете ключ 363. Какая из следующих последовательностей *не может* быть последовательностью просмотренных ключей?

```
a.  2 252 401 398 330 363
b.  399 387 219 266 382 381 278 363
c.  3 923 220 911 244 898 258 362 363
d.  4 924 278 347 621 299 392 358 363
e.  5 925 202 910 245 363
```

4.4.15. Допустим, следующие ключи встречаются (в некотором порядке) в бинарном дереве поиска высоты 4:

```
10 15 18 21 23 24 30 30 38 41 42 45 50 55 59
60 61 63 71 77 78 83 84 85 86 88 91 92 93 94 98
```

Изобразите три верхних узла дерева (корень и два его потомка).

4.4.16. Постройте все различные бинарные деревья поиска, которые могут представлять последовательность ключей

```
best of it the time was
```

4.4.17. True или false: пусть для заданного бинарного дерева поиска x — узел-«лист», а p — его родитель. Тогда либо (1) ключ `p` является наименьшим ключом бинарного дерева поиска, большим ключа x, либо (2) ключ `p` является наибольшим ключом бинарного дерева поиска, меньшим ключа x.

4.4.18. Реализуйте метод `contains()` для `BST`.

4.4.19. Реализуйте метод `size()` для `BST`.

4.4.20. Добавьте в `BST` метод `remove()`, который получает аргумент `Key` и удаляет этот ключ (и соответствующее значение) из таблицы символов, если он существует. *Подсказка*: замените ключ (и связанное с ним значение) следующим наибольшим ключом из `BST` (и связанным с ним значением); затем удалите из `BST` узел, который содержал следующий наибольший ключ.

4.4.21. Реализуйте метод `toString()` для `BST`, используя рекурсивный вспомогательный метод (такой, как `traverse()`). Как обычно, следует предполагать квадратичное быстродействие из-за затрат на конкатенацию строк. *Дополнительное задание*: напишите для `BST` метод `toString()` с линейным временем, использующий класс `StringBuilder`.

4.4.22. Доработайте API таблицы символов так, чтобы таблица поддерживала множественные значения с одинаковыми ключами; для этого метод `get()` должен возвращать реализацию `Iterable` для значений с заданным ключом. Реализуйте `BST` и `Index` в соответствии с этим API. Проанализируйте достоинства и недостатки этого решения по сравнению с описанным в тексте.

4.4.23. Доработайте `BST` для реализации API `SET`, приведенного в конце раздела.

4.4.24. Доработайте `HashST` для реализации API `SET`, приведенного в конце раздела (исключите из API ограничение `Comparable`).

4.4.25. *Словоуказателем*, или *конкорданцией*, называется алфавитный список всех слов в тексте (наборе текстов) с указанием позиций, в которых встречается каждое слово. Таким образом, вызов `java Index 0 0` строит словоуказатель. Известный пример: одна из групп исследователей Кумранских рукописей («свитков Мертвого моря») прибегла к опубликованию словоуказателя текстов для подтверждения достоверности материалов при сохранении в тайне их подробностей. Напишите программу `InvertConcordance`, которая получает в командной строке аргумент n, читает словоуказатель из стандартного ввода и выводит первые n слов соответствующего (восстановленного) текста в стандартный поток вывода.

4.4.26. Проведите эксперименты для проверки утверждений о том, что операции чтения и записи для `Lookup` и `Index` при использовании `ST` связаны с размером таблицы логарифмической зависимостью. Разработайте тестовые приложения-клиенты, которые генерируют случайные ключи, а также проведите тестирование для разных наборов данных (возьмите с сайта книги или выберите самостоятельно).

4.4.27. Модифицируйте программу `BST`, добавив методы `min()` и `max()`, которые возвращают наименьший (или наибольший) ключ в таблице. При отсутствии ключа возвращается `null`.

4.4.28. Модифицируйте программу `BST`, добавив методы `floor()` и `ceiling()`, которые получают в качестве аргумента ключ и возвращают наибольший (наименьший) ключ в таблице символов, не больший (не меньший) заданного ключа. Если такой ключ не существует, возвращается `null`.

4.4.29. Модифицируйте программу `BST`, добавив метод `size()`, который возвращает количество пар «ключ — значение» в таблице символов. Используйте решение, при котором в каждом объекте `Node` хранится количество узлов в поддереве с этим корнем.

4.4.30. Модифицируйте программу `BST`, добавив метод `rangeSearch()`, который получает два аргумента `Key` и возвращает итератор (реализацию `Iterable`) для перебора всех ключей, находящихся в диапазоне между двумя заданными ключами. Время выполнения должно быть пропорционально сумме высоты дерева и количества ключей в диапазоне.

4.4.31. Модифицируйте программу `BST`, добавив метод `rangeCount()`, который получает в аргументах два ключа и возвращает количество ключей в `BST`, находящихся в диапазоне между двумя заданными ключами. Время выполнения должно быть пропорционально высоте дерева. *Подсказка*: начните с предыдущего упражнения.

4.4.32. Напишите приложение-клиент для `ST`, которое создает таблицу символов, связывающую буквенные обозначения оценок (учащихся) с их числовыми эквивалентами по приведенной ниже таблице, а затем читает из стандартного потока ввода список оценок и вычисляет их среднее числовое значение.

```
 A+   A    A-   B+   B    B-   C+   C    C-   D    F
4.33 4.00 3.67 3.33 3.00 2.67 2.33 2.00 1.67 1.00 0.00
```

Упражнения для бинарных деревьев

Эти упражнения помогут вам приобрести опыт работы с бинарными деревьями, которые не обязательно являются бинарными деревьями поиска. Во всех упражнениях предполагается существование класса Node с тремя переменными экземпляров: положительным значением double и двумя ссылками на Node. Как и в случае со связными списками, полезными будут рисунки наподобие приводившихся в тексте.

4.4.33. Реализуйте следующие методы, каждый из которых получает в качестве аргумента объект `Node`, являющийся корнем бинарного дерева.

`int size()`	количество узлов в дереве
`int leaves()`	количество узлов, у которых обе ссылки равны null
`double total()`	сумма значений ключей по всем узлам

Все методы должны выполняться за линейное время.

4.4.34. Реализуйте метод `height()` с линейным временем, который возвращает максимальное количество ссылок на любом пути от корня к листу (высота дерева, состоящего из одного узла, равна 0).

4.4.35. Бинарное дерево называется *пирамидально-упорядоченным*, если ключ корня больше ключа любого из его потомков. Реализуйте метод `heapOrdered()` с линейным временем, который возвращает `true`, если дерево является пирамидально-упорядоченным, или `false` в противном случае.

4.4.36. Бинарное дерево называется сбалансированным, если оба его поддерева сбалансированы, а их высота отличается не более чем на 1. Реализуйте метод `balanced()` с линейным временем, который возвращает `true`, если дерево является сбалансированным, или `false` в противном случае.

4.4.37. Два бинарных дерева называются *изоморфными*, если различаются только значения ключей (а структура одинакова). Реализуйте статический метод `isomorphic()` с линейным временем, который получает в аргументах две ссылки на деревья и возвращает `true`, если эти ссылки указывают на изоморфные деревья, или `false` в противном случае. Затем реализуйте статический метод `eq()` с линейным временем, который получает в аргументах две ссылки на деревья и возвращает `true`, если эти ссылки указывают на идентичные деревья (изоморфные деревья с одинаковыми значениями ключей), или `false` в противном случае.

4.4.38. Реализуйте метод `isBST()` с линейным временем, который возвращает `true`, если дерево является бинарным деревом поиска, или `false` в противном случае.

Решение: эта задача немного сложнее, чем кажется на первый взгляд. Используйте перегруженный рекурсивный метод `isBST()`, который получает два дополнительных аргумента `lo` и `hi` и возвращает `true`, если дерево является бинарным деревом поиска, а все его значения лежат в диапазоне от `lo` до `hi`; для представления как наименьшего, так и наибольшего возможного ключа используйте `null`.

```
public static boolean isBST()
{  return isBST(root, null, null);  }
private boolean isBST(Node x, Key lo, Key hi)
{
   if (x == null) return true;
   if (lo != null && x.key.compareTo(lo) <= 0) return false;
   if (hi != null && x.key.compareTo(hi) >= 0) return false;
   if (!isBST(x.left, lo, x.key))   return false;
   if (!isBST(x.right, x.key, hi))  return false;
}
```

4.4.39. Напишите метод `levelOrder()`, который выводит ключи бинарного дерева поиска в порядке уровней: сначала выводится корень, затем узлы на один уровень ниже корня (слева направо); затем узлы на два уровня ниже корня (слева направо) и т. д. *Подсказка*: используйте `Queue<Node>`.

4.4.40. Вычислите значение, возвращаемое функцией `mystery()` для некоторых бинарных деревьев. Сформулируйте гипотезу относительно поведения этой функции и докажите ее.

```
public int mystery(Node x)
{
   if (x == null) return 0;
   return mystery(x.left) + mystery(x.right);
}
```

Ответ: возвращает 0 для любого бинарного дерева.

Упражнения повышенной сложности

4.4.41. *Проверка орфографии.* Напишите приложение-клиент для `SET` с именем `SpellChecker`, которое получает в качестве аргумента командной строки имя файла, содержащего словарь, а затем читает строки из стандартного ввода и выводит все строки, отсутствующие в словаре. Файл словаря можно найти на сайте книги. Дополнительное задание: доработайте свою программу, чтобы она правильно обрабатывала стандартные суффиксы английского языка (такие, как *-ing* или *-ed*).

4.4.42. *Исправление орфографии.* Напишите приложение-клиент для `ST` с именем `SpellCorrector`, которое заменяет слова с распространенными ошибками из стандартного ввода рекомендуемой заменой с выводом результата в стандартный вывод. В качестве аргумента командной строки передается имя файла с распространенными ошибками и способами их замены. Пример такого файла можно найти на сайте книги.

4.4.43. *Веб-фильтр.* Напишите приложение-клиент для `SET` с именем `WebBlocker`, которое получает в качестве аргумента командной строки имя файла со списком подозрительных сайтов, а затем читает строки из стандартного ввода и выводит только сайты, не входящие в список.

4.4.44. *Операции с множествами.* Добавьте в `SET` методы `union()` и `intersection()`. Эти методы получают в качестве аргументов два множества и возвращают (соответственно) объединение и пересечение этих двух множеств.

4.4.45. *Таблица символов для подсчета событий.* Разработайте тип данных `FrequencyTable` с поддержкой операций (методов) `click()` и `count()` с аргументами строкового типа. Тип данных отслеживает количество вызовов операции `click()` с передачей в качестве аргумента заданной строки. Операция `click()` увеличивает счетчик на 1, а операция `count()` возвращает значение счетчика (которое может быть нулевым). Примеры приложений, использующих этот тип данных, — анализатор веб-трафика; музыкальный плеер, подсчитывающий количество воспроизведений каждой песни; телефонная программа с подсчетом звонков и т. д.

4.4.46. *Одномерный диапазонный поиск.* Разработайте тип данных с поддержкой следующих операций: вставка даты, поиск даты, подсчет количества хранимых в структуре дат, лежащих в заданном интервале. Для работы с датами используйте тип данных Java `java.util.Date`.

4.4.47. *Неперекрывающийся интервальный поиск.* Для заданного списка неперекрывающихся интервалов целых чисел напишите функцию, которая получает целочисленный аргумент и определяет, в каком интервале лежит это значение (если такой интервал есть). Например, если заданы интервалы 1643–2033, 5532–7643, 8999–10332 и 5666653–5669321, точка 9122 лежит в третьем интервале, а точка 8122 не принадлежит никакому интервалу.

4.4.48. *Определение страны по IP-адресу.* Напишите клиент `BST`, который использует файл данных ip-to-country.csv (доступен на сайте книги) для определения страны, к которой относится заданный IP-адрес. Файл данных содержит пять полей: начало диапазона IP-адресов, конец диапазона IP-адресов, двухсимвольный код страны, трехсимвольный код страны и название страны. IP-адреса не перекрываются. Такая база данных может применяться для выявления мошенничества с кредитными картами, фильтрации спама, автоматического выбора языка на сайте и анализа журнала веб-сервера.

4.4.49. *Инвертированный (обратный) индекс.* Для заданного списка веб-страниц создайте таблицу слов, содержащихся в этих веб-страницах. Свяжите с каждым словом список веб-страниц, в которых встречается это слово. Напишите программу, которая читает список веб-страниц, создает таблицу символов и по запросу, содержащему одно слово, возвращает список веб-страниц, в которых встречается это слово.

4.4.50. *Инвертированный (обратный) индекс.* Усовершенствуйте предыдущий пример, чтобы он поддерживал запросы из нескольких слов. На этот раз выводите список веб-страниц, в которых каждое слово из запроса встречается хотя бы один раз.

4.4.51. *Поиск по нескольким словам.* Напишите программу, которая получает k слов из командной строки, читает последовательность слов из стандартного ввода и находит наименьший фрагмент текста, содержащий все k слов (не обязательно в том же порядке). Рассматривать неполные слова не нужно.

Подсказка: для каждого индекса i найдите наименьший интервал $[i, j]$, содержащий k слов из запроса. Подсчитайте количество вхождений каждого из k слов. Для заданного $[i, j]$ вычислите $[I + 1, j']$, уменьшив счетчик для слова i. Затем постепенно увеличивайте j, пока интервал не будет содержать хотя бы один экземпляр каждого из k слов (или эквивалентное слово i).

4.4.52. *Ничья из-за повторения позиции в шахматах.* Если в шахматной партии позиция на доске трижды повторяется и при этом очередность хода принадлежит одной и той же стороне, то эта сторона может объявить ничью. Опишите, как бы вы проверили это условие в компьютерной программе.

4.4.53. *Планирование расписаний.* Недавно секретарь факультета в одном известном университете на северо-востоке страны назначил преподавателя на проведение двух разных учебных курсов в одно и то же время. Помогите секретарю предотвратить подобные ошибки в будущем — опишите метод выявления таких конфликтов. Для простоты считайте, что все занятия продолжаются 50 минут и начинаются в 9, 10, 11, 1, 2 и 3 часа.

4.4.54. *Случайный элемент.* Добавьте в `BST` метод `random()`, который возвращает случайный ключ (присутствующий в дереве). Храните в каждом узле размер поддерева (см. упражнение 4.4.29). Время выполнения должно быть пропорционально высоте дерева.

4.4.55. *Данные с учетом упорядочивания.* Добавьте в `BST` метод `select()`, который получает целочисленный аргумент k и возвращает k-й наименьший ключ из `BST`. Храните в каждом узле размер поддерева (см. упражнение 4.4.29). Время выполнения должно быть пропорционально высоте дерева.

4.4.56. *Ранжирование.* Добавьте в `BST` метод `rank()`, который получает целочисленный аргумент `key` и возвращает целое число `i` — номер ключа `key` среди наименьших ключей бинарного дерева поиска в порядке возрастания. Храните в каждом узле размер поддерева (см. упражнение 4.4.29). Время выполнения должно быть пропорционально высоте дерева.

4.4.57. *Расширенная очередь.* Реализуйте класс, который поддерживает следующий API гибрида стека и очереди, с поддержкой удаления i-го по порядку добавления элемента (см. упражнение 4.3.40):

`public class GeneralizedQueue<Item>`	
`GeneralizedQueue()`	создает пустую расширенную очередь
`boolean isEmpty()`	расширенная очередь пуста?
`void add(Item item)`	вставляет item в расширенную очередь
`Item remove(int i)`	извлекает и возвращает i-й из последних вставленных элементов
`int size()`	количество элементов в очереди

API обобщенной расширенной очереди

Используйте бинарное дерево поиска, связывающее k-й по порядку добавления в дерево элемент с ключом k и хранящее в каждом узле размер поддерева с корнем в этом узле. Чтобы найти i-й по порядку добавления элемент, найдите в бинарном дереве поиска i-й в порядке возрастания ключ.

4.4.58. *Разреженные векторы.* d-мерный вектор называется *разреженным*, если количество ненулевых значений в нем относительно мало. Ваша цель — найти представление вектора с затратами памяти, пропорциональными количеству ненулевых элементов, и реализовать возможность сложения двух разреженных векторов за время, пропорциональное общему количеству ненулевых элементов. Реализуйте класс, поддерживающий следующий API.

public class SparseVector		
	SparseVector()	создает вектор
void	put(int i, double v)	присваивает a_i значение v
double	get(int i)	возвращает a_i
double	dot(SparseVector b)	скалярное произведение векторов
SparseVector	plus(SparseVector b)	сумма векторов

API разреженного вектора с элементами double

4.4.59. *Разреженные матрицы.* Матрица $n \times n$ называется *разреженной*, если количество ненулевых элементов в ней пропорционально n (или ниже). Ваша задача — найти представление матрицы с затратами памяти, пропорциональными n, и реализовать возможность сложения и умножения двух разреженных матриц за время, пропорциональное общему количеству ненулевых элементов (возможно, с дополнительным множителем $\log n$). Реализуйте класс, поддерживающий следующий API.

public class SparseMatrix		
	SparseMatrix()	создает матрицу
void	put(int i, int j, double v)	присваивает a_{ij} значение v
double	get(int i, int j)	возвращает a_{ij}
SparseMatrix	plus(SparseMatrix b)	сложение матриц
SparseMatrix	times(SparseMatrix b)	умножение матриц

API разреженной матрицы с элементами double

4.4.60. *Очередь без повторов.* Создайте тип данных очереди, в которой любой элемент может встречаться не более одного раза. Если элемент уже находится в очереди, запрос на его вставку игнорируется.

4.4.61. *Изменяемая строка.* Создайте тип данных, поддерживающий следующий строковый API. Используйте ST для реализации всех операций за логарифмическое время.

public class MutableString	
MutableString()	создает пустую строку.
char get(int i)	возвращает i-й символ в строке
void insert(int i, char c)	вставляет c и делает его i-м символом
void delete(int i)	удаляет i-й символ
int length()	возвращает длину строки

API изменяемой строки

4.4.62. *Команды присваивания.* Напишите программу для разбора и выполнения программ, состоящих из команд присваивания и вывода, с арифметическими выражениями с полным набором круглых скобок (см. листинг 4.3.5). Например, для входных строк

```
A = 5
B = 10
C = A + B
D = C * C
print(D)
```

ваша программа должна выводить значение `225`. Предполагается, что все переменные и значения относятся к типу `double`. Для отслеживания имен переменных используйте таблицу символов.

4.4.63. *Энтропия.* Относительная энтропия текстового корпуса из n слов, k из которых различны, вычисляется по формуле

$$E = 1/(n \lg n)\ (p_0 \lg(k/p_0) + p1 \lg(k/p_1) + \ldots + p_{k-1} \lg(k/p_{k-1})),$$

где p_i — доля вхождений (частота появлений) слова i. Напишите программу, которая читает текстовый корпус и выводит относительную энтропию. Преобразуйте все буквы к нижнему регистру и интерпретируйте знаки препинания как пропуски.

4.4.64. *Динамическое дискретное распределение.* Создайте тип данных, поддерживающий две операции: `add()` и `random()`. Метод `add()` добавляет новый элемент в структуру данных, если он еще не встречался ранее; в противном случае его счетчик увеличивается на 1. Метод `random()` возвращает случайный элемент, при этом вероятность выбора каждого из элементов пропорциональна величине его счетчика. Храните в каждом узле размер поддерева (см. упражнение 4.4.29). Время выполнения должно быть пропорционально высоте дерева.

4.4.65. *Портфель акций.* Реализуйте методы `buy()` и `sell()` в программе `StockAccount` (листинг 3.2.8). Используйте таблицу символов для хранения количества акций каждого вида.

4.4.66. *Таблица использования кодонов.* Напишите программу, которая использует таблицу символов для вывода статистики по каждому кодону в геноме, прочитан-

ном из стандартного потока ввода (частота — количество вхождений на тысячу), в следующем виде:

```
UUU 13.2  UCU 19.6  UAU 16.5  UGU 12.4
UUC 23.5  UCC 10.6  UAC 14.7  UGC  8.0
UUA  5.8  UCA 16.1  UAA  0.7  UGA  0.3
UUG 17.6  UCG 11.8  UAG  0.2  UGG  9.5
CUU 21.2  CCU 10.4  CAU 13.3  CGU 10.5
CUC 13.5  CCC  4.9  CAC  8.2  CGC  4.2
CUA  6.5  CCA 41.0  CAA 24.9  CGA 10.7
CUG 10.7  CCG 10.1  CAG 11.4  CGG  3.7
AUU 27.1  ACU 25.6  AAU 27.2  AGU 11.9
AUC 23.3  ACC 13.3  AAC 21.0  AGC  6.8
AUA  5.9  ACA 17.1  AAA 32.7  AGA 14.2
AUG 22.3  ACG  9.2  AAG 23.9  AGG  2.8
GUU 25.7  GCU 24.2  GAU 49.4  GGU 11.8
GUC 15.3  GCC 12.6  GAC 22.1  GGC  7.0
GUA  8.7  GCA 16.8  GAA 39.8  GGA 47.2
```

4.4.67. *Уникальные подстроки длины k.* Напишите программу, которая получает целое число k как аргумент командной строки, читает текст из стандартного потока ввода и вычисляет количество содержащихся в нем уникальных подстрок длины k. Например, если на ввод подается строка `CGCCGGGCGCG`, она содержит 5 уникальных подстрок длины 3: `CGC`, `CGG`, `GCG`, `GGC` и `GGG`. Эти вычисления могут использоваться для сжатия данных. *Подсказка*: используйте метод `substring(i,i+k)` для извлечения i-й подстроки и вставки ее в таблицу символов. Протестируйте свою программу на большом геноме с сайта книги и первых 10 миллионах цифр числа π.

4.4.68. *Случайные телефонные номера.* Напишите программу, которая получает целое число n как аргумент командной строки и выводит n случайных телефонных номеров в форме *(xxx) xxx-xxxx*. Используйте `SET` для того, чтобы избежать повторной выдачи одного телефонного номера. Используйте только действительные коды зон (файл с такими кодами можно найти на сайте книги).

4.4.69. *Проверка паролей.* Напишите программу, которая получает `String` в аргументе командной строки, читает словарь (набор слов) из стандартного ввода и проверяет, является ли аргумент командной строки «сильным» паролем. В данном случае под «сильным» подразумевается, что пароль:

1) имеет длину не менее 8 символов;

2) не является одним из слов в словаре;

3) не является словом из словаря, за которым следует цифра 0–9 (например, `hello5`);

4) не является двумя словами, разделенными цифрой (например, `hello2world`), и

5) ни одно из условий 2–4 не выполняется для слов из словаря, прочитанных в обратном порядке.

4.5. Пример: феномен «тесного мира»

Математическая модель, которую мы используем для изучения природы попарных соединений между сущностями, называется *графом*. Графы исключительно важны для изучения окружающего мира; кроме того, они помогают нам лучше понять и уточнить сетевые структуры, которые мы создаем. От моделей нервной системы в нейробиологии, от изучения распространения инфекционных заболеваний до прокладки телефонных сетей — графы сыграли важнейшую роль в науке и технике в прошлом веке, включая разработку самого Интернета.

Некоторые графы обладают особым свойством, называемым феноменом «тесного мира». Возможно, вы знакомы с этим свойством, которое иногда называют *шестью ступенями разделения*, или *правилом шести* (*пяти*) *рукопожатий*. Основная идея заключается в том, что у каждого из нас знакомых относительно немного, однако любого человека можно связать с любым другим по относительно короткой цепочке знакомых. Эта гипотеза была экспериментально проверена Стэнли Милгрэмом в 1960-е годы, а в 1990-е годы Дункан Уоттс и Стивен Строгац построили для нее математическую модель. За последние годы выяснилось, что этот принцип играет важную роль во многих приложениях. Графы «тесного мира» представляют интерес для ученых, потому что они моделируют природные явления, а инженеры заинтересованы в построении сетей, использующих естественные свойства графов «тесного мира».

В этом разделе мы обратимся к основным вычислительным вопросам, связанным с изучением графов «тесного мира». В действительности простой вопрос:

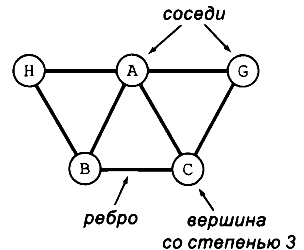
Обладает ли заданный граф свойством «тесного мира»?

представляет значительную вычислительную сложность. Для получения ответа на этот вопрос мы рассмотрим тип данных для работы с графами, а также несколько полезных приложений-клиентов для обработки графов. Особое внимание будет уделено клиенту для вычисления *кратчайших путей* — эта задача сама по себе находит огромное количество важных применений.

В этом разделе следует постоянно помнить, что изучаемые алгоритмы и структуры данных играют центральную роль в обработке графов. Вы увидите, что некоторые фундаментальные типы данных, представленные ранее в этой главе, помогают разрабатывать элегантный и эффективный код для изучения свойств графов.

Графы

Чтобы с самого начала предотвратить какую-либо терминологическую путаницу, начнем с определений. *Граф* состоит из множества *вершин* и множества *ребер*. Каж-



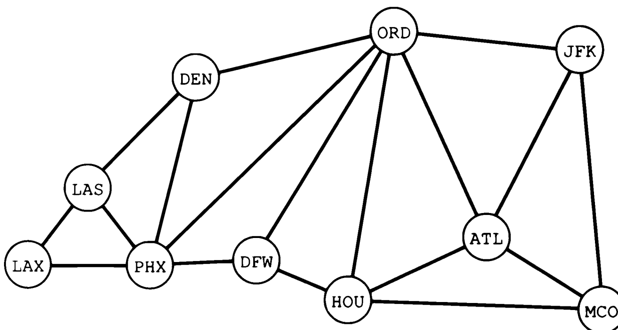
Граф: определение

дое ребро соединяет две вершины. Две вершины называются *смежными*, если они соединены ребром, а *степенью* вершины называется количество смежных с ней вершин (соседей.) Графы часто представляются в виде кружков с подписями (вершины), соединенных линиями (ребра), но всегда важно помнить, что важны связи, а не способ их представления.

Следующий список дает представление о разнообразии систем, при изучении которых графы становятся хорошей отправной точкой.

Транспортные системы

Станции соединяются рельсами, перекрестки соединяются дорогами, а маршруты самолетов соединяют аэропорты; все эти системы естественным образом укладываются в простую модель графа. Несомненно, вы уже использовали приложения, основанные на этой модели, при получении указаний от интерактивных картографических программ или устройств GPS или при планировании путешествий при помощи онлайн-сервисов. Как лучше всего добраться из одной точки в другую?



вершины	*ребра*
JFK	JFK MCO
MCO	ORD DEN
ATL	ORD HOU
ORD	DFW PHX
HOU	JFK ATL
DFW	ORD DFW
PHX	ORD PHX
DEN	ATL HOU
LAX	DEN PHX
LAS	PHX LAX
	JFK ORD
	DEN LAS
	DFW HOU
	ORD ATL
	LAS LAX
	ATL MCO
	HOU MCO
	LAS PHX

Модель графа транспортной системы

Биология

Органы соединяются артериями и венами, нейроны соединяются синапсами, а кости соединяются суставами; таким образом, понимание человеческой биологии тоже зависит от понимания соответствующих моделей графов. Пожалуй, самым большим, сложным и важным объектом моделирования в этой области является человеческий мозг. Как локальные соединения между нейронами превращаются в сознание, память и интеллект?

Социальные сети

Люди устанавливают отношения с другими людьми. Графы таких связей чрезвычайно важны для нашего понимания самых разных областей, от изучения инфекционных заболеваний до анализа политических тенденций. Другая интереснейшая задача — понимание механизмов распространения информации в социальных сетях.

система	вершины	ребра
природные явления		
кровообращение	органы	кровеносные сосуды
скелет	суставы	кости
нервная система	нейроны	синапсы
социальные системы	люди	отношения
эпидемиология	люди	заболевания
химия	молекулы	связи
задача n тел	частицы	силы
генетика	гены	мутации
биохимия	протеины	взаимодействия
технические системы		
транспортные системы	аэропорты	маршруты
	перекрестки	дороги
коммуникационные системы	телефоны	провода
	компьютеры	кабели
	веб-страницы	ссылки
распределительные системы	электростанции/дома	линии электропередач
	хранилища/дома	трубы
	склады/розничные магазины	грузовые линии
механические системы	соединения	балки
программные системы	модули	вызовы
финансовые системы	портфели акций	операции

Типичные модели на базе графов

Физические системы

Соединения атомов образуют молекулы, соединения молекул образуют вещества и кристаллические структуры, а частицы соединяются силами взаимодействия, такими как гравитация или магнетизм. Например, модель графа хорошо подходит для изучения задачи протекания, рассмотренной в разделе 2.4. Как локальные взаимодействия распространяются в таких системах в процессе их эволюции?

Коммуникационные системы

Электрические цепи, телефонные системы, Интернет, беспроводные сети — в основе всех их лежат соединения устройств. По крайней мере, в прошлом веке основанные на графах модели сыграли критическую роль в разработке таких систем. Как лучше всего соединить такие устройства?

Распределение ресурсов

Линии электропередач соединяют электростанции с домашними электрическими системами, трубы соединяют резервуары с домашним водопроводом, а грузовые линии соединяют склады с магазинами. Анализ эффективных и надежных средств распределения ресурсов зависит от точности графовых моделей. Где в распределительной системе существуют «узкие места»?

Механические системы

Стальные тросы и балки, узлы конструкций — мостов или зданий. Модели, основанные на графах, помогают проектировать такие системы и понимать их свойства. Какую нагрузку должно выдерживать соединение или балка?

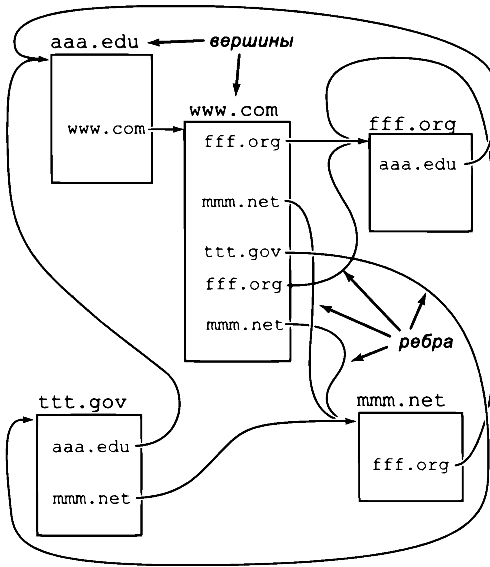
Программные системы

Методы одного программного модуля вызывают методы других модулей. Как вы уже видели в этой книге, понимание отношений такого рода — ключ к успеху проектирования программных систем. На каких модулях отразится изменение в API?

Финансовые системы

Операции связывают портфели акций, а портфели акций связывают клиентов с финансовыми учреждениями. Это лишь некоторые графовые модели, которые используются для изучения сложных финансовых операций и извлечения выгоды из более глубокого их понимания. Какие операции являются рутинными, а какие свидетельствуют о значительных событиях, которые могут превратиться в прибыль?

Некоторые из этих моделей отражают природные явления, а при их разработке мы стремимся к лучшему пониманию реального мира за счет разработки простых моделей и использования их для формулирования гипотез, которые могут быть проверены экспериментальным путем. Другие модели на базе графов представляют сети, которые мы можем построить, и целью является проектирование более эффективной сети или повышение эффективности управления сетью за счет понимания ее базовых характеристик.

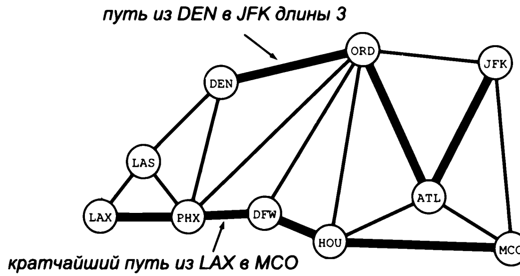


вершины	ребра	
aaa.edu	aaa.edu	www.com
www.com	www.com	fff.org
mmm.net	www.com	mmm.net
fff.org	www.com	ttt.gov
ttt.gov	www.com	fff.org
	www.com	mmm.net
	mmm.net	fff.org
	fff.org	aaa.edu
	ttt.gov	aaa.edu
	ttt.gov	mmm.net

Графовая модель Интернета

На практике используются графовые модели различных размеров. Граф с десятками вершин и ребер (например, моделирующий химическое соединение, где вершины представляют молекулы, а ребра — связи) уже является сложным комбинаторным объектом из-за огромного количества возможных графов (на том же наборе вершин), поэтому понимание структур конкретного имеющегося графа достаточно важно. Граф с миллиардами и триллионами вершин и ребер (например, правительственная база данных с метаданными всех телефонных звонков или модель человеческой нервной системы) намного сложнее, а его обработка связана со значительными сложностями. Как правило, обработка графа подразумевает построение графа на основании информации из файлов и последующем получении ответов на вопросы о нем. Кроме вопросов, связанных со спецификой конкретной области (как в только что рассмотренных примерах), часто задаются базовые вопросы о графах в целом. Сколько вершин и ребер содержит граф? Какие вершины являются соседями для заданной вершины? Некоторые вопросы зависят от понимания структуры графа. Например, *путь* в графе представляет собой последовательность смежных вершин, соединенных ребрами. Существует ли путь, связывающий две заданные вершины? Какова *длина* (количество ребер) кратчайшего пути, соединяющего две вершины? В этой книге уже приводились примеры вопросов из научной области, намного более сложных. Какова вероятность того, что при случайном серфинге пользователь попадет на конкретную вершину? Какова вероятность того, что в системе, представленной графом, существует протекание?

Когда вы начнете изучать сложные системы на курсах более высокого уровня, вам наверняка встретятся графы в самых разных контекстах. Также не исключено, что вам придется заняться изучением их свойств в более поздних курсах математики,



Пути в графе

анализа операций или информатики. Некоторые задачи обработки графов создают непреодолимые вычислительные сложности; другие решаются относительно легко с реализациями типов данных вроде тех, которые мы рассматриваем.

Тип данных для графа

Алгоритмы обработки графов обычно начинают с построения внутреннего представления графа и добавления ребер, после чего переходят к перебору вершин и ребер, прилегающих к заданной вершине. Следующий API обеспечивает такую обработку:

`public class Graph`

	`Graph()`	создает пустой граф
	`Graph(In in, String delimiter)`	читает данные графа из входного потока
`void`	`addEdge(String v, String w)`	добавляет ребро v–w
`int`	`V()`	количество вершин
`int`	`E()`	количество ребер
`Iterable<String>`	`vertices()`	вершины графа
`Iterable<String>`	`adjacentTo(String v)`	соседи v
`int`	`degree(String v)`	количество соседей v
`boolean`	`hasVertex(String v)`	является ли v вершиной графа?
`boolean`	`hasEdge(String v, String w)`	является ли v–w ребром графа?

API для графа с вершинами String

Как обычно, в API отражены некоторые решения из области проектирования. Каждое решение было выбрано из нескольких возможных вариантов; некоторые из них будут кратко описаны ниже.

Ненаправленный граф

Ребра графа не имеют направления; ребро, соединяющее v с w, — то же самое, которое соединяет w с v. Нас интересует сам факт существования связи, а не ее направление. Направленные ребра или *дуги* (например, улицы с односторонним движением на картах) требуют несколько иного типа данных (см. упражнение 4.5.41).

Тип вершины String

Можно также использовать обобщенный тип вершины, чтобы клиент мог создавать графы из объектов любого типа. Впрочем, реализация такого рода остается вам для самостоятельной работы, потому что полученный код получается довольно громоздким (см. упражнение 4.5.9). Для применений, которые мы рассмотрим, типа вершины `String` вполне достаточно.

Недействительные имена вершин

Методы `adjacentTo()`, `degree()` и `hasEdge()` выдают исключение при вызове со строковым аргументом, не соответствующим имени вершины. Клиент может вызвать `hasVertex()` для предварительной проверки таких ситуаций.

Неявное создание вершин

Когда в аргументе `addEdge()` передается строка, мы предполагаем, что это имя вершины. Если вершина с таким именем еще не была включена в граф, наша реализация добавляет ее. Альтернативное решение с отдельным методом `addVertex()` увеличивает объем клиентского кода, а код реализации становится более громоздким (он должен проверять существование ребер, соединяющих вершины).

Петли и параллельные ребра

И хотя в API эта проблема не упоминается явно, мы предполагаем, что реализации допускают существование петель (ребер, соединяющих вершину саму с собой), но запрещают параллельные ребра (две копии одного ребра). Проверка петель и параллельных ребер проста; мы решили опустить обе проверки.

Клиентские методы получения информации

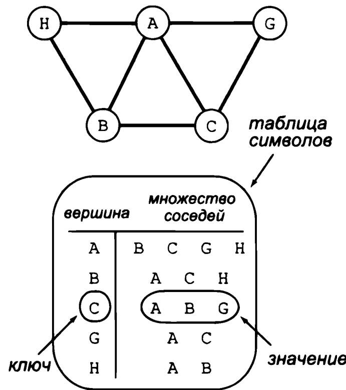
Мы также включили в API методы `V()` и `E()`, которые возвращают клиенту количество вершин и ребер в графе. Аналогичным образом в клиентском коде могут использоваться методы `degree()`, `hasVertex()` и `hasEdge()`. Реализация этих методов остается для упражнений, но мы будем считать, что они входят в API `Graph`.

Ни одно из этих решений не является непреложным; это всего лишь варианты, выбранные нами для кода в книге. В других ситуациях могут быть уместны другие решения, а некоторые решения остаются на усмотрение разработчиков конкретных реализаций. Постарайтесь тщательно продумать решения, принимаемые вами при проектировании, и *будьте готовы обосновать их*.

Класс `Graph` (листинг 4.5.1) реализует этот API. Во внутреннем представлении используется таблица символов, содержащая множества: ключами являются вершины, а значениями — множества соседей (вершин, смежных с ключом). В представлении задействованы типы данных `ST` и `SET`, представленные в разделе 4.4. Оно обладает тремя важными свойствами.

- Клиенты могут эффективно перебирать вершины графа.
- Клиенты могут эффективно перебирать соседей заданной вершины.
- Затраты памяти пропорциональны количеству ребер.

Эти свойства непосредственно следуют из базовых свойств `ST` и `SET`. Как вы увидите, эти два итератора лежат в основе обработки графов.



Представление графа в виде таблицы символов ST множеств SET

В качестве простого примера клиентского кода рассмотрим задачу вывода содержимого `Graph`. Естественное решение заключается в выводе списка вершин со списком соседей каждой вершины. Мы используем этот метод для реализации `toString()` в `Graph`:

```
public String toString()
{
   String s = "";
   for (String v : vertices())
   {
      s += v + "  ";
      for (String w : adjacentTo(v))
         s += w + " ";
      s += "\n";
   }
   return s;
}
```

Листинг 4.5.1. Тип данных Graph

```
public class Graph
{
   private ST<String, SET<String>> st;

   public Graph()
   {  st = new ST<String, SET<String>>();  }

   public void addEdge(String v, String w)
   {  // v заносится в SET для w, а w - в SET для v.
      if (!st.contains(v)) st.put(v, new SET<String>());
      if (!st.contains(w)) st.put(w, new SET<String>());
      st.get(v).add(w);
      st.get(w).add(v);
   }

   public Iterable<String> adjacentTo(String v)
   {  return st.get(v);  }

   public Iterable<String> vertices()
   {  return st.keys();  }

   // За реализациями V(), E(), degree(), hasVertex()
   // и hasEdge() обращайтесь к упражнениям 4.5.1-4.5.4.

   public static void main(String[] args)
   {  // Чтение ребер из стандартного потока ввода; вывод полученного графа.
      Graph G = new Graph();
      while (!StdIn.isEmpty())
         G.addEdge(StdIn.readString(), StdIn.readString());
      StdOut.print(G);
   }
}
```

st	таблица символов для множеств соседей

Реализация использует типы ST и SET (см. раздел 4.4) для реализации типа данных графа. Клиент строит граф, добавляя ребра, и обрабатывает его, перебирая вершины и элементы множества вершин, смежных с каждой вершиной. Реализация toString() и конструктора, читающего граф из входного потока, приведена далее в тексте.

```
% more tinyGraph.txt
A B
A C
C G
A G
H A
B C
B H
```

```
% java Graph < tinyGraph.txt
A  B C G H
B  A C H
C  A B G
G  A C
H  A B
```

Этот код выводит два представления для каждого ребра — сначала когда обнаруживает, что w является соседом v, а потом когда обнаруживает, что v является соседом w. Многие алгоритмы графов базируются на парадигме обработки всех ребер, и важно помнить, что каждое ребро обрабатывается дважды. Как обычно, эта реализация подходит только для очень небольших графов, потому что время выполнения находится в квадратичной зависимости от длины строки, так как конкатенация строк выполняется за линейное время.

Рассмотренный формат вывода определяет адекватный формат файла: каждая строка содержит имя вершины, за которым следуют имена соседей этой вершины. Соответственно, наш базовый API для работы с графом включает конструктор для построения графа из входного потока в этом формате (список вершин с соседями). Для большей гибкости мы разрешаем использовать другие разделители, кроме пробелов (так что имена вершин, например, могут содержать пробелы):

```
public Graph(In in, String delimiter)
{
   st = new ST<String, SET<String>>();
   while (in.hasNextLine())
   {
      String line = in.readLine();
      String[] names = line.split(delimiter);
      for (int i = 1; i < names.length; i++)
         addEdge(names[0], names[i]);
   }
}
```

При добавлении в `Graph` этого конструктора и `toString()` вы получаете полноценный тип данных, который, как вы сейчас убедитесь, может использоваться в самых разных ситуациях. Обратите внимание: тот же конструктор (с пробелом в качестве разделителя) нормально работает и в том случае, когда набор входных данных представляет собой список ребер, по одному на строку, как в тестовом клиенте для листинга 4.5.1.

Пример приложения-клиента для Graph

В своем первом приложении, использующем `Graph`, мы рассмотрим пример социальных отношений — наверняка ситуация хорошо знакома вам, и для нее доступны обширные данные.

На сайте книги имеется файл movies.txt (и другие аналогичные файлы) со списком фильмов и актеров, которые в них играли. В каждой строке содержится название фильма фильма и список актеров. Так как в именах могут присутствовать пробелы и запятые, в качестве разделителя используется символ `/` (теперь вы видите, почему второй конструктор `Graph` получает разделитель как отдельный аргумент).

Если вы внимательно присмотритесь к movies.txt, то заметите несколько характерных особенностей, которые необходимо учесть при работе с базой данных, несмотря на их незначительность.

- У фильмов после названия всегда указывается год в круглых скобках.
- В тексте встречаются специальные символы.
- Актеры с одинаковыми именами различаются по римским цифрам в круглых скобках.
- Списки актеров не упорядочены по алфавиту.

В зависимости от настроек терминального окна и операционной системы специальные символы могут заменяться пробелами или вопросительными знаками. Такие аномалии часто встречаются при работе с большими объемами реальных данных. Вы можете либо принять эти аномалии как есть, либо внести соответствующие настройки в свою среду (за подробностями обращайтесь на сайт книги).

```
% more movies.txt
...
Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/... /Geppi, Cindy/Hershey, Barbara
Tirez sur le pianiste (1960)/Heymann, Claude/.../Berger, Nicole (I)
Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/.../Winslet, Kate/...
Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/.../McEwan, Geraldine
To Be or Not to Be (1942)/Verebes, Ernö (I)/.../Lombard, Carole (I)
To Be or Not to Be (1983)/.../Brooks, Mel (I)/.../Bancroft, Anne/...
To Catch a Thief (1955)/París, Manuel/.../Grant, Cary/.../Kelly, Grace/...
To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/.../ Tucci, Maria
...
```

Фрагмент базы данных с информацией о фильмах

При помощи Graph можно написать простое и удобное клиентское приложение для извлечения информации из файла movies.txt. Мы начнем с построения объекта Graph для структурирования информации. Что должны моделировать вершины и ребра? Вершины представляют фильмы, а ребра соединяют два фильма, если актер играл в обоих фильмах? Или вершины представляют актеров, а ребра соединяют двух актеров, если они играли в одном фильме? Оба варианта выглядят разумно, но какой из них использовать? Решение повлияет как на клиентский код, так и на код реализации. Еще в одном варианте (который мы выбрали, потому что он ведет к простому коду реализации) вершины представляют как фильмы, так и актеров, а ребра соединяют каждый фильм с каждым актером, который в этом фильме играл. Как вы увидите, программы для работы с графом могут ответить на множество интересных вопросов. Программа IndexGraph (листинг 4.5.2) — первый пример такого рода. Она получает запрос (например, название фильма) и выводит список актеров, которые играли в этом фильме.

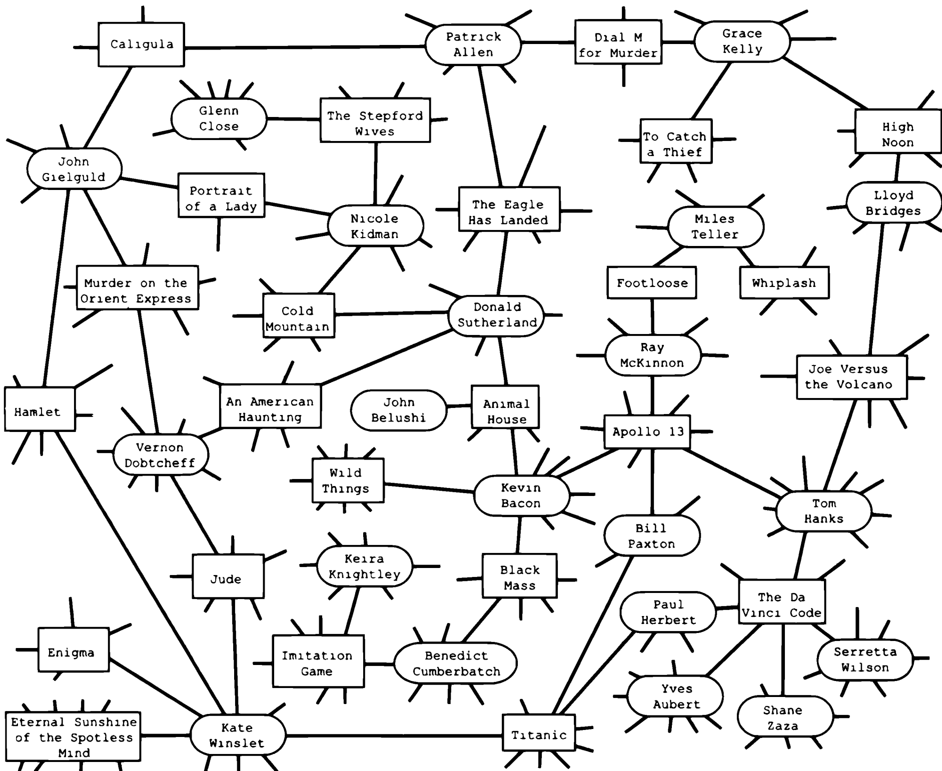
Если ввести название фильма, вы получите список актеров — фактически просто выводится информация из соответствующей строки movies.txt (хотя IndexGraph

выводит список отсортированным по фамилиям, так как этот порядок перебора предоставляется `SET` по умолчанию). У `IndexGraph` есть и более интересная возможность: вы можете ввести имя актера и получить список фильмов, в которых он снимался. Почему этот режим работает?

Хотя файл movies.txt вроде бы связывает фильмы с актерами, а не наоборот, ребра графа не имеют направления и поэтому также связывают актеров с фильмами.

Граф, в котором все соединения связывают одну разновидность вершин с другой разновидностью вершин, но между собой, называется *двудольным графом*. Как видно из этого примера, двудольные графы обладают многими естественными свойствами, для которых можно найти много интересных применений.

Как было показано в начале раздела 4.4, парадигма индексирования универсальна и хорошо знакома разработчикам. Но, по сути, построение двудольного графа предоставляет простой механизм для автоматического инвертирования любого индекса! Файл movies.txt индексируется по фильмам, но программа может получать запросы по имени актера. `IndexGraph` может точно так же использоваться для получения списка слов, встречающихся на заданной странице, или кодонов, соответствующих



Крошечный фрагмент графа с фильмами и исполнителями

Листинг 4.5.2. Использование графа для инвертирования индекса

```
public class IndexGraph
{
   public static void main(String[] args)
   {  // Построение графа и выполнение запросов.
      In in = new In(args[0]);
      String delimiter = args[1];
      Graph G = new Graph(in, delimiter);
      while (StdIn.hasNextLine())
      {  // Чтение вершины и вывод ее соседей.
         String v = StdIn.readLine();
         for (String w : G.adjacentTo(v))
            StdOut.println("  " + w);
      }
   }
}
```

in	входной поток
delimiter	разделитель
G	граф
v	запрос
w	сосед v

Приложение-клиент для Graph создает граф из файла, заданного в командной строке, после чего читает имена вершин из стандартного потока ввода и выводит их соседей. Для файла, содержащего список фильмов и актеров, граф получается двудольным, а программа фактически работает как интерактивный обратный (инвертированный) индекс.

```
% java IndexGraph tinyGraph.txt " "
C
  A
  B
  G
A
  B
  C
  G
  H
```

```
% java IndexGraph movies.txt "/"
Da Vinci Code, The (2006)
  Aubert, Yves
  ...
  Herbert, Paul
  ...
  Wilson, Serretta
  Zaza, Shane
Bacon, Kevin
  Animal House (1978)
  Apollo 13 (1995)
  ...
  Wild Things (1998)
  River Wild, The (1994)
  Woodsman, The (2004)
```

заданной аминокислоте, или для инвертирования любых других индексов, упоминавшихся в начале раздела 4.2. Так как программа IndexGraph получает разделитель в аргументе командной строки, вы можете использовать ее для создания интерактивного обратного (инвертированного) индекса для файла .csv.

Функциональность инвертирования индекса — дополнительное свойство, напрямую обусловленное *структурой данных* графа. А теперь рассмотрим другие полезные свойства, которые следуют из *алгоритмов*, обрабатывающих структуру данных.

Кратчайшие пути в графах

Для двух вершин графа *путь* представляет собой последовательность ребер, соединяющих эти вершины. *Кратчайшим путем* является путь с минимальной

длиной, или расстоянием (выраженным в количестве ребер), среди всех таких путей (часто кратчайших путей бывает несколько). Поиск кратчайшего пути, соединяющего две вершины графа, относится к числу фундаментальных задач в информатике. Концепция кратчайших путей успешно применялась при решении крупномасштабных задач в разнообразных областях, от маршрутизации в Интернете до планирования финансовых операций или описания динамики нейронов в мозге.

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
...
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine

% java IndexGraph amino.csv ","
TTA
  Lue
  L
  Leucine
Serine
  TCT
  TCC
  TCA
  TCG
```

Инвертирование индекса

Представьте, что вы являетесь клиентом авиакомпании эконом-класса, которая обслуживает ограниченное число городов с ограниченным набором маршрутов. Предполагается, что лучший способ попасть из одного города в другой состоит из минимального числа полетных сегментов, потому что задержки при пересадках между рейсами могут быть довольно значительными. Алгоритм вычисления кратчайшего пути — именно то, что необходимо для планирования таких поездок. Такое применение соответствует нашим интуитивным представлениям о базовой задаче и разумном подходе к ее решению. После рассмотрения этой темы в контексте примера будет рассмотрен другой пример с более абстрактной моделью графа.

В каждой конкретной ситуации клиенты обладают разными потребностями в отношении кратчайших путей. Что нужно клиенту — кратчайший путь, соединяющий две заданные вершины? Или только длина такого пути? Ожидается ли большое количество таких запросов? Представляет ли одна какая-то вершина особый интерес? При больших графах или большом количестве запросов необходимо обращать на такие вопросы особое внимание, потому что затраты на вычисление кратчайших путей могут оказаться неприемлемыми. Начнем со следующего API.

`public class PathFinder`

`PathFinder(Graph G, String s)`	конструктор
`int distanceTo(String v)`	длина кратчайшего пути из s в v в графе G
`Iterable<String> pathTo(String v)`	кратчайший путь из s в v в графе G

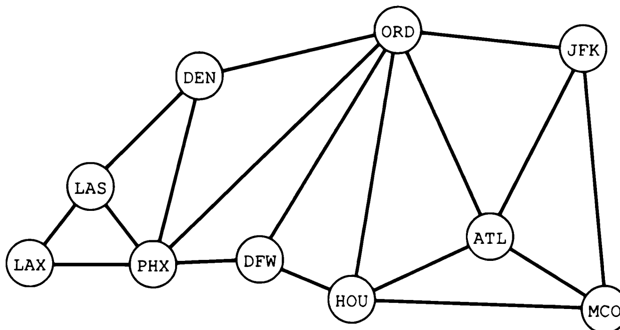
API для нахождения кратчайших путей из одной начальной вершины в графе

Программа-клиент может создать объект `PathFinder` для заданного графа `G` и начальной вершины `s`, а затем использовать этот объект для нахождения длины кратчайшего пути или для перебора вершин кратчайшего пути из `s` в любую другую вершину `G`. Реализация этих методов называется алгоритмом *кратчайшего пути из*

одной начальной вершины. Мы рассмотрим *поиск в ширину* — классический алгоритм для решения этой задачи, предоставляющий простое и элегантное решение.

Клиент с одной начальной вершиной

Предположим, вы рассматриваете граф вершин и соединений на карте маршрутов вашей авиакомпании. Затем, используя свой родной город в качестве пункта отправления, вы пишете приложение-клиент, которое строит маршрут перед каждой поездкой. В листинге 4.5.3 представлена такая программа, использующая `PathFinder`, который предоставляет нужную функциональность для любого графа. Клиенты такого рода особенно полезны в приложениях, в которых можно ожидать множественных запросов от одного источника, предполагающих одну и ту же начальную вершину. В такой ситуации статические затраты на построение объекта `PathFinder` амортизируются за счет времени выполнения множественных запросов. Поэкспериментируйте с исследованием свойств кратчайших путей, запустив `PathFinder` для входного файла routes.txt.



начальная вершина	*конечная вершина*	*длина пути*	*кратчайший путь*
JFK	LAX	3	JFK-ORD-PHX-LAX
LAS	MCO	4	LAS-PHX-DFW-HOU-MCO
HOU	JFK	2	HOU-ATL-JFK

Примеры кратчайших путей на графе

Степени разделения

Одно из классических применений алгоритмов кратчайших путей — определение *степеней разделения* пользователей в социальных сетях. Для конкретности рассмотрим это применение в контексте популярного развлечения, называемого *игрой Кевина Бэйкона*, для которого используется уже рассмотренный нами граф фильмов и актеров. Кевин Бэйкон — популярный актер, снимавшийся во многих фильмах. Каждому актеру назначается *число Кевина Бэйкона*: самому Бэйкону — 0; всем актерам, которые снимались с ним в одном фильме, — 1; любому актеру, снимавшемуся в одном фильме с актером с числом 1 (кроме самого Бэйкона), — 2,

и т. д. Например, у Мэрил Стрип число Кевина Бэйкона равно 1, потому что она снималась с Кевином Бэйконом в фильме «Дикая река». У Николь Кидман это число равно 2: хотя она не снималась с Кевином Бэйконом ни в одном фильме, она снималась в «Холодной горе» с Дональдом Сазерлендом, а Сазерленд снимался с Кевином Бэйконом в фильме «Зверинец». В простейшем варианте игры для заданного актера требуется найти последовательность фильмов и актеров, приводящую к Кевину Бэйкону. Например, киноман может знать, что Том Хэнкс снимался в фильме «Джо против вулкана» с Ллойдом Бриджесом, который снимался в фильме «Ровно в полдень» с Грейс Келли, которая снималась в фильме «В случае убийства набирайте "М"» с Патриком Алленом, который снимался в фильме «Орел приземлился» с Дональдом Сазерлендом, который, как мы уже знаем, снимался в «Зверинце» с Кевином Бэйконом. Но этой информации недостаточно для установления числа Бэйкона для Тома Хэнкса (которое на самом деле равно 1, потому что он снимался с Кевином Бэйконом в фильме «Аполлон-13»). Как видите, число Кевина Бэйкона должно определяться подсчетом фильмов в кратчайшей цепочке, поэтому без использования компьютера трудно быть уверенным в результате игры. Примечательно, что тестовый клиент `PathFinder` из листинга 4.5.3 содержит весь код, необходимый для нахождения кратчайшего пути с целью определения числа Кевина Бэйкона для любого актера в movies.txt, — это число равно в точности половине расстояния[1]. Поэкспериментируйте с этой программой или попробуйте доработать ее для получения ответов на интересные вопросы из области кинематографа или многих других областей. Например, математики играют в аналогичную игру с графами соавторства статей и их связей с Палом Эрдёшем (Paul Erdös), выдающимся математиком XX века. И похоже, у всех жителей Нью-Джерси число Брюса Спрингстина равно 2, потому что у любого жителя этого штата есть хотя бы один знакомый, который утверждает, что он лично знаком с Брюсом.

```
% java PathFinder movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
   Bacon, Kevin
   Animal House (1978)
   Sutherland, Donald (I)
   Cold Mountain (2003)
   Kidman, Nicole
distance 4
Hanks, Tom
   Bacon, Kevin
   Apollo 13 (1995)
   Hanks, Tom
distance 2
```

Степени разделения

[1] Легко заметить, что «степень разделения» — это и есть расстояние в графе, где вершины соответствуют пользователям. Удвоение длины пути в ранее описанном `PathFinder` объясняется двудольностью используемого в нем графа. — *Примеч. науч. ред.*

Листинг 4.5.3. Клиент для поиска кратчайших путей

```
public class PathFinder
{
   // Реализация приведена в листинге 4.5.4.
   public static void main(String[] args)
   {
      // Прочитать граф и вычислить кратчайшие пути от s.
      In in = new In(args[0]);
      String delimiter = args[1];
      Graph G = new Graph(in, delimiter);
      String s = args[2];
      PathFinder pf = new PathFinder(G, s);

      // Обработать запросы.
      while (StdIn.hasNextLine())
      {
         String t = StdIn.readLine();
         int d = pf.distanceTo(t);
         for (String v : pf.pathTo(t))
            StdOut.println("   " + v);
         StdOut.println("distance " + d);
      }
   }
}
```

in	входной поток
delimiter	разделитель входных данных
G	граф
s	источник
pf	PathFinder от s
t	запрос конечной вершины
v	вершина на пути

Этот клиент получает имя файла, разделитель и вершину-источник в аргументах командной строки. Он строит граф на основании файла (предполагается, что каждая строка файла задает вершину и список вершин, соединенных с ней, с заданным разделителем). Вводя в стандартном вводе конечную вершину, вы получаете кратчайший путь от источника к этой конечной вершине.

```
% more routes.txt
JFK MCO
ORD DEN
PHX LAX
ORD HOU
DFW PHX
ORD DFW
...
JFK ORD
HOU MCO
LAS PHX
```

```
% java PathFinder routes.txt " " JFK
LAX
   JFK
   ORD
   PHX
   LAX
distance 3
DFW
   JFK
   ORD
   DFW
distance 2
```

Другие примеры

`PathFinder` — универсальный тип данных, который находит много практических применений. Например, можно легко написать приложение, поддерживающее запросы с произвольными начальной и конечной вершинами на входе, — для этого следует построить объект `PathFinder` для каждой вершины (см. листинг 4.5.17).

Сервисы планирования поездок применяют именно этот способ для быстрой обработки запросов. Так как клиент строит `PathFinder` для каждой вершины (что может требовать затрат памяти, пропорциональных количеству вершин), для очень больших графов затраты памяти могут стать ограничивающим фактором. В качестве примера концептуально сходного, но еще более критичного по быстродействию можно привести интернет-маршрутизатор, который хранит граф соединений между компьютерами и выбирает следующий узел для пересылки пакетов, предназначенных для заданного получателя. Для этого он может создать объект `PathFinder`, в котором начальной точкой маршрута является он сам; тогда для отправки пакета получателю *w* он вычисляет `pf.pathTo(w)` и отправляет пакет первой вершине на этом пути — следующей остановке на кратчайшем пути к *w*. Также некий управляющий центр может создать объекты `PathFinder` для каждого из подчиненных маршрутизаторов и использовать эти объекты для передачи инструкций по маршрутизации. Возможность обработки таких запросов с высокой скоростью — одна из главных обязанностей интернет-маршрутизаторов, и алгоритмы нахождения кратчайшего пути являются важнейшей частью этого процесса.

Длина кратчайшего пути

Чтобы понять суть поиска в ширину, начать следует с задачи вычисления длин кратчайших путей от начальной вершины до каждой другой вершины. Наше решение вычисляет и сохраняет все расстояния в конструкторе `PathFinder`, а затем просто возвращает запрошенное значение при вызове клиентом `distanceTo()`. Чтобы связать целочисленное расстояние с каждым именем вершины, мы воспользуемся таблицей символов:

```
ST<String, Integer> dist = new ST<String, Integer>();
```

Эта таблица символов связывает каждую вершину с целым числом: длиной кратчайшего пути (расстоянием) от `s` до этой вершины. Выполнение алгоритма начинается с назначения `s` расстояния 0 вызовом `dist.put(s, 0)`, а всем соседям `s` назначается расстояние 1:

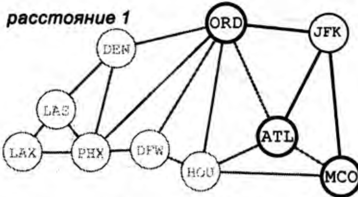
```
for (String v : G.adjacentTo(s))
   dist.put(v, 1)
```

Но что происходит потом? Если просто назначить всем соседям каждого соседа расстояние 2, то мы не только сталкиваемся с возможностью излишнего повторного присваивания многих значений (у соседей могут быть общие соседи), но и присвоим `s` расстояние 2 (так как `s` является соседом каждого из своих соседей), а это явно не то, что нам требуется. Проблема решается просто.

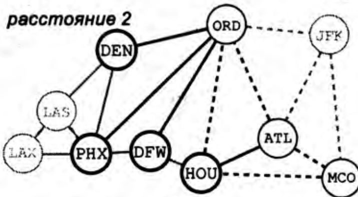
- Рассматривать вершины в порядке их расстояния от `s`.
- Игнорировать вершины, расстояние которых от `s` уже известно.

v	содержимое очереди	ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
	JFK					0					

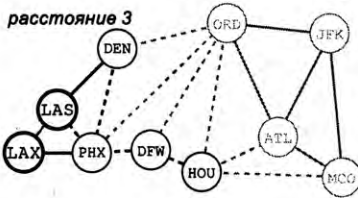
инициализация значением расстояния 1



v	содержимое очереди	ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
JFK						0					
JFK	ATL	1				0					
JFK	ATL MCO	1				0			1		
JFK	ATL MCO ORD	1				0			1	1	



v	содержимое очереди	ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
ATL	MCO ORD	1				0			1	1	
ATL	MCO ORD HOU	1			2	0			1	1	
MCO	ORD HOU	1			2	0			1	1	
ORD	HOU	1			2	0			1	1	
ORD	HOU DEN	1	2		2	0			1	1	
ORD	HOU DEN DFW	1	2	2	2	0			1	1	
ORD	HOU DEN DFW PHX	1	2	2	2	0			1	1	2



v	содержимое очереди	ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
HOU	DEN DFW PHX	1	2	2	2	0			1	1	2
DEN	DFW PHX	1	2	2	2	0			1	1	2
DEN	DFW PHX LAS	1	2	2	2	0	3		1	1	2
DFW	PHX LAS	1	2	2	2	0	3		1	1	2
PHX	LAS	1	2	2	2	0	3		1	1	2
PHX	LAS LAX	1	2	2	2	0	3	3	1	1	2

проверка для расстояния 4

v	содержимое очереди	ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
LAS	LAX	1	2	2	2	0	3	3	1	1	2
LAX		1	2	2	2	0	3	3	1	1	2

Использование поиска в ширину для вычисления кратчайших путей в графе

Для организации вычислений будет использоваться очередь FIFO. Начиная с занесения в очередь вершины s, мы выполняем следующие операции, пока очередь не окажется пустой.

- Извлечь из очереди вершину v.
- Назначить всем неизвестным (не встречавшимся ранее) соседям v расстояние, на 1 большее, чем у v.
- Поставить в очередь всех неизвестных соседей.

Алгоритм поиска в ширину извлекает из очереди вершины в порядке неубывания их расстояния от начальной вершины s. Трассировка этого алгоритма на конкретном графе поможет вам убедиться в его правильности. Доказательство того, что поиск в ширину связывает каждую вершину v с расстоянием от нее до s, проводится методом математической индукции (см. упражнение 4.5.12).

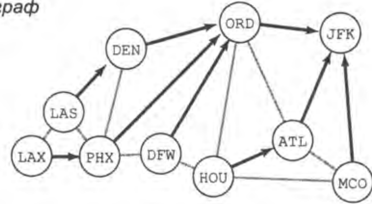
Дерево кратчайших путей

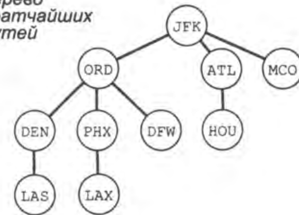
Нас интересуют не только длины кратчайших путей, но и сами кратчайшие пути. Чтобы реализовать `pathTo()`, мы воспользуемся подграфом, называемым *деревом кратчайших путей*. Он определяется следующим образом.

- Начальная вершина располагается в корне дерева.
- Соседи вершины `v` включаются в дерево, если они добавляются в очередь при обработке вершины `v`.

Так как каждая вершина помещается в очередь только один раз, эта структура соответствует определению дерева: она состоит из корня (начальной вершины), связанного с каждым соседом этой вершины. Анализ такого дерева показывает, что расстояние от каждой вершины до корня дерева равно длине кратчайшего пути от начальной вершины в графе. Что еще важнее, каждый путь в дереве представляет кратчайший путь в графе. Это важно, потому что данное свойство позволяет легко предоставить клиентам сами кратчайшие пути. Мы создаем таблицу символов, которая связывает каждую вершину с вершиной, находящейся на один шаг ближе на кратчайшем пути:

```
ST<String, String> prev;
prev = new ST<String, String>();
```





ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
JFK	ORF	ORD	ATL		DEN	PHX	JFK	JFK	ORD

Дерево кратчайших путей

С каждой вершиной `w` должна быть связана предыдущая остановка на кратчайшем пути от начальной вершины к `w`. Включить эту информацию в механизм поиска в ширину несложно: когда мы ставим в очередь вершину `w`, впервые обнаруживая ее как соседа `v`, мы делаем это из-за того, что `v` находится на предыдущем шаге кратчайшего пути от начальной вершины к `w`, поэтому мы вызываем `prev.put(w, v)` для сохранения этой информации. Структура данных `prev` представляет собой не что иное, как представление дерева кратчайших путей: она связывает каждый узел с его родителем в дереве. Таким образом, для получения ответа на клиентский запрос кратчайшего пути от начальной вершины к `v` эти связи отслеживаются *наверх* по дереву от `v`. Вершины, входящие в путь, при этом обходятся в обратном порядке, поэтому они заносятся в стек, который (в качестве `Iterable`) затем возвращается клиенту. На вершине стека находится начальная вершина `s`; в нижней позиции стека находится `v`; между ними хранятся вершины пути от `s` до `v`, поэтому при использовании возвращаемого значения от `pathTo()` в конструкции *foreach* клиент получает путь от `s` к `v`.

дерево кратчайших путей (представление связей с родителем) — *содержимое стека*

```
ATL DEN DFW HOU JFK LAS LAX MCO ORD PHX
JFK ORF ORD ATL     DEN PHX JFK JFK ORD      LAX ← конечная вершина

ATL DEN DFW HOU JFK LAS LAX MCO ORD PHX
JFK ORF ORD ATL     DEN PHX JFK JFK ORD      PHX LAX

ATL DEN DFW HOU JFK LAS LAX MCO ORD PHX
JFK ORF ORD ATL     DEN PHX JFK JFK ORD      ORD PHX LAX
                                                    путь
ATL DEN DFW HOU JFK LAS LAX MCO ORD PHX
JFK ORF ORD ATL     DEN PHX JFK JFK ORD      JFK ORD PHX LAX
                                 начальная вершина
```

Восстановление пути по дереву кратчайших путей

Поиск в ширину

Программа `PathFinder` (листинг 4.5.4) реализует API нахождения кратчайших путей из одной начальной вершины на основе только что рассмотренного алгоритма. Программа создает две таблицы символов: в одной хранятся расстояния от начальной вершины до каждой другой вершины, а во второй — предыдущая остановка в кратчайшем пути от начальной вершины до остальных. Конструктор использует очередь FIFO для хранения обнаруженных вершин (соседей вершин, кратчайший путь к которым был найден, но соседи еще не были проверены). Этот процесс называется *поиском в ширину* (BFS, Breadth-First Search), так как он просматривает граф «по горизонтали». С другой стороны, другой важный метод поиска в графе, называемый *поиском в глубину*, основан на рекурсивном методе вроде того, что мы использовали для проверки протекания в листинге 2.4.5, а граф просматривается «по вертикали». Поиск в глубину обычно находит длинные пути; поиск в ширину заведомо находит кратчайшие пути.

Быстродействие

Эффективность алгоритмов обработки графов обычно зависит от двух параметров графа: количества вершин `V` и количества ребер `E`. В реализации из `PathFinder` время, необходимое для поиска в ширину, связано с размером ввода линейно-логарифмической зависимостью и в худшем случае пропорционально $E \log V$. Чтобы убедиться в этом, сначала заметьте, что внешний цикл (`while`) выполняется не более V раз (по одному для каждой вершины), потому что мы следим за тем, чтобы каждая вершина ставилась в очередь не более одного раза Затем заметим, что внутренний цикл (`for`) выполняется не более $2E$ раз по всем итерциям, потому что в этом алгоритме каждое ребро проверяется не более двух раз, по одному для каждой из двух вершин. Каждая итерация цикла требует не менее одной операции `contains()` и, возможно, двух операций `put()` в таблицах символов размером не более V. Это линейно-логарифмическое быстродействие зависит от использования

Листинг 4.5.4. Реализация кратчайших путей

```
public class PathFinder
{
   private ST<String, Integer> dist;
   private ST<String, String>  prev;

   public PathFinder(Graph G, String s)
   {  // Использование поиска в ширину для вычислений кратчайшего пути
      // от начальной вершины s до всех остальных вершин графа G.
      prev = new ST<String, String>();
      dist = new ST<String, Integer>();
      Queue<String> queue = new Queue<String>();
      queue.enqueue(s);
      dist.put(s, 0);
      while (!queue.isEmpty())
      {  // Обработка следующей вершины в очереди.
         String v = queue.dequeue();
         for (String w : G.adjacentTo(v))
         {  // Проверка уже известных расстояний.
            if (!dist.contains(w))
            {  // Добавление в очередь и сохранение кратчайшего пути.
               queue.enqueue(w);
               dist.put(w, 1 + dist.get(v));
               prev.put(w, v);
            }
         }
      }
   }

   public int distanceTo(String v)
   {  return dist.get(v);  }

   public Iterable<String> pathTo(String v)
   {  // Вершины кратчайшего пути из s в v.
      Stack<String> path = new Stack<String>();
      while (v != null && dist.contains(v))
      {  // Вставка текущей вершины и переход к предыдущей вершине в пути.
         path.push(v);
         v = prev.get(v);
      }
      return path;
   }
}
```

`dist`	расстояние от s
`prev`	предыдущая вершина на кратчайшем пути из s

`G`	граф
`s`	начальная вершина
`q`	очередь вершин
`v`	текущая вершина
`w`	соседи v

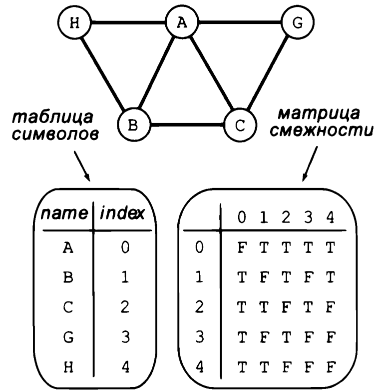
`PatchFinder()`	конструктор для s в G
`distanceTo()`	расстояние от s до v
`pathTo()`	путь из s в v

Класс использует поиск в ширину для вычисления кратчайших путей от заданной начальной вершины s до каждой другой вершины в графе G (пример клиента приведен в листинге 4.5.3).

таблицы символов, базирующейся на бинарных деревьях поиска (такой, как `ST` или `java.util.TreeMap`), с логарифмическим временем поиска и вставки. Использование таблицы символов на основе хеш-таблицы (такой, как `java.util.HashMap`) сокращает время выполнения до линейной зависимости от размера входных данных (пропорционально E для типичных графов).

Матрица смежности

Хорошее быстродействие алгоритмов обработки графов в значительной мере зависит от специально спроектированных структур данных, так что не стоит считать, что оно обеспечивается автоматически. Например, альтернативное представление графов — так называемая *матрица смежности* — использует таблицу символов для связывания имен вершин с целыми числами от 0 до *V*21, а затем создает массив $V \times V$ из элементов `boolean`. Элемент на пересечении строки `i` и столбца `j` равен `true`, если вершина, соответствующая `i`, соединена ребром с вершиной, соответствующей `j`; если такого ребра не существует, элемент равен `false`. Похожее представление уже использовалось в книге при изучении модели случайного серфинга для ранжирования веб-страниц в разделе 1.6. Представление матрицы смежности выглядит просто, но для очень больших графов оно неприемлемо — у графа с миллионом вершин матрица смежности состоит из триллиона элементов. Понимание этой особенности позволяет распознать возможность или невозможность решения конкретной практической задачи.



Представление графа в виде матрицы смежности

Поиск в ширину — фундаментальный алгоритм, с помощью которого можно проложить маршрут на карте воздушных перелетов, спланировать переезд в городском метрополитене (см. упражнение 4.5.38) или во многих других аналогичных ситуациях. Как показывает наш пример со степенями разделения, он также находит применение во многих других областях, от индексирования веб-страниц и маршрутизации пакетов в Интернете до изучения инфекционных заболеваний, моделирования работы мозга и отношений между геномными последовательностями. Во многих из этих применений задействованы большие графы, поэтому эффективность алгоритма исключительно важна.

В одном важном обобщении задачи нахождения кратчайшего пути каждому ребру назначается вес и требуется найти путь с минимальной суммой весов ребер. Если позднее вы будете проходить курсы алгоритмов или исследования операций, вы познакомитесь с обобщенным вариантом задачи поиска в ширину, известным под названием *алгоритма Дейкстры*; он решает задачу за линейно-логарифмическое время. При получении указаний с устройства GPS или с картографического приложения в сети алгоритм Дейкстры лежит в основе решения сопутствующих задач нахождения кратчайшего пути. Например, в основе построения кратчайшего маршрута с помощью GPS-навигатора или картографического приложения лежит алгоритм Дейкстры. Подобные важные, повсеместно встречающиеся применения составляют всего лишь верхушку айсберга, потому что область применения моделей с графами не ограничивается картографией.

Графы «тесного мира»

Ученые выделили подкласс графов, представляющих отдельный интерес, — так называемые графы «тесного мира», встречающиеся во многих областях естественных и общественных наук. Графы «тесного мира» характеризуются следующими тремя особенностями.

- *Разреженность*: реальное количество ребер намного меньше потенциально возможного для графа с заданным количеством вершин.
- *Короткие средние пути*: если выбрать две случайные вершины, длина кратчайшего пути между ними невелика.
- *Локальная кластеризация*: если две вершины являются соседями третьей вершины, то эти две вершины с большой вероятностью являются соседями друг друга.

Если граф обладает этими тремя свойствами, говорят, что он являет собой *феномен «тесного мира»*. Под термином «тесный мир» имеется в виду, что подавляющая часть вершин обладает как локальной кластеризацией, так и короткими путями к другим вершинам. Слово «*феномен*» указывает на то, что неожиданно очень многие графы, встречающиеся на практике, разреженны, обладают локальной кластеризацией и содержат короткие пути. Помимо только что рассмотренных примеров из области социальных отношений, графы «тесного мира» применимы для изучения маркетинга продуктов или идей, формирования и распространения слухов, анализа Интернета, построения безопасных одноранговых сетей, разработки алгоритмов маршрутизации и беспроводных сетей, проектирования энергетических сетей, моделирования обработки информации человеческим мозгом, изучения колебательных процессов при фазовых переходах, распространения вирусов (в живых организмах и компьютерных сетях) и т. д. Интенсивные исследования в области анализа феномена «тесного мира» идут начиная с фундаментальной работы Уоттса и Строгаца, опубликованной в 1990-е годы.

Ключевой вопрос в таких исследованиях звучит так: как для заданного графа определить, является ли он графом «тесного мира»? Чтобы ответить на этот вопрос, мы сначала установим дополнительные условия: граф достаточно большой (допустим, 1000 вершин и более) и связный (для любой пары вершин существует соединяющий их путь). Затем необходимо установить пороговые значения для каждого из свойств «тесного мира».

- *Разреженность*: средняя степень вершины меньше $20 \lg V$.
- *Короткие средние пути*: средняя длина кратчайшего пути между двумя вершинами меньше $10 \lg V$.
- *Локальная кластеризация*: некоторая величина, называемая *коэффициентом кластеризации*, должна быть больше 10%.

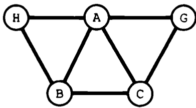
Определение локальной кластерзации несколько сложнее определений разреженности и средней длины путей. На интуитивном уровне коэффициент кластеризации вершины представляет вероятность того, что при случайном выборе ее двух соседей

они также будут соединены ребром. А если говорить точнее, у вершины, имеющей t соседей, существуют $t(t-1)/2$ возможных ребер, которые соединяют этих соседей; ее *коэффициент локальной кластеризации* определяется как отношение фактического числа ребер между соседями вершины к этому потенциально возможному либо как 0, если степень вершины равна 0 или 1[1]. *Коэффициент кластеризации графа* определяется как среднее значение коэффициентов локальной кластеризации по всем его вершинам. Если среднее значение больше 10%, мы считаем, что граф обладает локальной кластеризацией. На следующей диаграмме приведен пример вычисления этих трех характеристик для небольшого графа.

средняя степень вершины

вершина	*степень*
A	4
B	3
C	3
G	2
H	2
total	14

средняя степень = 14/5 = 2.8



средняя длина пути

пара вершин	*кратчайший путь*	*длина*
A B	A-B	1
A C	A-C	1
A G	A-G	1
A H	A-H	1
B C	B-C	1
B G	B-A-G	2
B H	B-H	1
C G	C-G	1
C H	C-A-H	2
G H	G-A-H	2
	итого	13

$$\frac{\text{сумма длин}}{\text{количество пар}} = 13/10 = 1.3$$

коэффициент кластеризации

вершина	*степень*	*количество ребер среди соседей: фактическое*	*потенциальное*
A	4	3	6
B	3	2	3
C	3	2	3
G	2	1	1
H	2	1	1

$$\frac{3/6 + 2/3 + 2/3 + 1/1 + 1/1}{5} \approx 0.767$$

Вычисление характеристик графа «тесного мира»

Чтобы вы лучше познакомились с этими определениями, мы рассмотрим несколько простых моделей графов и выясним, описывают ли они графы «тесного мира», проверив все три обязательных свойства.

Полные графы

Полный граф с V вершинами содержит $V(V-1)/2$ ребер, каждое из которых соединяет одну пару вершин. Полные графы не являются графами «тесного мира». Они обладают малой средней длиной пути (каждый кратчайший путь имеет длину 1), проявляют свойство локальной кластеризации (коэффициент кластеризации равен 1), но не являются разреженными (средняя степень вершины равна $V-1$, что намного больше $20 \lg V$ для больших V).

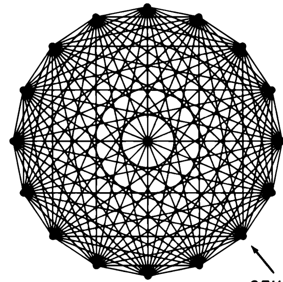
[1] Выше было определено, что граф «тесного мира» — связный, поэтому вершин со степенью 0 в нем быть не должно. Но сейчас речь идет о *проверке* графов, для которых выполнение условий *еще* не подтверждено. — *Примеч. науч. ред.*

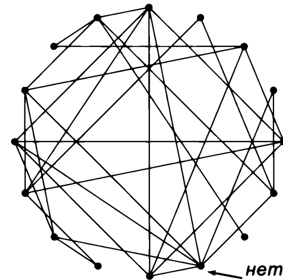
Кольцевые графы

Кольцевой граф представляет собой множество из V вершин, равномерно распределенных на окружности; каждая вершина имеет две смежные вершины (по одной с каждой из сторон). У k-кольцевого графа каждая вершина является смежной с k ближайшими вершинами с каждой стороны. На диаграмме изображен 2-кольцевой граф с 16 вершинами. Кольцевые графы также не являются графами «тесного мира». Например, 2-кольцевые графы разрежены (степень каждой вершины равна 4) и обладают локальной кластеризацией (коэффициент кластеризации равен 1/2), но их средние пути не являются короткими (см. упражнение 4.5.20).

Случайные графы

Модель Эрдёша — Реньи — хорошо изученная модель генерирования случайных графов. В этой модели для построения случайного графа с V вершинами каждое возможное ребро включается с вероятностью p. Случайные графы с достаточным количеством ребер с очень высокой вероятностью являются связными и имеют короткие средние пути, но они не являются графами «тесного мира», потому что не обладают локальной кластеризацией (см. упражнение 4.5.46).





Три модели графов

Эти примеры демонстрируют, что разработать модель графа, обладающую всеми тремя свойствами, является непростым делом. Попробуйте создать модель графа, которая, по вашему мнению, решает эту задачу. Немного поразмыслив над этой задачей, вы поймете, что в этих вычислениях вам бы помогла программа. Также приведенное выше утверждение о том, что графы «тесного мира» часто встречаются на практике, начинает казаться удивительным. Существует ли вообще хоть один граф «тесного мира»?

Выбор 10% в качестве порога кластеризации вместо какого-либо другого фиксированного процента кажется произвольным (как и выбор $20 \lg V$ для порога разреженности, или $10 \lg V$ для порога кратчайших путей), но часто мы даже близко не подходим к этим граничным значениям. Для примера возьмем *веб-граф*, в котором каждая веб-страница представлена вершиной, а вершины соединяются ребром, если они связаны ссылкой. По оценкам ученых, для перехода по

ссылкам с одной страницы на любую другую редко требуется более 30 переходов. Так как количество веб-страниц исчисляется миллиардами, эта оценка подразумевает, что средняя длина пути очень мала, намного ниже порога 10 lg V (около 300 для 1 миллиарда вершин).

модель	разреженный?	короткие пути?	локальная кластеризация?
полный	–	+	+
2-кольцевой	+	–	+
случайный	+	+	–

Даже после согласования всех определений проверка того, проявляет ли заданный граф феномен «тесного мира», может стать существенным вычислительным бременем. Как вы, вероятно, догадывались, рассмотренные нами типы данных для работы с графами предоставляют именно те средства, которые нам нужны. Программа `SmallWorld` (листинг 4.5.5) является клиентом `Graph` и `PathFinder`, реализующим эти проверки. Без эффективных структур данных и алгоритмов, которые рассматривались ранее, эти вычисления могут потребовать неприемлемо больших затрат ресурсов. Впрочем, даже с ними для больших графов (таких, как movies.txt) приходится прибегать к статистической выборке для оценки средней длины пути и коэффициента кластеризации за разумное время (см. упражнение 4.5.44), потому что функции `averagePathLength()` и `clusteringCoefficient()` выполняются за квадратичное время.

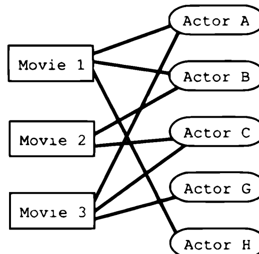
Классический граф «тесного мира»

Наш граф с фильмами и актерами не является графом «тесного мира», потому что он является двудольным, и, следовательно, его коэффициент кластеризации равен 0. Кроме того, некоторые пары актеров не связываются друг с другом ни одним путем. Однако простой граф актеров, в котором два актера соединяются ребром в том случае, если они снимались в одном фильме, является классическим примером графа «тесного мира» (после исключения актеров, не связанных с Кевином Бэйконом). На следующей диаграмме изображены графы «фильм — актер» и «актер — актер» для крошечного файла данных.

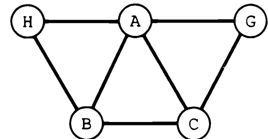
файл данных

```
% more tinyMovies.txt
Movie 1/Actor A/Actor B/Actor H
Movie 2/Actor B/Actor C
Movie 3/Actor A/Actor C/Actor G
```

граф «фильм — актер»



граф «актер — актер»



Два разных представления файла данных в виде графа

Листинг 4.5.5. Проверка графа «тесного мира»

```
public class SmallWorld
{
   public static double averageDegree(Graph G)
   {  return 2.0 * G.E() / G.V();  }

   public static double averagePathLength(Graph G)
   {  // Вычисление среднего расстояния между вершинами.
      int sum = 0;
      for (String v : G.vertices())
      {  // Прибавление к суммарному расстоянию от v.
         PathFinder pf = new PathFinder(G, v);
         for (String w : G.vertices())
            sum += pf.distanceTo(w);
      }
      return (double) sum / (G.V() * (G.V() - 1));
   }

   public static double clusteringCoefficient(Graph G)
   {  // Вычисление коэффициента кластеризации.
      int total = 0;
      for (String v : G.vertices())
      {  // Вычисление коэффициента локальной кластеризации для вершины v.
         int possible = G.degree(v) * (G.degree(v) - 1);
         int actual = 0;
         for (String u : G.adjacentTo(v))
            for (String w : G.adjacentTo(v))
               if (G.hasEdge(u, w)) actual++;
         if (possible > 0)
            total += 1.0 * actual / possible;
      }
      return (double) total / G.V();
   }

   public static void main(String[] args)
   {  /* См. упражнение 4.5.24. */  }
```

G	граф
sum	накапливаемая сумма расстояний между вершинами
v	итератор для вершин
w	соседи v

G	граф
possible	накапливаемая сумма возможных локальных ребер
actual	накапливаемая сумма фактических локальных ребер
delta	смещение
v	итератор для вершин
u, w	соседи v

Клиент читает граф из стандартного потока ввода и вычисляет значения различных параметров графа для проверки того, проявляет ли граф феномен «тесного мира».

```
% java SmallWorld "/" tinyGraph.txt
5 vertices, 7 edges
average degree         = 2.800
average path length    = 1.300
clustering coefficient = 0.767
```

Программа `Performer` (листинг 4.5.6) строит граф «актер — актер» на основе файла данных. Напомним, что каждая строка файла состоит из названия фильма, за которым перечисляются все актеры, снимавшиеся в этом фильме (имена разделяются символом `/`). `Performer` добавляет ребро между каждой парой актеров, снимавшихся в этом фильме. При проведении обработки для каждого фильма во входных данных создается граф, связывающий актеров.

Листинг 4.5.6. Граф «актер — актер»

```
public class Performer
{
   public static void main(String[] args)
   {
      String filename  = args[0];
      String delimiter = args[1];
      Graph G = new Graph();

      In in = new In(filename);
      while (in.hasNextLine())
      {
         String line = in.readLine();
         String[] names = line.split(delimiter);
         for (int i = 1; i < names.length; i++)
            for (int j = i+1; j < names.length; j++)
               G.addEdge(names[i], names[j]);
      }

      double degree  = SmallWorld.averageDegree(G);
      double length  = SmallWorld.averagePathLength(G);
      double cluster = SmallWorld.clusteringCoefficient(G);
      StdOut.printf("number of vertices     = %7d\n", G.V());
      StdOut.printf("average degree         = %7.3f\n", degree);
      StdOut.printf("average path length    = %7.3f\n", length);
      StdOut.printf("clustering coefficient = %7.3f\n", cluster);
   }
}
```

G	граф
in	входной поток
line	одна строка файла
names[]	фильм и актеры
i, j	индексы двух актеров

Программа-клиент SmallWorld получает в аргументах командной строки имя файла данных и разделитель и создает соответствующий граф «актер — актер». Программа направляет в стандартный поток вывода количество вершин, среднюю их степень, среднюю длину пути и коэффициент кластеризации графа. Предполагается, что граф «актер — актер» является связным (см. упражнение 4.5.29); без выполнения этого условия средняя длина пути не определена.

```
% java Performer tinyMovies.txt "/"
number of vertices     = 5
average degree         =   2.800
average path length    =   1.300
clustering coefficient =   0.767
```

```
% java Performer moviesG.txt "/"
number of vertices     = 19044
average degree         = 148.688
average path length    =   3.494
clustering coefficient =   0.911
```

Так как граф «актер — актер» обычно содержит много больше ребер, чем соответствующий граф «фильм — актер», мы пока будем работать с меньшим графом «актер — актер», созданным на базе файла moviesG.txt, содержащего 1261 фильм и 19 044 актера (которые все соединены с Кевином Бэйконом). Программа `Performer` сообщает, что граф «актер — актер», связанный с moviesG.txt, содержит 19 044 вершин и 1 415 808 ребер, так что средняя степень вершины равна 148,7 (около половины от 20 lg V = 284,3), что означает, что граф является разреженным. Средняя длина пути равна 3,494 (много меньше 10 lg V = 142,2), то есть граф

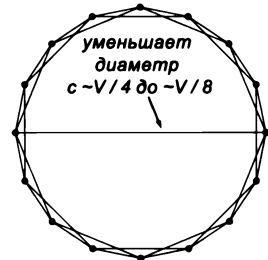
имеет короткие пути. Коэффициент кластеризации равен 0,911, а значит, в графе существует локальная кластеризация. Мы нашли граф «тесного мира»! Эти вычисления подтверждают гипотезу о том, что подобные графы социальных отношений проявляют феномен «тесного мира». Попробуйте поискать другие реальные графы и проверить их в программе `SmallWorld`.

Один из подходов к анализу таких явлений, как феномен «тесного мира», основан на разработке математической модели, которая может использоваться для проверки гипотез и построения прогнозов. Напоследок мы вернемся к задаче разработки модели графа, которая поможет лучше понять феномен «тесного мира». Хитрость разработки такой модели заключается в объединении двух разреженных графов: 2-кольцевого графа (с высоким коэффициентом кластеризации) и случайного графа (с малой средней длиной пути).

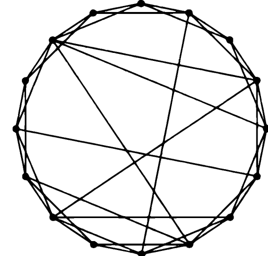
Кольцевые графы со случайными сокращениями

Одно из самых удивительных следствий из работы Уоттса и Строгаца заключается в том, что добавление относительно небольшого количества случайных ребер в разреженный граф с локальной кластеризацией создает граф «тесного» мира. Чтобы получить некоторое представление о том, почему это происходит, рассмотрим 2-кольцевой граф, у которого диаметр (длина пути между наиболее удаленной парой вершин) равен ~$V/4$ (см. рисунок). Добавление одного ребра, соединяющего антиподальные (противолежащие) вершины, сокращает диаметр до ~$V/8$ (см. упражнение 4.5.21). Добавление $V/2$ случайных «сокращений» в 2-кольцевой граф с очень высокой вероятностью приведет к значительному сокращению средней длины пути, в результате чего она становится логарифмической (см. упражнение 4.5.25). Средняя степень при этом увеличивается всего на 1, а коэффициент кластеризации не опускается заметно ниже 1/2. Иначе говоря, 2-кольцевой граф с $V/2$ случайными сокращениями с исключительно высокой вероятностью приведет к созданию графа «тесного мира»!

2-кольцевой граф с антиподальным ребром



2-кольцевой граф со случайными сокращениями



Новая модель графа

Генератор для построения графов на базе таких моделей прост в разработке, а программа `SmallWorld` поможет определить, проявляют ли такие графы феномен «тесного мира» (см. упражнение 4.5.24). Также можно проверить аналитические результаты, полученные для простых графов, таких как tinyGraph.txt, полных и кольцевых графов. Впрочем, как это часто бывает при научных исследованиях, новые вопросы появляются едва ли не быстрее, чем мы находим ответы на старые. Сколько случайных сокращений необходимо добавить для получения короткого среднего пути? Какими значениями средней длины пути и коэффициента кластеризации обладает слу-

чайный связный граф? Какие еще модели графов хорошо подойдут для анализа? Сколько выборок понадобится для точной оценки коэффициента кластеризации или средней длины пути в очень большом графе? В упражнениях вы найдете подсказки для поиска ответов на подобные вопросы и для дальнейшего изучения феномена «тесного мира».

модель	средняя степень	средняя длина пути	коэффициент кластеризации
полный граф	999 –	1 +	1,0 +
2-кольцевой граф	4 +	125,38 –	0,5 +
случайный связный граф с p = 10/V	10 +	3,26 +	0,010 –
2-кольцевой граф с V/2 случайных сокращений	5 +	5,71 +	0,343 +

Параметры «тесного мира» для различных графов с 1000 вершинами

Выводы

Этот пример показывает, какую важную роль играют алгоритмы и структуры данных в научных исследованиях. Также он подтверждает некоторые уроки, вынесенные из этой книги, — которые стоит повторить.

Тщательно проектируйте типы данных

В книге мы постоянно возвращаемся к одному важному правилу: эффективное программирование основано на точном понимании множества допустимых значений типа данных и множества операций, определенных для этих значений. Современный объектно-ориентированный язык программирования, такой как Java, открывает путь к этому пониманию, потому что вы сами проектируете, строите и используете свои типы данных.

Наш тип данных Graph появился в результате многочисленных переработок и опыта применения тех решений, которые обсуждались в тексте. Ясность и простота клиентского кода наглядно убеждают, насколько важно серьезно относиться к проектированию и реализации базовых типов данных в любой программе.

Используйте поэтапную разработку

Как и во всех остальных примерах, мы строим программный продукт по модулям, тестируя и изучая каждый модуль перед тем, как переходить к следующему.

Решайте понятные задачи перед тем, как переходить к непонятным

Задача нахождения кратчайших путей была рассмотрена на примере с авиарейсами между несколькими городами — простом и понятном. Она достаточно сложна, чтобы разработчик не потерял интереса во время отладки и анализа трассировки, но не настолько, чтобы эти задачи стали излишне трудоемкими.

Продолжайте тестировать и проверять результаты

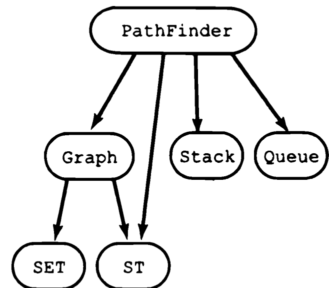
При работе со сложными программами, обрабатывающими большие объемы данных, внимательность при проверке результатов никогда не бывает лишней. Оценивайте каждый фрагмент выходных данных программы с позиций здравого смысла. Неопытные программисты склонны к оптимизму («Если программа выдает ответ, наверное, он правильный»); опытные программисты знают, что пессимистическое мировоззрение («Наверняка с этим результатом что-то не так») гораздо лучше.

Используйте реальные данные

Файл movies.txt из базы данных Internet Movie Database — всего лишь один пример файлов данных, которые сейчас повсеместно встречаются в Интернете. В прошлые годы такие данные часто скрывались за закрытыми или узкоспециализированными форматами, но большинство людей сейчас понимает, что простые текстовые форматы намного удобнее. Различные методы типа данных Java `String` упрощают работу с реальными данными, а это лучший способ формулировки гипотез о явлениях реального мира. Начните с малых файлов с реальными данными, чтобы протестировать программу и оценить ее быстродействие, перед тем как браться за большие файлы.

Повторно используйте готовый код

Еще одно правило, к которому мы постоянно возвращаемся к книге, — эффективное программирование основано на понимании фундаментальных типов данных, чтобы нам не приходилось заново писать код реализации базовой функциональности. Типичным примером такого рода служит использование `ST` и `SET` в `Graph` — многие программисты продолжают использовать при работе с графами низкоуровневые представления и реализации со связными списками и массивами. А это неизбежно заставляет их заново писать код простых операций, таких как управление связными списками и перемещения по ним. Наш класс кратчайших путей `PathFinder` использует `Graph`, `ST`, `SET`, `Stack` и `Queue` — целую обойму первоклассных фундаментальных структур данных.



Повторное использование кода `PathFinder`

Обеспечивайте гибкость

Повторное использование кода часто подразумевает использование классов из различных библиотек Java. Обычно такие классы имеют очень широкие интерфейсы (то есть содержат большое количество методов), поэтому всегда стоит определить и реализовать собственные API с узкими интерфейсами между клиентами и реализациями, даже если ваши реализации сводятся к простому вызову библиотечных методов Java. Такой подход обеспечивает гибкость, необходимую для переключения на более эффективные реализации в случае необходимости, и избегает зависимости от изменений в тех частях библиотеки, которые вы не собираетесь использовать. Например, использование `ST` в нашей реализации `Graph` (листинг 4.5.1) предоставляет возможность использования любых наших реализаций таблиц символов (таких, как `HashST` и `BST`) или же одной из их реализаций из библиотек Java (`java.util.TreeMap` и `java.util.HashMap`) без каких-либо изменений в коде `Graph`.

Следите за эффективностью

Без хороших алгоритмов и структур данных многие задачи, встречавшиеся в этой главе, остались бы нерешенными, потому что примитивные решения требуют недопустимых затрат времени или памяти. Очень важно представлять хотя бы приблизительные потребности ваших программ в ресурсах.

Разобранный пример станет хорошим завершением главы, потому что он наглядно показывает, что все рассмотренные программы — всего лишь отправная точка, а не завершенный результат. Описанные в этой главе приемы программирования также положат начало для дальнейшего изучения математики, техники или любой области, в которой вычисления играют важную роль (в наши дни это практически любая область). Методология программирования и инструменты, представленные в этой главе, подготовят вас к решению любых вычислительных задач.

К настоящему моменту вы освоили программирование на одном из современных языков и сейчас вполне готовы к тому, чтобы оценить некоторые важные интеллектуальные концепции, связанные с вычислениями. Ваше понимание программирования поднимется на новый уровень, а полученные знания наверняка пригодятся вам, когда вы столкнетесь с этими концепциями в будущем. Мы отправимся в это путешествие в следующей главе.

Вопросы и ответы

В. Сколько существует разных графов для заданных V вершин?

О. Без петель или параллельных ребер существуют $V(V-1)/2$ возможных ребер, каждое из которых может либо присутствовать, либо не присутствовать в графе. Таким образом, общее количество составляет $2^{V(V-1)/2}$. Это значение очень быстро растет, как видно из следующей таблицы:

V	1	2	3	4	5	6	7	8	9
$2^{V(V-1)/2}$	1	2	8	64	1024	32 768	2 097 152	268 435 456	68 719 476 736

Эти огромные числа дают некоторое представление о сложности социальных отношений. Например, если просто взять 9 человек, которых вы встретили на улице, количество вариантов взаимных знакомств превышает 68 триллионов!

В. Может ли граф содержать вершину, не смежную ни с одной другой вершиной?

О. Хороший вопрос. Такие вершины называются *изолированными*, и в нашей реализации они запрещены. Другая реализация может разрешить создание изолированных вершин; для этого в нее добавляется отдельный метод `addVertex()` для создания вершины.

В. Можно использовать представление связного списка для соседей каждой вершины?

О. Можно, но скорее всего, вам придется заново реализовывать базовый код связного списка, когда выяснится, что вам нужна операция получения размера, итератор и т. д.

В. Почему методы `V()` и `E()` должны иметь реализации с фиксированным временем?

О. Может показаться, что большинство клиентов вызывает такие методы всего один раз. Однако в одной в высшей степени распространенной идиоме используется код вида:

```
for (int i = 0; i < G.E(); i++)
{  ...  }
```

Этот код будет выполняться за квадратичное время, если вместо хранения количества ребер в переменной экземпляра каждый раз будет задействоваться алгоритм подсчета ребер (см. упражнение 4.5.1).

В. Почему `Graph` и `PathFinder` определяются в разных классах? Разве не логичнее включить методы `PathFinder` в API `Graph`?

О. Нахождение кратчайших путей — всего лишь одна из многих задач обработки графов. Включение всех таких методов в один API — признак некачественного проектирования. Пожалуйста, перечитайте обсуждение широких интерфейсов в разделе 3.3.

Упражнения

4.5.1. Добавьте в класс `Graph` реализации `V()` и `E()`, которые возвращают соответственно количество вершин и ребер в графе. Убедитесь в том, что ваши реализации выполняются за фиксированное время. *Подсказка*: для `V()` можно считать,

что метод `size()` в `ST` выполняется за фиксированное время; для `E()` определите переменную экземпляра, в которой хранится текущее количество ребер в графе.

4.5.2. Добавьте в класс `Graph` метод `degree()`, который получает строковый аргумент и возвращает степень заданной вершины. Используйте этот метод для поиска в файле movies.txt актера, снявшегося в наибольшем количестве фильмов.

Ответ:

```
public int degree(String v)
{
   if (st.contains(v)) return st.get(v).size();
   else                return 0;
}
```

4.5.3. Добавьте в класс `Graph` метод `hasVertex()`, который получает строковый аргумент и возвращает `true`, если в графе существует вершина с таким именем, или `false` в противном случае.

4.5.4. Добавьте в класс `Graph` метод `hasEdge()`, который получает два строковых аргумента и возвращает `true`, если эти значения определяют ребро (соединяющее вершины с этими именами) в графе, или `false` в противном случае.

4.5.5. Создайте для `Graph` копирующий конструктор, который получает в аргументе граф `G`, создает и инициализирует новую независимую копию графа. Любые будущие изменения в `G` не должны отражаться на вновь созданном графе.

4.5.6. Напишите версию `Graph`, которая поддерживает явное создание вершин и допускает петли, параллельные ребра и изолированные вершины. *Подсказка*: используйте для хранения списков смежности `Queue` вместо `SET`.

4.5.7. Добавьте в класс `Graph` метод `remove()`, который получает два строковых аргумента и удаляет из графа заданное ими ребро (если оно существует).

4.5.8. Добавьте в класс `Graph` метод `subgraph()`, который получает в аргументе `SET<String>` и возвращает *индуцированный подграф* (граф, состоящий из заданных вершин и всех ребер исходного графа, соединяющих любые две из них).

4.5.9. Напишите версию `Graph`, которая поддерживает обобщенные типы вершин с поддержкой `Comparable` (простое задание). Затем напишите версию `PathFinder`, которая использует вашу реализацию для нахождения кратчайших путей с использованием обобщенных типов вершин с поддержкой `Comparable` (более сложное задание).

4.5.10. Создайте версию `Graph` из предыдущего упражнения для поддержки двудольных графов (графы, ребра которых соединяют вершину одного обобщенного типа с поддержкой `Comparable` с вершиной другого обобщенного типа с поддержкой `Comparable`).

4.5.11. *Возможно или нет*: в какой-то момент во время поиска в ширину очередь может содержать две вершины, у одной из которых расстояние от начальной вершины равно 7, а у другой это расстояние равно 9.

Ответ: нет. Очередь может содержать вершины не более чем с двумя разными расстояниями d и $d + 1$. Поиск в ширину проверяет вершины по возрастанию расстояния от начальной. При проверке вершины на расстоянии d в очередь могут ставиться только вершины с расстоянием $d - 1$.

4.5.12. Докажите посредством индукции, что `PathFinder` вычисляет кратчайшие пути (и расстояния по кратчайшим путям) от начальной вершины до всех остальных вершин.

4.5.13. Допустим, для проведения поиска в ширину в `PathFinder` используется стек вместо очереди. Будет ли при этом вычисляться путь от начальной вершины до всех остальных? Будут ли при этом вычисляться кратчайшие пути? В каждом случае докажите или приведите контрпример.

4.5.14. К чему приведет использование очереди вместо стека при формировании кратчайшего пути в `pathTo()`?

4.5.15. Добавьте в класс `PathFinder` метод `isReachable(v)`, который возвращает `true`, если существует путь от начальной вершины к вершине `v`, и `false` в противном случае.

4.5.16. Напишите клиент `Graph`, который читает данные `Graph` из файла (в формате, описанном в тексте), а затем выводит список ребер графа, по одному на строку.

4.5.17. Реализуйте клиент `PathFinder` с именем `AllShortestPaths`, который создает объект `PathFinder` для каждой вершины, с тестовым клиентом, который получает из стандартного ввода запросы с двумя вершинами и выводит кратчайший путь, соединяющий эти вершины. Обеспечьте поддержку разделителя, чтобы запрос с двумя вершинами можно было ввести в одной текстовой строке. *Примечание*: для файла movies.txt в запросе могут быть указаны как два актера, так и два фильма или же один актер и один фильм.

4.5.18. Напишите программу, которая строит график зависимости средней длины пути в 2-кольцевом графе из 1000 вершин от количеста добавленных в него случайных «сокращающих» ребер.

4.5.19. Добавьте в программу `SmallWorld` (листинг 4.5.5) перегруженную функцию `clusterCoefficient()`, которая получает целочисленный аргумент `k` и вычисляет коэффициент локальной кластеризации для графа на основании фактического и теоретически возможного количества ребер по множеству вершин в пределах расстояния k от каждой вершины. Для $k = 1$ функция должна выдавать такой же результат, что и версия функции без аргумента.

4.5.20. Покажите, что коэффициент кластеризации в k-кольцевом графе равен $(2k - 2)/(2k - 1)$. Выведите формулу для средней длины пути в k-кольцевом графе с V вершинами как функции V и k.

4.5.21. Покажите, что диаметр 2-кольцевого графа с V вершинами равен $\sim V/4$. Покажите, что при добавлении одного ребра, соединяющего две антиподальные (противолежащие) вершины, диаметр уменьшается до $\sim V/8$.

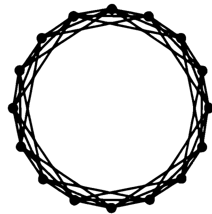
4.5.22. Проведите вычислительные эксперименты, доказывающие, что средняя длина пути в кольцевом графе с V вершинами равна ~$1/4V$. Повторите эти эксперименты, но добавьте одно случайное ребро в кольцевой граф и убедитесь в том, что средняя длина пути уменьшается до ~$3/16V$.

4.5.23. Добавьте в программу `SmallWorld` (листинг 4.5.5) функцию `isSmallWorld()`, которая получает в качестве аргумента граф и возвращает `true`, если граф проявляет феномен «тесного мира» (в соответствии с определениями в тексте), или `false` в противном случае.

4.5.24. Реализуйте для программы `SmallWorld` (листинг 4.5.5) тестовый клиентский метод `main()`, который выводит результат, приведенный в тексте. Программа получает как аргументы командной строки имя файла с данными графа и разделитель; выводит количество вершин, среднюю степень, среднюю длину пути и коэффициент кластеризации графа; а также сообщает, не слишком ли эти значения велики (или малы) для проявления феномена «тесного мира».

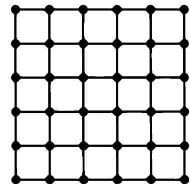
4.5.25. Напишите программу, которая генерирует случайные связные графы и 2-кольцевые графы со случайными сокращениями. Используя программу `SmallWorld`, сгенерируйте 500 случайных графов обоих типов (по 1000 вершин каждый) и вычислите для них среднюю степень вершин, среднюю длину пути и коэффициент кластеризации. Сравните свои результаты с соответствующими значениями из таблицы на с. 676.

4.5.26. Напишите приложение-клиент `SmallWorld` и `Graph`, которое генерирует k-кольцевые графы и проверяет, являются ли они графами «тесного мира» (сначала выполните упражнение 4.5.23).



3-кольцевой граф

4.5.27. В *решетчатом графе* вершины образуют решетку $n \times n$, а ребра соединяют каждую вершину с ее соседними узлами: сверху, снизу, слева и справа. Напишите приложение-клиент `SmallWorld` и `Graph`, которое генерирует решетчатые графы и проверяет, являются ли они графами «тесного мира» (сначала выполните упражнение 4.5.23).



Решетчатый граф 6x6

4.5.28. Расширьте свои решения двух предыдущих упражнений, чтобы они также получали аргумент командной строки m и добавляли в граф m случайных ребер. Поэкспериментируйте в своих программах с графами, содержащими около 1000 вершин, чтобы найти графы «тесного мира» с относительно небольшим количеством вершин.

4.5.29. Напишите приложение-клиент `Graph` и `PathFinder`, которое получает в качестве аргументов имя файла данных о фильмах и разделитель. Клиент должен записывать новый файл данных, из которого исключены все фильмы, не связанные с Кевином Бэйконом.

Упражнения повышенной сложности

4.5.30. *Большие числа Бэйкона.* Найдите в movies.txt актеров с наибольшим, но конечным числом Кевина Бэйкона.

4.5.31. *Гистограмма.* Напишите программу `BaconHistogram`, которая выводит гистограмму чисел Кевина Бэйкона, показывающую, сколько актеров из movies.txt имеет число Бэйкона 0, 1, 2, 3, ... Включите категорию для актеров с бесконечным числом (то есть вообще не связанных с Кевином Бэйконом).

4.5.32. *Граф «актер — актер».* Как упоминалось в тексте, альтернативный подход к вычислению чисел Кевина Бэйкона основан на построении графа, в котором каждый актер (но не фильм) представлен вершиной, а два актера соединяются ребром, если они снимаются в одном фильме (см. листинг 4.5.6). Вычислите числа Кевина Бэйкона, выполнив поиск в ширину по графу «актер — актер». Сравните время выполнения со временем выполнения для файла movies.txt. Объясните, почему это решение работает настолько медленнее. Также объясните, почему необходимо включать фильмы, находящиеся на пути, как это происходит автоматически в нашей первой реализации.

4.5.33. *Компоненты связности. Компонентой связности* в графе называется максимальное множество взаимосвязанных вершин[1]. Напишите клиент для `Graph` с именем `CCFinder`, который находит компоненты связности графа. Включите конструктор, который получает `Graph` в аргументе и находит все компоненты связности методом поиска в ширину. Добавьте метод `areConnected(v, w)`, который возвращает `true`, если вершины `v` и `w` принадлежат одной связной компоненте, или `false` в противном случае. Также добавьте метод `components()`, который возвращает количество компонент связности.

4.5.34. *Заливка/обработка изображений.* Класс `Picture` содержит двумерный массив значений `Color` (см. раздел 3.1) для представления пикселов. *Пятном* называется совокупность соседних пикселов одного цвета. Напишите класс-клиент `Graph`, конструктор которого создает решетчатый граф (см. упражнение 4.5.27) для заданного изображения и поддерживает операцию *заливки*. Для заданных координат пиксела `col` и `row` и цвета `color` цвет этого пиксела и всех пикселов, входящих в то же пятно, заменяется цветом `color`.

4.5.35. *Цепочки слов.* Напишите программу `WordLadder`, которая получает как аргументы командной строки две строки из пяти символов, читает из стандартного ввода список слов из пяти букв и выводит кратчайшую цепочку, связывающую две строки (если такая цепочка существует). Два слова занимают соседние позиции в цепочке, если они различаются ровно одной буквой. Например, следующая цепочка связывает слова `green` и `brown`:

```
green greet great groat groan grown brown
```

[1] «Максимальность» в данном случае означает, что к этому множеству нельзя добавить больше ни одну вершину. Но самих компонент связности у графа, очевидно, может быть несколько. — *Примеч. науч. ред.*

Напишите фильтр для получения пятибуквенных слов из системного словаря или загрузите список с сайта книги. (Изобретателем этой игры является Льюис Кэрролл.)

4.5.36. *Все пути.* Напишите клиент `Graph` с именем `AllPaths`, конструктор которого получает в аргументе объект `Graph`. Клиент должен поддерживать операции подсчета или вывода всех простых путей между двумя заданными вершинами `s` и `t` в графе. *Простым путем* называется путь, в котором вершины не повторяются. На двумерной сетке такие пути называются *блужданиями без самопересечений* (см. раздел 1.4). Перечисление путей относится к числу фундаментальных задач статической физики и теоретической химии — например, оно используется для моделирования пространственного расположения молекул линейных полимеров в растворе. *Предупреждение*: рост количества путей может быть экспоненциальным.

4.5.37. *Порог протекания.* Разработайте граф, моделирующий задачу протекания, и напишите клиент `Graph`, который выполняет те же вычисления, что и `Percolation` (листинг 2.4.5). Оцените порог протекания для треугольной, квадратной и шестигранной решетки.

4.5.38. *Граф метро.* В токийском метро линии помечаются буквами, а станции — цифрами (например, G-8 или A-3). Станция с возможностью пересадки представляется как множество станций. Найдите в Интернете схему токийского метро, разработайте простой файловый формат и напишите приложение-клиент `Graph` для чтения файла и обработки запросов на нахождение кратчайшего пути в токийском метро. При желании решите задачу для парижского метро, в котором линии представлены последовательностью названий, а пересадки возможны в том случае, если две станции имеют одинаковые названия.

4.5.39. *Центр голливудской Вселенной.* Чтобы понять, насколько хорошим центром графа может стать Кевин Бэйкон, можно вычислить *голливудское число* каждого актера. Голливудским числом Кевина Бэйкона является среднее число Бэйкона по всем актерам (из его компоненты связности). Голливудское число другого актера вычисляется аналогичным образом, но начальной точкой пути вместо Кевина Бэйкона становится этот актер. Вычислите голливудское число Кевина Бэйкона и найдите актера с лучшим голливудским числом, чем у Кевина Бэйкона. Найдите актеров (в одной компоненте связности с Кевином Бэйконом) с лучшим и худшим голливудским числом.

4.5.40. *Диаметр. Эксцентриситет* вершины графа определяется как наибольшее расстояние до максимально удаленной от нее вершины. *Диаметром* графа называется максимальное расстояние между двумя вершинами (максимальный эксцентриситет любой вершины). Напишите клиент `Graph` с именем `Diameter`, который может вычислить эксцентриситет вершины и диаметр графа. Используйте его для определения диаметра графа «актер — актер», построенного по данным из movies.txt.

4.5.41. *Направленные графы.* Реализуйте тип данных `Digraph` для представления *направленных графов*, у которых учитывается направление ребра: вызов `addEdge(v,w)` создает ребро из вершины `v` в вершину `w`, но не из `w` в `v`. Замените `adjacentTo()` двумя методами: первый возвращает множество вершин, к которым ведут напра-

ленные ребра из вершины-аргумента, а второй возвращает множество вершин, из которых ведут направленные ребра в вершину-аргумент. Объясните, как нужно изменить `PathFinder` для нахождения кратчайших путей в направленных графах.

4.5.42. *Случайный серфинг.* Модифицируя класс `Digraph` из предыдущего упражнения, создайте класс `MultiDigraph` с поддержкой параллельных ребер. Проведите в тестовом клиенте моделирование случайного серфинга по образцу `RandomSurfer` (листинг 1.6.2).

4.5.43. *Транзитивное замыкание.* Напишите приложение-клиент `Digraph` с именем `TransitiveClosure`, конструктор которого получает в аргументе объект `Digraph`, а метод `isReachable(v, w)` возвращает `true`, если существует направленный путь из `v` в `w`, или `false` в противном случае. *Подсказка*: выполните поиск в ширину от каждой вершины.

4.5.44. *Статистическая выборка.* Используйте статистическую выборку для оценки средней длины пути и коэффициента кластеризации графа. Например, для оценки коэффициента кластеризации выберите `trials` случайных вершин и вычислите среднее значение коэффициентов кластеризации этих вершин. Ваши функци должны работать на порядки быстрее соответствующих функций из `SmallWorld`.

4.5.45. *Время покрытия.* Случайное блуждание в ненаправленном связном графе предполагает переходы от вершины к одной из соседних с ней, при этом все варианты выбираются с равной вероятностью. (Этот процесс может рассматриваться как аналог случайного серфинга для ненаправленных графов.) Напишите программы для проведения экспериментов, на основании которых можно будет сформулировать гипотезу о количестве шагов для посещения каждой вершины в графе. Как выглядит время покрытия для полного графа с V вершинами? Для кольцевого графа? Сможете ли вы найти семейство графов, у которых время покрытия растет пропорционально V^3 или 2^V?

4.5.46. *Модель случайного графа Эрдёша — Реньи.* В классической модели случайного графа Эрдёша — Реньи случайный граф с V вершинами строится включением каждого возможного ребра с вероятностью p независимо от других ребер. Создайте приложение-клиент `Graph` для проверки следующих свойств.

- *Пороги связности*: если $p < 1/V$ при больших V, то большинство компонент связности имеет малый размер, а самые большие имеют логарифмический размер. Если $p > 1/V$, то почти наверняка существует огромная компонента, содержащая почти все вершины. Если $p > \ln V/V$, то граф с высокой вероятностью является связным; если $p < \ln V/V$, то граф с высокой вероятностью связным не является.
- *Распределение степеней*: степени вершин подчиняются биномиальному распределению, центром которого является среднее значение, поэтому у большинства вершин одинаковые степени. Вероятность того, что вершина является смежной с k другими вершинами, убывает экспоненциально с ростом k.
- *Отсутствие точек концентрации*: максимальная степень вершины при постоянном p находится в не более чем логарифмической зависимости от V.

- *Отсутствие локальных кластеров*: если граф является разреженным и связным, коэффициент кластеризации близок к 0. Случайные графы не являются графами «тесного мира».
- *Короткие пути*: если $p > \ln V/V$, то диаметр графа (см. упражнение 4.5.40) является логарифмическим.

4.5.47. *Степенной закон для веб-ссылок.* Полустепени входов и выходов для страниц Всемирной паутины подчиняются степенному закону, который может моделироваться процессом предпочтительного присоединения. Допустим, каждая веб-страница имеет ровно одну исходящую ссылку. Страницы создаются последовательно, начиная с единственной страницы, которая ссылается на саму себя. С вероятностью $p < 1$ страница ведет на одну из существующих страниц, равномерно выбираемых случайным образом. С вероятностью $1 - p$ страница ведет на одну из существующих страниц, причем вероятность каждой такой страницы пропорциональна количеству входящих ссылок на эту страницу. Это правило отражает распространенную тенденцию: новые веб-страницы часто ведут на популярные страницы. Напишите программу для моделирования этого процесса, постройте гистограмму количества входящих ссылок.

Частичное решение. Доля страниц с полустепенью входа k пропорциональна $k^{1/(1 p)}$.

4.5.48. *Коэффициент глобальной кластеризации.* Добавьте в `SmallWorld` функцию для вычисления *коэффициента глобальной кластеризации* графа. Коэффициент глобальной кластеризации определяет условную вероятность того, что две случайные вершины, являющиеся соседями общей вершины, являются соседями друг друга. Найдите графы, для которых коэффициенты локальной и глобальной кластеризации различны.

4.5.49. *Модель графа Уоттса — Строгаца.* (См. упражнение 4.5.27 и упражнение 4.5.28.) Уоттс и Строгац предложили гибридную модель, которая содержит типичные связи между вершинами, расположенными поблизости (люди знают своих географических соседей), а также некоторые случайные ссылки дальних связей. Постройте график зависимости средней длины пути и коэффициента кластеризации при добавлении случайных ребер в решетчатый граф $n \times n$ для $n = 100$. Сделайте то же самое для k-кольцевых графов с V вершинами, для $V = 10\,000$ и различных значений k до $10 \log V$.

4.5.50. *Модель графа Боллобаса — Чуна.* Боллобас (Bollobás) и Чун (Chung) предложили гибридную модель, которая совмещает 2-кольцевой граф с V вершинами (при нечетном V) и случайное паросочетание. Паросочетанием называется граф, в котором степень каждой вершины равна 1. Чтобы сгенерировать случайное паросочетание, сгенерируйте случайную перестановку V вершин и добавьте ребро между вершинами i и $i + 1$ в перестановке. Определите степень каждой вершины для графов в этой модели. Используя программу `SmallWorld`, оцените среднюю длину пути и коэффициент локальной кластеризации для графов, сгенерированных по этой модели, для $V = 1000$.

Глава 5

Теория вычислений[1]

Реальные машины могут показаться сложными и, следовательно, плохо поддающимися анализу, но один тот факт, что мы строим и используем их, позволяет предположить, что мы можем понимать их основные свойства и особенности. В этой главе мы покажем, что при тщательном изучении возможностей и ограничений вычислительных машин между всеми известными их типами проявляется поразительное сходство, а мы получаем возможность рассматривать некоторые фундаментальные вопросы:

- Являются ли одни компьютеры более мощными, чем другие, принципиально, в силу своего внутреннего устройства?
- Какие задачи можно решать с помощью компьютера?
- Существуют ли пределы тому, на что способны компьютеры?
- Как ограничивается то, что может сделать компьютер, в условиях ограниченных ресурсов?

Это очень глубокие вопросы, и в прошлом веке математики приложили немало сил в поисках ответов на них. Как ни странно, ответы на них можно получить в результате тщательного анализа упрощенных, идеализированных абстрактных машин, которые тем не менее сохраняют важнейшие свойства настоящих современных компьютеров.

Если слово «теория» внушает вам страх — не отчаивайтесь. Мы будем работать с математическими моделями, достаточно простыми для понимания. В этой главе мы рассмотрим ряд красивых теорем, связанных с вычислениями, и докажем их истинность. Мы не будем учить вас доказывать математические теоремы (этой способности, как и программированию, можно научиться), но вы не должны жалеть времени на то, чтобы шаг за шагом разобраться в представленных рассуждениях. Заодно это повысит вашу квалификацию программиста, но наша главная цель — помочь вам понять некоторые основы компьютерных вычислений.

---

[1] Содержание главы в действительности шире, чем это удается отразить в заголовке: также затрагиваются основы организации вычислительных машин и «машинных» языков, некоторые фундаментальные алгоритмы, их трудоемкость и т. д. — *Примеч. науч. ред.*

Зачем вообще заниматься теоретическими вопросами? Разве не лучше оставить их специалистам по теории вычислений? Этот вопрос можно задать о любой научной дисциплине, и ответ для информатики будет таким же, как для физики, химии или биологии. Модели, используемые нами для анализа вычислений, фундаментальны, а выводы, сформулированные в результате их изучения, имеют глубокие последствия для мира, в котором мы живем. Многие люди считают сам факт того, что мы можем решать такие фундаментальные вопросы, удивительным и интересным. Возможно, после чтения этой главы вам даже захочется объяснить кому-нибудь из друзей или близких, почему все устроено именно так.

Почему программиста должна интересовать теория вычислений? В мире, в котором подростки стараются превратить свои навыки программирования в несметные богатства, ощущение всемогущества компьютеров витает в воздухе. Но, как ни странно, теория вычислений говорит, что эти ожидания определенно не соответствуют действительности. Невозможно научиться эффективно использовать компьютер без понимания теоретических вопросов, изложенных в этой главе. Теория закладывает фундамент, который помогает понять, какие виды задач можно рассчитывать успешно решить. Кроме того, теория вычислений породила немало практических применений. Многие из используемых нами инструментов (хотя бы языки программирования) напрямую базируются на результатах этой теории.

Почему теория вычислений должна интересовать ученого или инженера? Конечно, в наши дни каждый ученый и каждый инженер тоже является программистом, так что предыдущий абзац относится и к ним. Но теория может показаться в высшей степени абстрактной, чем-то удаленным от мира, в котором мы живем. Ученые и инженеры старшего возраста склонны рассматривать компьютер как инструмент — что-то вроде калькулятора и ничего более. Такие представления далеки от истины. После почти вековой борьбы с проблемами в области вычислений исследователи окончательно перешли на качественно новый уровень: в XX веке и ранее наука базировалась на понимании *математических* моделей Вселенной, а в XXI веке мы начинаем все больше зависеть от *вычислительных* моделей. В современном мире уже никто не оспаривает важную роль фундаментальных идей, принесенных теорией вычислений, для научного прогресса.

Почему теория вычислений представляет интерес для гуманитария? Помимо того факта, что многие гуманитарии в наши дни тоже занимаются программированием, есть простой ответ: в мире есть много вещей, которые просто стоит знать, и вся история этой главы (и следующих двух глав) безусловно принадлежит к их числу. С философской точки зрения отношения между человеком и компьютером требуют понимания, а необходимость в таком понимании становится все более и более насущной по мере повсеместного проникновения компьютеров в нашу повседневную жизнь. Понимание теории вычислений как минимум станет хорошей отправной точкой. Как вы увидите, одной из первопричин теории стал поиск ответа на фундаментальный философский вопрос о математике; в наши

дни люди заняты поиском ответов на фундаментальные философские вопросы о вычислениях.

Кроме этих общих причин, также существует особая причина для того, чтобы уделить внимание теории вычислений в этой книге. Она предоставляет историческую перспективу, которая определяет контекст для понимания того, как на самом деле работают компьютеры (это будет нашей целью в главах 6 и 7). Как и во многих научных областях, теория и история захватывающим образом переплетаются, способствуя лучшему пониманию друг друга. И хотя это не историческая книга и мы никак не сможем изложить историю во всех подробностях, каждому читателю, проявляющему хотя бы мимолетный интерес к вычислениям, будет полезно ознакомиться с ключевыми моментами этой истории.

Алан Тьюринг (1912–1954)

Центральная фигура этой главы — Алан Тьюринг, английский математик, работавший в 1930-е годы в Кембриджском и Принстонском университетах. Во время войны он работал в Блетчли-парке, где, по мнению многих, внес важный вклад в завершение Второй мировой войны, разработав метод взлома шифра немецкой «Энигмы». После войны Тьюринг внес ряд эпохальных идей в информатику, включая первые представления о концепции искусственного интеллекта. В начале 1950-х годов его работа была прекращена, когда он был обвинен в гомосексуализме. Вскоре Тьюринга нашли мертвым: он съел яблоко, отравленное цианидом (точные обстоятельства его смерти неизвестны и противоречивы). Только в 2013 году королева Англии подписала посмертное помилование, отменившее обвинительный приговор. Сегодня Тьюринга обычно считают «отцом информатики». За дополнительной информацией об истории жизни Тьюринга обращайтесь к книгам, перечисленным на с. 752.

В этой главе мы сосредоточимся на революционных положениях, которые были высказаны Тьюрингом в статье «О вычислимых числах применительно к проблеме разрешимости», опубликованной в сборнике Proceedings of the London Mathematical Society (1937). Разделы 5.2, 5.3 и 5.4 посвящены результатам, опубликованным в статье, которая считается одной из важнейших научных публикаций XX века.

Чтобы понять работу Тьюринга, необходимо изменить точку зрения. Мы встанем на точку зрения математика, постараемся отбросить все второстепенные подробности и сосредоточиться на понимании базовых проблем, применимых в общей ситуации. В разделе 5.1 закладывается основа, необходимая для понимания статьи Тьюринга.

В разделе 5.5 мы рассмотрим современное состояние исследований, являющихся попыткой решить фундаментальную задачу, которая возникла сразу же после смерти Тьюринга и до сих пор остается нерешенной.

5.1. Формальные языки

Для начала мы определим ряд очень простых абстрактных понятий, на которых будет базироваться последующее рассмотрение важных теоретических вопросов. Интересно, что рассуждения вскоре приведут нас к популярному программному инструменту и сопутствующим механизмам, которые находят собственные применения.

Основные определения

Начнем с абстрактной концепции *символа* (symbol). С математической точки зрения символом может быть все, что угодно, что можно отличить от любого другого символа. На первых порах проще всего рассматривать символ как букву текста или цифру числа. После этого можно переходить к формулировке фундаментальных определений.

Определение. *Алфавит* представляет собой конечное множество символов.

Определение. *Строка* представляет собой конечную последовательность алфавитных символов.

Определение. *Формальный язык* представляет собой множество строк (возможно, бесконечное), принадлежащих одному алфавиту.

Первые два определения могут показаться настолько простыми и очевидными, что даже не требуют отдельного упоминания, но за ними стоят фундаментальные концепции, поэтому они должны иметь четкие и однозначные определения. Третье определение, возможно, покажется вам новым; поразмыслите и постарайтесь понять его суть. Это простое определение, и в дальнейшем мы будем использовать термины «множество строк» и «формальный язык» как синонимы.

Например, вам хорошо знаком такой алфавит, как множество десятичных цифр: *{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}*. Целое число вида 2147483648 является строкой из символов этого алфавита (говорят: «над этим алфавитом»), и мы можем определить формальный язык *положительные числа* как множество строк, состоящих из символов этого алфавита и не начинающихся с 0.

Двоичные строки

Начнем с примеров двоичных строк (формальные языки над двоичным алфавитом), в которых встречаются всего два символа. Конкретный вид символов значения не имеет; можно использовать `{0, 1}`, но в этой главе часто используется запись `{a, b}` для предотвращения путаницы с целыми числами 0 и 1. Простейший способ определения формального языка основан на перечислении его строк. Также наряду с таким точным способом часто пользуются неформальными описаниями.

Например, неформальное описание *двоичные строки длины 3* может использоваться для обозначения формального языка (множества строк) { aaa, aab, aba, abb, baa, bab, bba, bbb }.

	входит в язык	не входит в язык
предпоследним символом является a	aa bbbab bbbbbbbbbababab	a aaaba bbbbbbbbbbbbbb
равное количество символов a и b	ba bbaaba aaaabbbbbbbaaaba	a bbbaa abababababababa
палиндромы	a aba abaabaabaaba	ab bbbba ababababababab
содержит шаблон abba	abba abaababbabbababbba bbbbbbbbbbabbabbbbb	abb bbabaab aaaaaaaaaaaaaaaa
количество символов b кратно 3	bbb baaaaabaaaab bbbabbaaabaaabababaaa	bb abababab aaaaaaaaaaaaaaab

Примеры формальных языков над двоичным алфавитом

Первая сложность заключается в том, что языки могут быть большими или бесконечными множествами, поэтому вариант с перечислением всех строк в языке работает не всегда. Например, когда мы говорим, что язык *палиндромы* является множеством всех палиндромов (двоичных строк, одинаково читающихся в прямом и обратном направлении), или называем язык *равное количество a и b*, вы отлично понимаете, что имеется в виду, хотя оба языка бесконечны и перечислить все их элементы невозможно. После некоторых размышлений становится ясно, что одна из причин вашей уверенности в правильном понимании смысла заключается в том, что вы можете легко решить, входит ли заданная двоичная строка в один из таких языков. Часто мы дополнительно задействуем вашу интуицию и характеризуем языки примерами строк, классифицируемых по признаку принадлежности к языку. Например, когда мы называем язык *количество b кратно 3*, у вас может возникнуть вопрос: возможно ли присутствие *a* в строках? Но потом мы приводим примеры, указывая, что bbb и baaaaabaaaab входят в язык, а bb и abababab не входят, и читатель намного лучше понимает, что имеется в виду (символы a не учитываются). Эти и другие примеры представлены в таблице.

Другие алфавиты

Располагая подходящей интерпретацией определения двоичных строк, мы можем легко определять формальные языки, относящиеся к разным нетривиальным вычислительным задачам. Как вы знаете, двоичная кодировка используется для

всех данных, обрабатываемых компьютерными программами; то же самое можно сделать и с формальными языками. Например, можно определить язык *простые числа* как множество всех двоичных строк, содержащих двоичное представление простого числа, хотя для определения такого языка более естественно использовать десятичное представление. Нет причин ограничиваться двоичным представлением, и при работе с формальными языками можно использовать наиболее подходящий алфавит: стандартный латинский алфавит для работы с текстом, цифры для работы с числами, алфавит { A, T, C, G } для обработки генетических данных и т. д. Если используемый алфавит понятен из контекста, часто мы даже не тратим время на его определение. В следующей таблице приведены примеры часто используемых алфавитов.

<table>
<tr><th></th><th>символы</th><th>название символа</th><th>название строки</th></tr>
<tr><td>двоичный алфавит</td><td>01 (или ab)</td><td>бит</td><td>двоичная строка</td></tr>
<tr><td>латинский алфавит</td><td>abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ</td><td>буква</td><td>слово</td></tr>
<tr><td>десятичные цифры</td><td>0123456789</td><td>цифра</td><td>целое число</td></tr>
<tr><td>специальные символы</td><td>~`!@#$%^&*()_-+={[}]|\:;"'<,>.?/</td><td></td><td></td></tr>
<tr><td>клавиатурные символы</td><td>латинские буквы + десятичные цифры + специальные символы</td><td>клавиша</td><td>печатный текст</td></tr>
<tr><td>генетический код</td><td>ATCG</td><td>основные нуклеотиды</td><td>ДНК</td></tr>
<tr><td>код белка</td><td>ACDEFGHIKLMNPQRSTVWY</td><td>аминокислота</td><td>белок</td></tr>
<tr><td>ASCII</td><td>см. раздел 6.1</td><td>byte</td><td>String</td></tr>
<tr><td>Юникод</td><td>см. раздел 6.1</td><td>char</td><td>String</td></tr>
</table>

Часто используемые алфавиты и сопутствующая терминология

Другие примеры

Другие примеры формальных языков над различными алфавитами приведены в следующей таблице (некоторые из них подробно описываются позднее в этом разделе). Эти примеры наглядно демонстрируют широту применения этой концепции. Некоторые из них основаны на математике. Возьмем множество строк, являющихся десятичными представлениями целых чисел z, для которых существуют положительные целые x, y и $n > 2$, для которых $x^n + y^n = z^n$. В настоящее время этот язык считается пустым, но убедиться в этом будет достаточно сложно, потому что определение представляет собой разновидность знаменитой теоремы Ферма, которая оставалась недоказанной более 300 лет, пока Эндрю Уайлс (Andrew Wiles) не разработал доказательство в 1990-е годы. Другие языки базируются на английском языке, Java или геномике. Можно определить язык *Шекспир* как множество текстов

пьес Шекспира, или язык *Тейлор* как все файлы .wav с песнями Тейлора Свифта, или язык *Звезды* как множество координат звезд Вселенной... словом, что угодно. Все, что нам нужно, — точное определение множества строк.[1]

	входит в язык	**не входит в язык**
палиндромы	madamimadam amanaplanacanalpanama amoraroma	madamimbob madam, i'm adam not a palindrome
нечетные целые числа	3 101 583805233	2 100 2147483648
простые целые числа	3 101 583805233	0003[1] 100 2147483648
целые числа z, для которых выполняется условие $x^2 + y^2 = z^2$ при некоторых целых x, y	5 13 9833	2 16 9999
целые числа z, для которых выполняется условие $x^n + y^n = z^n$ при некоторых целых x, y, n > 2	–	все целые числа
аминокислотные кодировки	AAA ACA ATA AGA CAA CCA CTA CGA TCA TTA GAA GCA GTA GGA AAC ACC	CCC AAAAAAAAA ABCDE
телефонные номера в США	(609) 258-3000 (800) 555-1212	(99) 12-12-12 2147483648
слова английского языка	and middle computability	abc niether misunderestimate
допустимые английские предложения	This is a sentence. I think I can.	xya a b.c.e?? Cogito ergo sum.
допустимые идентификаторы Java	a class $xyz3_XYZ	12 123a a((BC))*
допустимые программы Java	public class Hi { public static void main(String[] args) { } }	int main(void) { return 0; }

Другие примеры формальных языков над различными алфавитами

[1] Ранее было оговорено, что целыми числами считаются строки из десятичных цифр, *не* начинающиеся с 0. В языках программирования незначащие нули обычно допустимы — они просто игнорируются при разборе. — *Примеч. науч. ред.*

Задача описания

Неформальные описания на естественном языке неплохо справляются с задачей в одних случаях (например, палиндромы и простые целые числа), но совершенно не подходят для других (например, английские предложения и программы Java). Почему бы просто не работать с точными, полными определениями, спросите вы? Так в этом и заключается суть проблемы! Наша цель — работать с точными, полными определениями формальных языков. Эта задача называется *задачей описания формальных языков* (или *синтаксиса формальных языков*).

Как полно и точно определить формальные языки? Неформальные описания дают некоторые подсказки, но их понимание зависит от понимания естественного языка, который сам по себе определен не особенно точно. Описания довольно четко проясняют некоторые из рассмотренных примеров, но они удручающе неполны. Например, нужно ли указать, что простые целые числа не должны иметь начальных нулей, или можно считать, что строки вида 000017 тоже входят в язык? Обратите внимание: последнее решение приводит к тому, что каждому простому числу может соответствовать бесконечное количество строк. Такие подробности существенно усложняют определение формального языка. Однако фундаментальные трудности не ограничиваются подробностями, и в этом заключается суть задачи описания. Оказывается, мы можем идентифицировать классы формальных языков, для которых задача описания решается легко. Вскоре мы сделаем это для фундаментального класса языков — так называемых *регулярных языков*.

Задача распознавания

Если способ описания языка известен, появляется другая проблема. Для заданного языка L и строки x требуется ответить на вопрос: входит ли x в L или нет? Эта задача называется *задачей распознавания для формальных языков*. Для решения задачи распознавания понадобится компьютер (как иначе определить, является ли строка из миллиарда бит палиндромом или содержит ли она равные количества a и b?). Более того, вы должны разбираться в математике, синтаксисе естественных языков и множестве других областей знаний. К счастью, мы снова можем выделить классы формальных языков, для которых задача распознавания решаема (причем эти языки достаточно полезны на практике).

Наше изучение формальных языков начнется с важного класса языков — регулярных языков, для которых можно разработать простые решения как задачи описания, так и задачи распознавания. Затем мы покажем, что эти решения играют чрезвычайно важную роль на практике: в естественных языках, в языках программирования (например, Java), в генетических кодах и многих других областях. Затем в разделе 5.2 мы вернемся к фундаментальным теоретическим вопросам. Интересно, что несколько относительно простых расширений механизмов, которые используются для решения задач описания и распознавания для регулярных языков, сильно расширяют множество формальных языков, для которых мы можем решить эти задачи, и приводят нас прямо к фундаментальным принципам компьютерных вычислений. Между формальными языками и вычислениями существует глубокая связь.

Рассматриваемые далее механизмы для описания формальных языков просты, компактны и элегантны. В них используется один из двух базовых подходов. Первый подход — *языковой*: некоторым символам, не входящим в язык, назначается особый смысл, чтобы мы могли записывать строки, определяющие языки. Второй подход — *машинный*: мы определяем класс абстрактных машин, каждая из которых может решать задачи описания для языка.

Регулярные языки

Знакомство с темой начнется с класса формальных языков, называемых *регулярными языками*. Для решения задач описания и распознавания для регулярных языков мы разработаем один языковой и два разных машинных подхода. Затем в конце раздела будут рассмотрены связи между всеми тремя подходами.

Базовые операции

Поскольку формальный язык представляет собой множество строк, для создания эффективного механизма определения формальных языков можно воспользоваться базовыми операциями с множествами. В частности, нам понадобятся операции *объединения*, *конкатенации* и *замыкания*.

Строка входит в *объединение* двух множеств строк в том и только в том случае, если она входит в одно или в оба из этих множеств. Объединение двух формальных языков R и S обозначается R | S. Например,

```
{ a, ba } | { ab, ba, b } = { a, ab, ba, b }
```

Строки, входящие в язык, могут перечисляться в любом порядке, но присутствие дубликатов в объединении недопустимо.

Конкатенацией двух строк называется строка, сформированная присоединением второй строки к первой. Например, в результате конкатенации abb и aab будет получена строка abbaab. В более общей форме конкатенацией *RS* двух формальных языков R и S является множество всех строк, которые могут быть созданы присоединением строки из R к строке из S. Например,

```
{ a, ab } { a, ba, bab } = { aa, aba, abab, abba, abbab }[1]
```

Как и в предыдущем случае, в результате не могут присутствовать дубликаты. (В данном примере строка aba может быть образована присоединением ba к a или присоединением a к ab.)

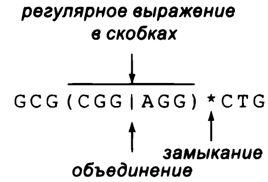
Замыкание языка представляет собой конкатенацию нуля или более строк, входящих в язык. Если R — непустой язык, то запись R* определяет бесконечное множество строк:

```
R* = ∈ | R | RR | RRR | RRRR | RRRRR | RRRRRR ...
```

[1] Как можно заметить, результат соответствует декартову произведению множеств (в данном случае множеством является каждый из языков). — *Примеч. науч. ред.*

Обратите внимание: каждый раз, когда мы берем строку из R, можно взять любую строку из множества. Например, запись (a|b)* задает множество всех двоичных строк. Здесь ∈ обозначает пустую строку (строку, состоящую из 0 символов).

Если регулярное выражение содержит более одного оператора, то в каком порядке операторы должны применяться? Как и в арифметических выражениях, для разрешения таких неоднозначностей применяются круглые скобки или определенный приоритет операторов. В регулярных выражениях замыкание выполняется перед конкатенацией, а конкатенация — перед объединением. Например, регулярное выражение b|ab задает множество {b, ab}, а не множество {bb, ab}. Как и в арифметических выражениях, правила приоритета можно переопределить при помощи круглых скобок. Например, множество {bb, ab} можно представить регулярным выражением (b|a)b. Аналогичным образом регулярное выражение a|b* задает то же множество, что и a|(b*), но не (a|b)*.



Строение регулярного выражения

Регулярные выражения

Регулярное выражение (RE, Regular Expression) представляет собой строку символов, которая задает формальный язык. Для определения того, какие существуют регулярные выражения (и что они означают), используется рекурсия с применением операций объединения, конкатенации и замыкания к множествам, а также круглых скобок для определения приоритета операторов. Более конкретно, каждое регулярное выражение либо представляет собой символ алфавита, который задает одноэлементное множество, содержащее этот символ, либо образуется из следующих операций (где R и S — регулярные выражения).

- **Объединение**: R | S, объединение множеств R и S.
- **Конкатенация**: RS, конкатенация множеств R и S.
- **Замыкание**: R*, замыкание множества R.
- **Круглые скобки**: (R), то же множество, что и R.

Это определение неявно подразумевает, что символы |, *, (и) в алфавит языка не входят. Эти символы называются метасимволами; позднее мы расскажем, что делать с языками, которые содержат эти символы.

Регулярные языки

Рекурсивное определение не только позволяет строить регулярные выражения произвольной сложности, но и точно определяет их смысл. Каждое регулярное выражение полностью определяет некоторый формальный язык, но не каждый формальный язык может быть задан некоторым регулярным выражением. Позднее

будет приведено формальное доказательство с представлением языка, который не может быть задан никаким регулярным выражением. Однако класс языков, которые могут быть определены некоторым регулярным выражением, настолько важен, что заслуживает имени.

Определение. Язык называется *регулярным* в том и только в том случае, если он может быть определен регулярным выражением.

В таблице на следующей странице приведены примеры регулярных языков с регулярными выражениями, которые их описывают, и примерами строк — как входящих в язык, так и не входящих в него. Как можно быть уверенным в том, что каждое регулярное выражение задает тот же язык, что и неформальное описание? В общем случае этот факт должен доказываться для каждого регулярного выражения, а разработка такого доказательства часто сопряжена с трудностями. Важность регулярных выражений заключается в том, что они позволяют отойти от неформальных описаний: они предоставляют механизм определения языков, естественный для важной категории применений и одновременно формальный. Независимо от нашей уверенности в точности неформального описания, каждое регулярное выражение является точным и полным описанием некоторого регулярного языка. В самом деле, вы можете просто записать регулярное выражение вида `(ab*|aba)*(ab*a|b(a|b))*`, даже когда не представляете, какой язык оно задает.

Чтобы получше освоиться с регулярными выражениями, не пожалейте времени и убедитесь в том, что каждое регулярное выражение в таблице задает именно этот язык. Для этого сначала убедитесь в том, что строки, упомянутые как входящие в язык, подходят под неформальное описание и соответствуют регулярному выражению, а потом — что строки, не входящие в язык, не подходят под неформальное описание и не соответствуют регулярному выражению. Ваша цель — понять, почему регулярное выражение задает *все строки языка и никакие другие строки*. Более подробная информация о некоторых языках приводится ниже.

Решая кроссворд, вы можете столкнуться с вопросом вида: «Что это за слово из 8 букв, у которого две средние буквы hh?» Когда вы освоите механику использования регулярных выражений, описанную в этом разделе, кроссворды и словесные игры предстанут перед вами в совершенно новом свете.

В геномике регулярные выражения над алфавитом `{A,T,C,G}` используются для описания свойств генов. Например, в человеческом геноме присутствует участок, который может быть описан регулярным выражением `GCG(CGG|AGG)*CTG`, причем количество повторений `CGG/AGG` сильно различается для разных людей. Для некоего генетического заболевания, приводящего к умственной отсталости и другим симптомам, характерно высокое количество повторений. Регулярные выражения широко применяются на практике для получения ответов на важные научные вопросы такого рода.

<table>
<tr><th>регулярный язык</th><th>регулярное выражение</th><th>входит в язык</th><th>не входит в язык</th></tr>
<tr><td>двоичный алфавит</td><td></td><td></td><td></td></tr>
<tr><td>пятым символом с конца является a</td><td>(a|b)*a(a|b)(a|b)(a|b)(a|b)</td><td>aaaaa
bbbabbbb
bbbbbbababababa</td><td>a
bbbbbbbba
aaaaaaaaaaabaaaa</td></tr>
<tr><td>содержит подстроку abba</td><td>(a|b)*abba(a|b)*</td><td>abba
aababbabbababbba
bbbbbbbbabbabbbbb</td><td>abb
bbabaab
aaaaaaaaaaaaaaaa</td></tr>
<tr><td>не содержит строку bbb</td><td>(bba|ba|a*)*(a*|b|bb)</td><td>aa
ababababbaba
aaaaaaaaaaaaab</td><td>bbb
ababbbbabab
bbbbbbbbbbbbb</td></tr>
<tr><td>количество символов b кратно 3</td><td>a*|(a*ba*ba*ba*)*</td><td>bbb
aaa
bbbaababbaa</td><td>b
baaaaaaab
baabbbaaaaab</td></tr>
<tr><td>десятичные цифры</td><td></td><td></td><td></td></tr>
<tr><td>положительное целое число, кратное 5</td><td>5|(1|2|…|9)(0|1|…|9)*(0|5)</td><td>5
200
9836786785</td><td>1
0005
3452345234</td></tr>
<tr><td>положительное троичное число</td><td>(1|2)(0|1|2)*</td><td>11
2210221</td><td>011
19
9836786785</td></tr>
<tr><td>буквы нижнего регистра</td><td></td><td></td><td></td></tr>
<tr><td>содержит триграф spb</td><td>(a|b|c|…|z)*spb(a|b|c|…|z)*</td><td>raspberry
crispbread</td><td>subspace
subspecies</td></tr>
<tr><td>использует только верхний ряд клавиатуры</td><td>(q|w|e|r|t|y|u|i|o|p)*</td><td>typewriter
reporter</td><td>alfalfa
paratrooper</td></tr>
<tr><td>генетический код</td><td></td><td></td><td></td></tr>
<tr><td>синдром ломкой X-хромосомы</td><td>GCG(CGG|AGG)*CTG</td><td>GCGCTG
GCGCGGCTG
GCGCGGAGGCTG</td><td>GCGCGG
CGGCGGCGGCTG
GCGCAGGCTG</td></tr>
</table>

Примеры регулярных выражений над различными алфавитами

При обработке информации всегда важно иметь полное и точное описание этой информации. Например, когда вы вводите свое имя, адрес и прочую информацию на веб-форме, программа, которая обрабатывает эту информацию, должна прежде всего проверить, что введенный текст имеет смысл. Если вы ввели буквы там, где должны быть цифры, или знак доллара там, где должен быть телефонный номер, программа сообщает об ошибке и предлагает вам ее исправить. Вскоре вы увидите, как использовать регулярные выражения для описания знакомых низкоуровневых

абстракций: дат, номеров кредитных карт и идентификаторов Java для извлечения информации из баз данных (например, результатов научных экспериментов).

Регулярные выражения очень часто применяются при обработке информации. Как вы вскоре увидите, сегодня они представлены в самых разных областях — даже в Java существует ряд функций, основанных на регулярных выражениях. Они также представляют первый шаг процесса компиляции программ, написанных на языках высокого уровня (таких, как Java), на машинный язык. Но что еще важнее, как мы неоднократно подчеркивали, регулярные выражения становятся первым шагом на пути поиска ответов на фундаментальные вопросы, относящиеся к вычислениям.

В общем случае любой язык, который может быть задан регулярным выражением, является регулярным — это следует из определения. Но как насчет языков, которые выражаются другим способом? Является ли множество всех палиндромов регулярным языком? Среди прочего, в этом разделе мы попробуем понять, как же классифицировать языки. И хотя существует немало интересных и полезных регулярных языков, также есть много интересных и полезных языков, которые регулярными не являются. Позднее будет рассмотрена концепция более мощной системы спецификаций, чем регулярные выражения, применимой для нерегулярных языков.

Задача распознавания для регулярных выражений

Как упоминалось ранее, другая фундаментальная задача заключается в следующем: как для заданной двоичной строки определить, входит ли она в язык, определяемый заданным регулярным выражением? Например, входит ли строка `abaaabbbbbbabaabbba` в язык, определяемый выражением `(ab*|bab)*(bb*b|(b(a|b)))*`? Это пример задачи распознавания для регулярных языков. Регулярные выражения важны именно тем, что они позволяют точно определить эту задачу для большого класса языков (регулярных языков), но они не решают ее. Разумеется, мы должны сначала проанализировать задачу для регулярных языков, прежде чем браться за более сложные классы языков, такие как множество всех простых чисел или множество всех программ Java.

Для разработки решения задачи распознавания для регулярных языков позднее в этом разделе будут использоваться абстрактные машины. Вообще говоря, версия этого решения реализована в методе `matches()` из библиотеки Java `String`, поэтому сначала будет описано, как использовать этот метод. Если `s` — произвольный объект Java `String`, а `re` — произвольное регулярное выражение, то `s.matches(re)` возвращает `true`, если `s` входит в язык, определяемый `re`, или `false` в противном случае[1]. В листинге 5.1.1 этот метод используется для решения задачи распознавания: программа получает регулярное выражение в аргументе командной строки и для каждой строки в стандартном потоке ввода выводит `Yes`, если она входит

[1] В большинстве повседневных задач прикладного программирования это проверка строки на *соответствие* регулярному выражению; таким образом, получается своего рода «расширенное сравнение» строк. — *Примеч. науч. ред.*

Листинг 5.1.1. Проверка правильности (распознавание)

```
public class Validate
{
   public static void main(String[] args)
   {
      String re = args[0];
      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         if (s.matches(re)) StdOut.println("[Yes]");
         else               StdOut.println("[No]");
      }
   }
}
```

Программа демонстрирует базовый механизм использования регулярных выражений в Java: метод matches() из библиотеки Java String решает задачу распознавания для регулярных выражений. Метод main() получает регулярное выражение в аргументе командной строки, после чего для каждой строки из стандартного потока ввода выводит Yes, если строка входит в язык, определяемый регулярным выражением, или No в противном случае.

```
% java Validate "(a|b)*a(a|b)(a|b)(a|b)(a|b)"
bbbabbbb
[Yes]
bbbbbbbba
[Yes]

% java Validate "a*|(a*ba*ba*ba*)*"
bbbaababbaa
[Yes]
baabbbaaaaab
[No]

% java Validate "GCG(CGG|AGG)*CTG"
GCGCGGAGGCTG
[Yes]
CGGCGGCGGCTG
[No]
```

в язык, определяемый регулярным выражением, или No, если строка в этот язык не входит. Возможность простого выполнения таких проверок настолько полезна, что мы рассмотрим некоторые расширения и обобщения, прежде чем возвращаться к теории, — они помогут вам освоиться с регулярными выражениями и в то же время познакомят вас с незаменимыми инструментами программирования из стандартных библиотек Java.

Кроме задачи распознавания, полнота и точность описания, обеспечиваемые регулярными выражениями, открывают путь к ряду других естественных и четко определенных задач. Например, как для двух регулярных выражений проверить, определяют ли они один и тот же язык? Легко проверить, что a(b|ab|aab) и (a|aa)

(b|ab) определяют один язык, но определяет ли ((abb|baaa)*(ba*|b*a))* тот же язык, что и (ab*|bab)*(bb*b|a(a|b)))*? Эта задача называется *задачей эквивалентности* для регулярных языков. Как бы вы подошли к ее решению для пары регулярных выражений произвольной сложности, длина которых составляет (допустим) тысячи символов? Аналогичным образом, запрашивая из словаря 8-буквенные слова, в середине которых строят буквы hh, на самом деле вы запрашиваете *пересечение* двух множеств: строки, которые присутствуют в обоих. Регулярные выражения позволяют точно определять подобные задачи, но решение таких задач — совсем другое дело. Если подобные задачи кажутся вам интересными (как и многим людям), то, скорее всего, дальнейшее изучение теории вычислений вам понравится — ведь это только верхушка айсберга.

Обобщенные регулярные выражения

Приведенное определение регулярных выражений является минимальным. В него включены четыре базовые операции, характеризующие регулярные языки (конкатенация, объединение, замыкание и группировка). На практике это множество бывает полезно дополнить различными другими операциями. Ниже кратко описаны расширения регулярных выражений в языке Java, которые делятся на три основные категории:

- расширение алфавита;
- сокращенная запись для операции объединения;
- расширения операции замыкания.

Для краткости мы будем называть обобщенные регулярные выражения Java просто «регулярными выражениями Java» или «обобщенными регулярными выражениями» без уточнения различий. Аналогичные механизмы широко применяются в других языках и других областях, но единственно точного определения того, что считать «обобщенным», не существует.

Единственная характеристика, которая должна присутствовать во всех обобщениях, — то, что каждое обобщенное регулярное выражение должно описывать регулярный язык. Иначе говоря, любое обобщенное регулярное выражение может быть (теоретически) преобразовано в (более громоздкое) стандартное регулярное выражение вроде тех, которые рассматривались выше. Ограничение выглядит немного парадоксально; по сути, оно означает, что мы обобщаем не регулярные языки, а только язык, который используется для описания регулярных выражений. В свое время мы рассмотрим настоящие обобщения, а пока будьте бдительны, потому что многие системы (включая Java) поддерживают расширения регулярных выражений, которые не соответствуют этому ограничению. Мы вернемся к этой теме позднее[1].

[1] Версии регулярных выражений, поддерживаемые и используемые различными реальными приложениями, принято называть *диалектами*. — *Примеч. науч. ред.*

Алфавит

Самое очевидное обобщение — расширение алфавита. Символами регулярных выражений Java являются знаки Юникода без каких-либо ограничений. Но при подобном расширении алфавита до полного набора символов возникает проблема — нам понадобятся *механизмы экранирования*, чтобы метасимволы |, *, (и) могли использоваться и в регулярных выражениях, и в алфавите языка. А конкретнее, `\|` соответствует |, `*` соответствует *, `\(` соответствует), `\)` соответствует), а `\\` соответствует \. Все остальные метасимволы, используемые в описанных выше расширениях, также должны использовать экранирование \ при использовании в качестве символов языка.

Сокращенная запись

Наличие большого количества символов в алфавите немедленно приводит к необходимости сокращенной записи для операции объединения, чтобы несколькими нажатиями клавиш можно было определять целые группы символов. Например, универсальный символ (`.`) представляет любой символ алфавита; это сокращенная запись для длинной цепочки операций объединения всех символов алфавита. Например, при работе с десятичными цифрами вы наверняка предпочтете ввести одну точку, чем набирать `0|1|2|3|4|5|6|7|8|9`[1].

Обобщенные регулярные выражения поддерживают ряд аналогичных сокращений, в том числе:

- метасимвол ^ представляет начало текстовой строки, а `$` — конец текстовой строки (так называемые *якоря*);
- список или диапазон символов в квадратных скобках `[]` представляет любой символ из этого списка или диапазона;
- если первым символом в квадратных скобках является символ ^, то конструкция обозначает символы Юникода, *не входящие* в этот список или диапазон;
- некоторые последовательности, состоящие из обратного слеша и символа алфавита, представляют определенное множество символов. Например, `\s` представляет любой символ из категории «пробельных» (обычно это символы пробела и табуляции).

Например, выражение `^text$` представляет слово `text`, стоящее в отдельной строке, `[a-z]` представляет буквы нижнего регистра, `[0-9]` представляет десятичные цифры, `[^aeiou]` представляет символы, не являющиеся гласными буквами нижнего регистра, а `[A-Z][a-z]*` представляет слова, записанные с прописной буквы.

[1] Регулярному выражению «`.`» соответствует не только любая десятичная цифра, но и вообще *любой* символ. Это несущественно для такого простейшего примера, но в реальных задачах, конечно, необходимо обеспечивать точность и однозначность соответствий. — *Примеч. науч. ред.*

Замыкание

Операция замыкания часто оказывается слишком общей для прямого применения на практике. По этой причине в регулярных выражениях Java предусмотрены следующие варианты ограничения количества повторений (так называемые *квантификаторы*):

- один и более (+)
- нуль или один, то есть *необязательный* символ (?)
- ровно n ({n})
- между m и n ({m, n})

Например, `[^aeiou]{6}` определяет шестибуквенные слова, не содержащие гласных в нижнем регистре, — например, `rhythm` или `syzygy`.

Каждая из этих записей является сокращенной формой стандартной записи регулярных выражений, которая может быть очень длинной. Например, обобщенное регулярное выражение вида `[^aeiou]` является сокращением для длинной последовательности операций объединения для всех остальных символов, а `[^aeiou]{6}` — сокращением для шести экземпляров этой строки. Теоретически можно представить, что Java во внутренней реализации сначала преобразует обобщенные регулярные выражения в длинную стандартную форму; на практике каждое расширение позволяет проектировщикам алгоритмов блеснуть своей изобретательностью.

Строение обобщенного регулярного выражения

Далее мы будем приводить многочисленные примеры регулярных выражений Java; некоторые примеры также приведены на с. 705. Как и прежде, вы должны изучить каждый пример, разобраться в том, почему одни примеры входят в заданный язык, а другие строки в него не входят, и попытаться понять, как регулярные выражения выполняют свою работу.

Применения

Как использовать регулярные выражения на практике? Они встречаются в разнообразных ситуациях, и сейчас мы рассмотрим некоторые примеры. На фундаментальном уровне все сценарии направлены на достижение экономии: они используют относительно короткие регулярные выражения для определения относительно больших (и даже бесконечных) множеств строк. Используя регу-

лярные выражения, человек или программа могут работать с множеством строк как с единой сущностью.

Проверка корректности (валидация) данных

С обобщенными регулярными выражениями программа 5.1.1 более эффективна (и приносит много больше пользы). Возможно, вы не подозревали, что часто сталкивались с задачей распознавания во время работы в Интернете. Когда вы вводите дату или номер карты на коммерческом веб-сайте, программа обработки ввода должна убедиться в том, что ваш ответ имеет правильный формат. Один из способов выполнения такой проверки заключается в написании кода, проверяющего все возможные случаи: если в поле должна вводиться сумма в долларах, код сначала проверяет, что первым символом является **$**, что за **$** следует набор цифр и т. д. В другом, более эффективном решении определяется регулярное выражение, описывающее множество всех допустимых вариантов введенных данных. В этом случае проверка входных данных на правильность в точности эквивалентна задаче распознавания: входит ли введенная строка в язык, описываемый регулярным выражением? Появилось немало библиотек регулярных выражений для стандартных проверок данных, поскольку такие проверки получили широкое распространение. Как правило, регулярное выражение является намного более точным и компактным выражением множества всех допустимых строк, чем специальная программа для проверки всех возможных случаев. Несколько примеров на следующей странице демонстрируют это обстоятельство.

```
% java Validate "\\\$[1-9][0-9]*\\.[0-9][0-9]"
$22.99
[Yes]
$1,000,000.00
[No]

% java Validate "ATG(...)+(TAG|TAA|TGA)"
ATGCGCCTGCGTCTGTACTAG
[Yes]
ATGATTGTAG
[No]
```

Вычислительная биология

Как вы уже знаете, исследователи разработали ряд кодировок, упрощающих обработку и анализ генетических данных. Генетический код (над алфавитом `ATCG`), упоминавшийся ранее, является самым простым; другой стандартный однобуквенный код использует алфавит из 20 символов `ACDEFGHIKLMNPQRSTVWY` для аминокислот. Научные подробности в данном контексте несущественны, но мы приведем конкретный пример использования этого кода; сигнатура семейства «цинкового пальца» типа C_2H_2 определяется следующим образом:

- `C`, затем две, три или четыре аминокислоты, затем
- `C`, затем три аминокислоты, затем

	регулярное выражение	в языке
Юникод		

идентификаторы Java (частично)	`[$_a-zA-Z][$_a-zA-Z0-9]*`	`i`
		`_$_$`
		`System`

адрес электронной почты (частично)	`[a-zA-Z]+@([a-zA-Z]+\.)+...`	`XYZ@yahoo.com`
		`rs@princeton.edu`
		`xx@whitehouse.gov`

десятичные цифры и разделители		
номера социального страхования в США	`[0-9]{3}-[0-9]{2}-[0-9]{4}`	`330-12-3456` `213-44-5689`
телефонные номера в США	`\([0-9]{3}\) [0-9]{3}-[0-9]{4}`	`(800) 555-1212` `(609) 258-4345`

денежные суммы в долларах (частично)	`$[1-9][0-9]*\.[0-9][0-9]`	`$22.99`
		`$1000000.00`
		`$0.01`

генетический код		

потенциальные гены	`ATG(...)+(TAG|TAA|TGA)`	`ATGATGATGATGTGA`
		`ATGAAATAG`
		`ATGCGCCTGCGTCTGTACTAG`

аминокислоты		
сигнатура семейства так называемого цинкового пальца типа C_2H_2	`C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H`	`CCCCCCCCCCCCCCCCCHHHHH` `CAASCGGPYACGGWAGYHAGWH`

Примеры обобщенных регулярных выражений над разными алфавитами

- `L`, `I`, `V`, `M`, `F`, `Y`, `W`, `C` или `X`, затем восемь аминокислот, затем
- `H`, затем три, четыре или пять аминокислот, затем
- `H`.

Конечно, обобщенное регулярное выражение

```
C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H
```

выражает это определение в намного более компактной форме. Например, строка `CAASCGGPYACGGWAGYHAGWH` является сигнатурой семейства «цинкового пальца» типа C_2H_2, потому что в нее входят:

- `C`, затем `AAS` (от двух до четырех аминокислот), затем
- `C`, затем `GGP` (три аминокислоты), затем
- `Y` (аминокислота из множества `{L, I, V, M, F, Y, W, C, X}`), затем

- ACGGWAGY (восемь аминокислот), затем
- H, затем AGW (от трех до пяти аминокислот), затем
- H.

С этим обобщенным регулярным выражением можно просто воспользоваться программой 5.1.1 для проверки того, является ли любая заданная строка сигнатурой семейства «цинкового пальца» типа C_2H_2:

```
% java Validate "C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H"
CAASCGGPYACGGWAGYHAGWH
[Yes]
CAASCGGPYACGYGWAGYHAGWH
[No]
```

Поиск

Похоже, в современных компьютерных системах пользователь вечно занят поиском. Мы ищем данные во Всемирной паутине, ищем файлы и используем строковый поиск как встроенную функцию в самых разных приложениях. Обычно пользователь вводит строку и ищет совпадение, а с регулярными выражениями поиск становится как более гибким, так и более точным. Важнейший программный инструмент для поиска — grep — был разработан Кеном Томпсоном для системы Unix в 1970-х годах; эта программа до сих пор присутствует в качестве команды (утилиты) во многих компьютерных системах. В листинге 5.1.2 приведена реализация grep, использующая встроенные средства Java. Это фильтр, который получает регулярное выражение, заданное как аргумент командной строки, и направляет в стандартный поток вывода строки из стандартного потока ввода, в которых встречается любая подстрока из языка, описываемого этим регулярным выражением.

С помощью нашей программы Grep (или встроенной версии grep) можно быстро выполнять различные задачи поиска, в числе которых:

- поиск всех строк программы, содержащих заданный идентификатор;
- поиск всех слов из словаря, содержащих заданный шаблон;
- поиск части генома, содержащей заданные шаблоны;
- поиск фильмов, в которых снимался заданный актер;
- извлечение из файла данных строк, обладающих заданными характеристиками,
- и многие, многие другие.

В упражнениях этого раздела встречаются примеры поиска в самых разных областях; некоторые из них наверняка покажутся вам интересными. Опытные программисты, привыкшие к ним, считают grep бесценным инструментом. Функция поиска сейчас встраивается во многие высокоуровневые приложения, хотя пользователи часто упускают из виду возможность полноценного поиска по регулярному выражению.

Листинг 5.1.2. Поиск по шаблону с использованием обобщенных регулярных выражений

```
public class Grep
{
   public static void main(String[] args)
   {
      String re = args[0];
      while (StdIn.hasNextLine())
      {
         String line = StdIn.readLine();
         if (line.matches(".*" + re + ".*"))
            System.out.println(line);
      }
   }
}
```

Программа реализует функциональность классической утилиты grep: она получает регулярное выражение как аргумент командной строки и выводит в стандартный вывод все строки из стандартного потока ввода, содержащие подстроку языка, описываемого регулярным выражением. Для простоты эта реализация использует метод matches() из библиотеки Java String; классы Java Pattern и Matcher повышают эффективность реализации для сложных регулярных выражений или больших файлов (см. листинг 7.2.3). Файл words.txt (доступен на сайте книги) содержит все слова из словаря, по одному в строке. За описанием файла movies.txt обращайтесь к разделу 4.5.

```
% java Grep class < Grep.java
public class Grep

% java Grep "^[tuv][def][wxy].$" < words.txt
text

% java Grep "^...hh...$" < words.txt
withheld
withhold

% java Grep "Bacon, Kevin" < movies.txt | java Grep "Sedgwick, Kyra"
Murder in the First (1995)/ … /Bacon, Kevin/ ... /Sedgwick, Kyra
Woodsman, The (2004)/ … /Bacon, Kevin/ ... /Sedgwick, Kyra
```

Проверка данных и поиск — фундаментальные операции, которые с обобщенными регулярными выражениями выполнять проще, чем без них. Они приводятся исключительно в учебных целях; понимание регулярных выражений и умение ими пользоваться необходимо для каждого, кто стремится к более эффективной организации работы с компьютером. В наши дни мы буквально утопаем в научных, инженерных и коммерческих данных, и навык поиска по шаблону может сыграть важную роль для прогресса в исследованиях. Как видно из реализаций в листингах 5.1.1 и 5.1.2, средства Java позволяют легко включить поддержку поиска по регулярному выражению в приложение (впрочем, это относится и к другим современным средам

программирования), поэтому поддержка регулярных выражений будет встречаться в приложениях все чаще.

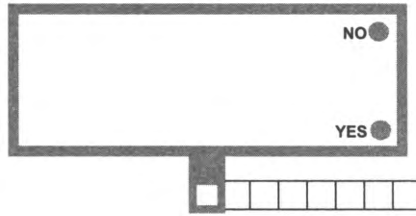
Что еще важнее, регулярные выражения демонстрируют фундаментальную парадигму. Идея использования формального механизма для определения языка с последующей проверкой принадлежности строки к заданному языку принадлежит к числу базовых концепций. Чтобы понять ее в полной мере, необходимо понять, как на самом деле работает процесс распознавания.

Для эффективного использования регулярных выражений необходимо решить задачу распознавания. Подумайте, как бы вы реализовали программу 5.1.1 без метода Java `matches()`, даже для базовых регулярных выражений над двоичным алфавитом? Как написать программу, которая получает регулярное выражение и строку и решает, принадлежит ли строка языку, определяемому регулярным выражением? Поиски ответа на этот вопрос выводят нас на путь, который ведет к фундаментальным концепциям, занимающим центральное место в информатике.

Абстрактные машины

Рассмотрим простую вычислительную модель для решения таких задач, как задача распознавания для регулярных выражений. На первый взгляд эта модель не имеет никакого отношения к регулярным выражениям, но будьте уверены: это первый шаг на пути к решению этих задач.

Абстрактная машина представляет собой вычислительную модель для решения задачи распознавания для формального языка. На самом общем уровне абстрактная машина представляет собой не что иное, как математическую функцию, связывающую входную строку с одним выходным битом. Обычно мы рассматриваем более подробную модель, в которой входные символы анализируются по отдельности. А именно, для любого заданного формального языка представляется устройство, взаимодействующее с внешним миром при помощи трех компонентов: переключателя «вкл/выкл», устройства ввода/вывода, способного читать (а возможно, и записывать) один символ с ленты, и по крайней мере двух световых индикаторов («да» и «нет».) Чтобы загрузить машину, мы помещаем нужную строку на ленту и включаем ее. Машина читает данные с ленты, и если строка входит в формальный язык машины, то загорается индикатор «да»; если строка не входит в формальный язык, то загорается индикатор «нет». Мы говорим, что машина *принимает* строку (находит совпадение), если загорается индикатор «да», и машина *отклоняет* строку, если загорается индикатор «нет». Таким образом, абстрактная машина задает формальный язык (множество всех принимаемых строк) и представляет абстрактное решение задачи распознавания для этого языка (поместить строку на ленту, включить машину и посмотреть, загорелся ли индикатор «да»); иначе говоря, машина *распознает* язык. Мы рассмотрим несколько абстрактных машин, отличающихся своими возможностями по работе с лентой и доступностью других устройств ввода/вывода и световых индикаторов.



Абстрактная машина

Сейчас нас интересует применение абстрактных машин в теории вычислений, но заметим, что эта направленность оправданна, потому что абстрактные машины подходят для моделирования практических вычислений любого типа. Например, когда вы работаете с банкоматом, последовательность нажатий кнопок соответствует входной строке, индикатор приема соответствует выдаче денег, а индикатор отклонения — уведомлению о нехватке средств. Также входная строка может быть десятичным целым числом; машина включает индикатор «да» в том и только в том случае, если целое число является простым (а может быть, она выводит на ленту наибольший множитель числа). Еще одним примером служит абстрактное представление программы Java, которое рассматривалось в начале книги. Идея настолько универсальна, что она распространяется практически на любые вычисления, которые вам только могут встретиться. Другие (более конкретные) примеры будут рассмотрены в книге.

Как абстрактная машина решает, какой из индикаторов следует включить: «да» или «нет»? Предполагается, что она читает входные данные (полностью или частично) и выполняет некоторые вычисления. Какие вычисления? Любая абстрактная машина характеризуется базовыми операциями, которые мы определяем для нее. Наша цель — отбросить все второстепенные подробности и сосредоточиться на машинах, использующих как можно более простые базовые операции.

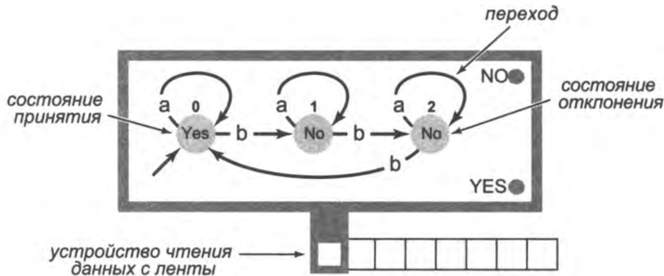
Абстрактные машины являются специализированными компьютерами. В этом разделе мы сосредоточимся на модели, в которой машина решает всего одну задачу: задачу распознавания для некоторого регулярного языка. В свое время вы увидите, как эта концепция помогает понять возможности знакомых компьютеров общего назначения по решению задач. Применение как можно более простой машины дает возможность формально доказывать факты, относящиеся к ее поведению. Что еще важнее (и еще удивительнее), мы сможем получать ответы на вопросы, имеющие очень широкий спектр применения.

Детерминированные конечные автоматы

Конечно, любая машина должна быть способна как минимум прочитать входной символ, поэтому мы начнем с изучения абстрактной машины, которая не делает почти ничего другого. Такие машины называются *детерминированными конечными автоматами* (ДКА). Каждый ДКА состоит из следующих компонентов.

- Конечное множество состояний, каждое из которых обозначается либо как состояние *принятия*, либо как состояние *отклонения*.
- Множество переходов, которое определяет, как изменяется состояние машины. Каждое состояние имеет один переход для каждого символа в алфавите.
- *Устройство чтения* данных с ленты, в исходном состоянии установленное на первый символ входной строки и способное только прочитать символ и переместиться к следующему символу.

Каждый ДКА представляется в виде графа, в котором состояния принятия представлены вершинами с меткой «да», состояния отклонения — вершинами с меткой «нет», а каждый переход — направленным ребром (такие ребра называют также *дугами*), помеченным символом из алфавита. В целях идентификации вершины нумеруются целыми числами, начиная с 0. Ниже изображен пример ДКА над двоичным алфавитом.



Детерминированный конечный автомат

Для этого ДКА используем символы алфавита a и b, чтобы избежать путаницы с индексами вершин. Если один и тот же переход должен происходить для нескольких символов, для экономии ребро просто помечается строкой из всех символов, по которым этот переход должен инициироваться.

Режим работы

Все ДКА начинают работу в состоянии 0: входная строка нанесена на ленту, а устройство чтения установлено на крайний левый символ входной строки. В процессе работы состояние машины дискретно изменяется по следующему правилу: машина читает символ, сдвигает устройство чтения вправо на одну позицию, а затем изменяет состояние в соответствии с переходом, помеченным только что прочитанным символом. Когда входные данные будут исчерпаны, ДКА прекращает работу. Если в этот момент он находится в состоянии принятия, загорается индикатор «да»; если он находится в состоянии отклонения, загорается индикатор «нет».

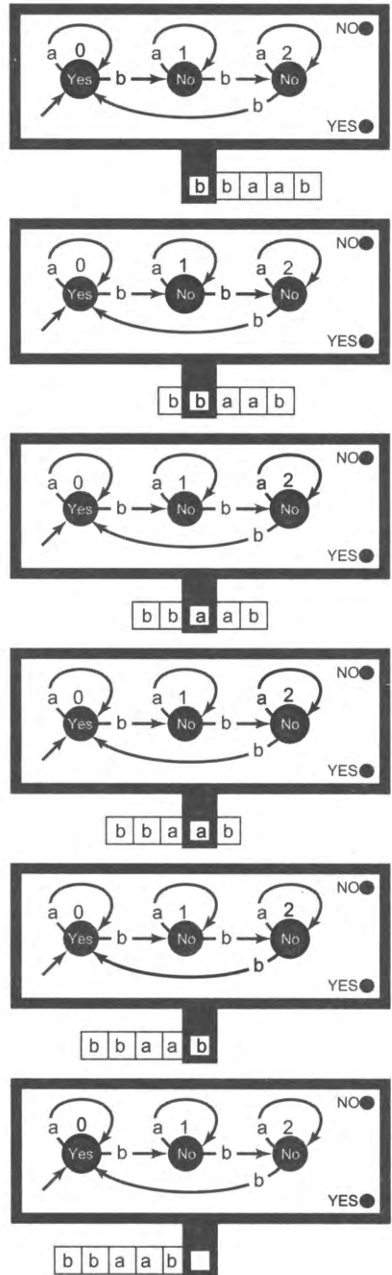
Например, ДКА из нашего примера с входными данными bbaab ведет себя следующим образом (см. рисунок): начиная с состояния 0 он читает первый символ b и переходит в состояние 1. Затем он читает второй символ и переходит в состо-

яние 2, потому что этим символом является b. Далее ДКА остается в состоянии 2, потому что читает третий и четвертый символ a, после чего переходит в состояние 0, потому что пятым будет прочитан символ b. На момент, когда входные данные будут исчерпаны, машина находится в состоянии 0. Это состояние имеет пометку «да», поэтому активизируется индикатор «да». Иначе говоря, машина получает двоичную строку bbaab. Из трассировки также видно, что машина отклоняет двоичные строки b, bb, bba и bbaa.

Гораздо более компактная запись трассировки — последовательность переходов состояний, выполняемых ДКА для принятия решения о том, принять или отклонить эту двоичную строку; завершающее состояние определяет, что нужно сделать с двоичной строкой: принять или отклонить. В нашем примере последовательность `0->1->2->2->2->0` представляет трассировку работы данного ДКА для входной строки bbaab. Обратите внимание: входная строка может быть определена по последовательности переходов[1].

Описание языка

Какие двоичные строки языка распознаются заданным ДКА? Чтобы ответить на этот вопрос, необходимо изучить автомат и сделать выводы относительно тех строк, которые он принимает или отклоняет. В нашем примере сразу видно, что символы a не влияют на результат, поэтому беспокоиться нужно только о b. Машина отклоняет b и bb, но принимает bbb; она отклоняет bbbb и bbbbb, но принимает bbbbbb (и игнорирует все a); разумно предположить, что ДКА распознает строки языка, в которых количество b кратно 3. Нетрудно убедиться в этом, проведя полное доказательство методом индукции по количеству прочитанных битов, что машина находится в состоянии 0 в том и только в том



Трассировка работы ДКА

[1] Благодаря тому, что в *этом* ДКА любой из переходов между состояниями инициируется одним и только одним конкретным символом. — *Примеч. науч. ред.*

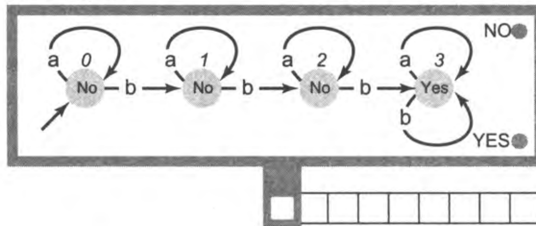
случае, если количество прочитанных b кратно 3, в состоянии 1 — в том и только в том случае, если количество прочитанных b кратно 3 плюс 1, в состоянии 2 — в том и только в том случае, если количество прочитанных b кратно 3 плюс 2 (см. упражнение 5.1.3).

Другие примеры

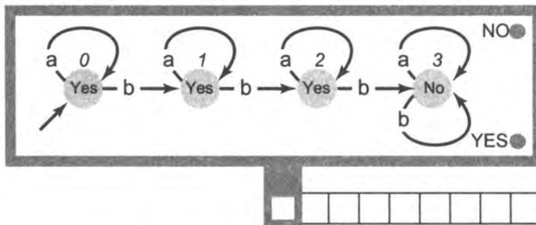
На иллюстрации ниже представлены еще три примера ДКА над двоичным алфавитом, на следующей странице — ДКА над алфавитом ДНК, распознающий потенциальные гены, и еще несколько примеров ДКА описано в упражнениях. В каждом примере убедитесь в том, что автомат распознает заявленный язык: сначала проведите трассировку его работы и убедитесь, что ДКА принимает строки, которые должны входить в язык, и отклоняет строки, которые входить в язык не должны. Затем попробуйте понять, почему это происходит со всеми такими строками.

Эти примеры должны убедить вас в том, что ДКА определяются просто и естественно, даже если их поведение далеко не так просто для понимания. Благодаря простоте определения ДКА широко применяются для решения нетривиальных

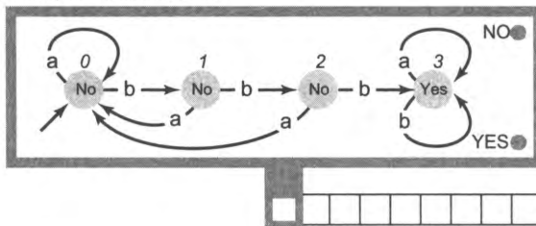
не менее трех b



менее трех b

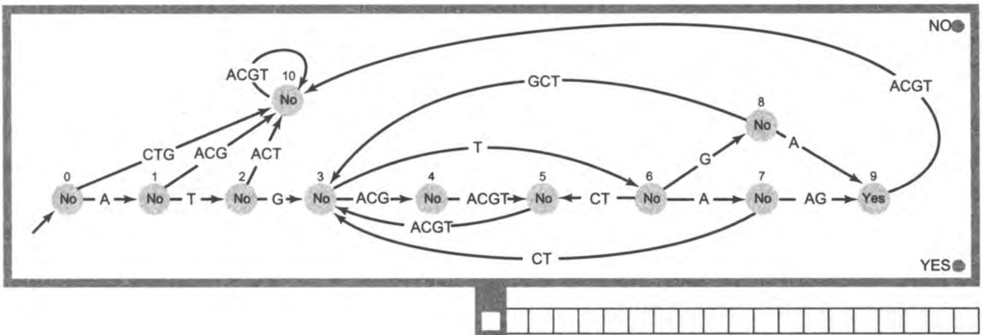


некоторые вхождения bbb



Другие примеры ДКА

вычислительных задач в различных практических областях: в текстовых редакторах для поиска по шаблону, в компиляторах для лексического анализа, в браузерах для разбора HTML, в операционных системах для пользовательских интерфейсов и т. д. Абстракция ДКА также подходит для описания управляющих модулей в физических системах: торговых автоматах, лифтах, автоматизированных системах регулирования движения, в процессорах компьютеров, не говоря уже о всевозможных естественных сущностях, от молекул и растений до человеческого генома. Во всех этих контекстах применима концепция системы с множеством дискретных состояний, которые сменяют друг друга по правилам, зависящим только от текущего состояния и от очередного входного значения.



ДКА для распознавания потенциальных геномов (см. листинг 3.1.1)

Говоря, что речь идет об абстрактных машинах, мы имеем в виду, что способ реализации такой машины в реальном мире не определен. Это решение позволяет свободно работать с разными представлениями ДКА в разных обстоятельствах. ДКА можно полностью задать в виде таблицы чисел, программы Java, электронной схемы или механического устройства. В самом деле, ДКА подходят для моделирования самых разных объектов, встречающихся в природе. Абстракция ДКА позволяет анализировать конкретные свойства сразу всех этих механизмов, как реальных, так и абстрактных.

Реализация ДКА на языке Java

Как построить ДКА? Собирать физическое устройство не нужно, потому что мы можем легко написать программу Java, называемую *универсальным виртуальным ДКА*. «Универсальность» подразумевает, что моделироваться может действие любого ДКА; «виртуальность» — что реализация представляет собой программу для некоторой машины, а не физическое устройство. В листинге 5.1.3 приведен код универсального виртуального ДКА, который получает описание (спецификацию) ДКА (из файла, переданного в командной строке) и последовательность строк из стандартного потока ввода и выводит результат выполнения ДКА для заданных входных строк.

Файл описания ДКА содержит количество состояний, за которым следует алфавит и перечисление состояний (по одному на строку). Каждая строка содержит

символы Yes или No (определяет, является состояние «принимающим» или «отклоняющим») и индекс состояния для каждого символа в алфавите; i-й индекс указывает направление перехода из этого состояния в том случае, если на вход ДКА подается i-й символ алфавита. В приведенном ниже файле b3.txt описан ДКА для распознавания языка *количество b кратно 3*, а в файле gene.txt — изображенный выше ДКА для распознавания потенциальных генов.

Конструктор создает структуры данных для хранения внутреннего представления ДКА, используя файл, который был передан через командную строку. Для этого он:

- читает количество состояний и алфавит;
- создает массив строк для статуса «принять/отклонить» каждого состояния и массив таблиц символов для переходов между состояниями;
- заполняет эти структуры данных, читая из файла признак «принять/отклонить» и переходы между состояниями для каждого состояния.

Код реализации конструктора вполне прост и прозрачен:

```
public DFA(String filename)
{
   In in = new In(filename);
   int n = in.readInt();
   String alphabet = in.readString();
   action = new String[n];
   next = (ST<Character, Integer>[]) new ST[n];
   for (int state = 0; state < n; state++)
   {
      action[state] = in.readString();
      next[state] = new ST<Character, Integer>();
      for (int i = 0; i < alphabet.length(); i++)
         next[state].put(alphabet.charAt(i), in.readInt());
   }
}
```

Другие методы листинга 5.1.3 тоже просты. Метод simulate() моделирует работу ДКА, а метод main() класса DFA вызывает метод simulate() для каждой строки в стандартном вводе.

Программа, приведенная в листинге 5.1.3, служит как исчерпывающим руководством по построению ДКА, так и незаменимым инструментом для изучения свойств конкретных ДКА. Вы предоставляете алфавит, табличное описание ДКА и последовательность входных строк, а программа сообщает, принимает ли ДКА каждую строку. Программа обеспечивает практическое решение задачи распознавания для ДКА и очень простой пример концепции виртуальной машины, к которой мы еще вернемся позднее. Это не реальное вычислительное устройство, а полное определение того, как работает такое устройство. Считайте, что ДКА — своего рода «компьютер», который вы «программируете», задавая множество вершин и переходов, которые подчиняются правилам ДКА. Каждый конкретный экземпляр ДКА представляет собой «программу» для этого компьютера.

Листинг 5.1.3. Универсальный виртуальный ДКА

```
public class DFA
{
   private String[] action;
   private ST<Character, Integer>[] next;
   public DFA(String filename)
   {  /* См. с. 714. */  }

   public String simulate(String input)
   {
      int state = 0;
      for (int i = 0; i < input.length(); i++)
         state = next[state].get(input.charAt(i));
      return action[state];
   }

   public static void main(String[] args)
   {
      DFA dfa = new DFA(args[0]);
      while (StdIn.hasNextLine())
         StdOut.println(dfa.simulate(StdIn.readLine()));
   }
}
```

Программа моделирует работу ДКА, определенного в первом аргументе командной строки, для каждой строки из стандартного потока ввода. Программа выводит Yes, если строка входит в язык, определяемый ДКА, или No в противном случае.

```
% more b3.txt
3
ab
Yes  0 1
No   1 2
No   2 0

% java DFA b3.txt
babbbabb
Yes
babbb
No
```

```
% more gene.txt
11
ATCG
No    1 10 10 10
No   10  2 10 10
No   10 10 10  3
No    4  6  4  4
No    5  5  5  5
No    3  3  3  3
No    7  5  5  8
No    9  3  3  9
No    9  3  3  3
Yes  10 10 10 10
No   10 10 10 10
```

```
% java DFA gene.txt
ATGTCTCTTTAG
Yes

ATGCCTCTTGA
No

ATGCCTCCTCCTTTCTAA
Yes

ATGTAA
Yes

ATGAAATAATAA
No
```

Недетерминированность

Теперь рассмотрим расширение нашей модели ДКА, которая отличается от наших типичных представлений о компьютерах в реальном мире. Для чего мы это делаем? Потому что эта модель не только позволяет определить более компактный автомат, с которым удобнее работать, но и помогает глубже разобраться в сути проблемы,

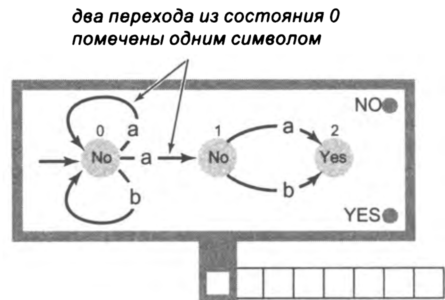
а это понимание способствует построению полезных практических программ и закладывает основу для поиска ответов на глубокие теоретические вопросы. В текущем контексте эта модель представляет связь между ДКА и регулярными выражениями, от которой мы придем к пониманию регулярных языков и практическому решению задачи распознавания для регулярных выражений.

Недетерминированные конечные автоматы

Поведение ДКА является *детерминированным*: для каждого входного символа и каждого состояния существует ровно один возможный переход в следующее состояние. Недетерминированный конечный автомат (НКА) представляет собой абстрактную машину с другими возможностями. А именно, НКА работают так же, как ДКА, но со снятием ограничения на переходы, ведущие из каждого состояния, поэтому в НКА:

- разрешены множественные переходы с одним и тем же входным символом;
- разрешены непомеченные (пустые) переходы;
- в переходах, выходящих из каждого состояния, могут отсутствовать некоторые из символов.

НКА не только может изменять состояние без чтения входного символа, но и для заданного текущего состояния НКА может иметь 0, 1 или несколько возможных переходов, соответствующих каждому из входных символов. Механизм выбора того, как и когда происходит переход, не определен. Введенные данные рассматриваются как приемлемые, если существует *любая* последовательность переходов, которая может перевести машину из исходного состояния (0) в состояние принятия. Пример НКА показан на рисунке. Когда автомат находится в состоянии 0 и читает символ a, он может решить остаться в состоянии 0 или перейти в состояние 1. А теперь проанализируем эту машину и определим, какие языки она распознает.



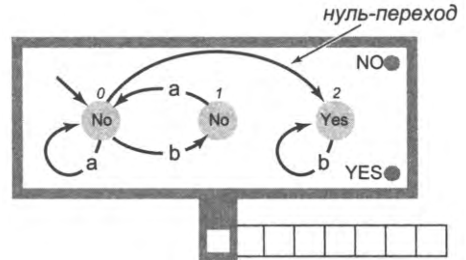
Недетерминированный конечный автомат

Примеры распознавания для НКА

Без детерминированных правил перехода между состояниями, характерных для ДКА, решение задачи распознавания для НКА потребует большего объема работы. Предположим, в нашем примере подается входная строка baaab: присмотревшись, вы увидите, что НКА может принять эти входные данные через переходы 0-0-0-1-2. Также возможны переходы 0-1-2 или 0-0-0-0, но эти последовательности не исчерпывают входных данных и не приводят к состоянию принятия; соответственно они нас не интересуют. В то же время легко увидеть, что для входной строки bbbb

НКА не сможет перейти из состояния 0 в состояние 1, поэтому машина должна включить индикатор «нет». Чтобы решить задачу распознавания для НКА, необходимо учесть все эти возможности. Эта необходимость резко отличается от соответствующей задачи для ДКА, где переход от состояния к состоянию происходит по простым правилам. В данном случае можно доказать, что НКА из нашего примера распознает язык всех строк, в которых предпоследним символом является a. Во-первых, он может принять любую строку, у которой предпоследним символом является a, оставаясь в состоянии 0 (исходном) до тех пор, пока не будет прочитан предпоследний символ; в этот момент НКА читает a, переходит в состояние 1, а затем в состояние 2 (состояние принятия) после чтения последнего символа. Во-вторых, не существует никакой последовательности переходов состояний, которая бы могла принять любую строку, у которой предпоследний символ отличен от a, потому что перейти в состояние 1 можно только так — с одним остающимся входным символом.

На этой иллюстрации представлен пример НКА с *нуль-переходом* (возможностью изменения состояния даже без чтения входного символа). Казалось бы, эта возможность только усложнит принятие решения о том, примет ли конкретный НКА конкретную строку, не говоря уже о том, какой язык распознает конкретный НКА. Чтобы проверить это утверждение, попробуйте убедиться в том, что язык, распознаваемый изображенным НКА, представляет собой множество всех двоичных строк, не содержащих подстроку bba. Назовем этот язык *отсутствие bba*.



НКА с нуль-переходом

Задача распознавания для НКА

Как решить задачу распознавания для НКА (то есть систематически проверить, принимает ли заданный НКА заданные входные данные)? Для ДКА этот процесс был простым, потому что на каждом шаге существовало только одно возможное изменение состояния. Однако для НКА проверяется существование *хотя бы одной* последовательности переходов состояний среди большого количества возможных.

Описывая НКА, мы не уточняем, как именно автомат выполняет свои вычисления; трудно представить, как построить подобную машину. Казалось бы, каждой машине придется изрядно потрудиться, чтобы не только найти последовательность переходов от исходного состояния в состояние принятия для вариантов входных данных, входящих в ее язык, но и доказать, что последовательности переходов для вариантов данных, не входящих в ее язык, не существует.

Работу такого НКА можно рассматривать как попытку подбора последовательности переходов к состоянию принятия: если последовательность существует, она

будет найдена. Интуиция подсказывает, что машина не может угадать результат, который она должна вычислить, но недетерминированность равносильна тому, что она на это способна. В таком языке, как Java, понадобилась бы конструкция вида:

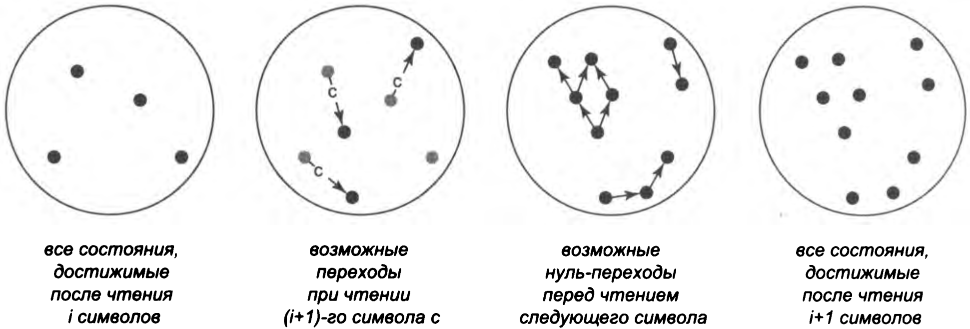
```
выполнить либо { одна команда } либо { другая команда};
```

Кажется, что недетерминированность только уводит нас от реального мира. Как программе Java решить, что ей делать? Однако концепция недетерминированности не только имеет практическую значимость, но и играет важнейшую роль в понимании фундаментальной природы вычислений.

К счастью, работа НКА легко моделируется отслеживанием *всех возможных* переходов состояний.

- Начать с нахождения всех возможных состояний, достижимых по нуль-переходам из исходных состояний. Результат определяет множество возможных состояний, которые могут быть достигнуты НКА перед чтением первого символа.
- Найти все состояния, которые могут быть достигнуты по ребрам, помеченным первым входным символом, а затем все состояния, достижимые от этих состояний по нуль-переходам. Результат определяет множество возможных состояний, которые могут быть достигнуты НКА перед чтением второго символа.
- Повторять процесс с отслеживанием всех возможных состояний, которые могут быть достигнуты НКА перед чтением каждого входного символа, пока входные данные не будут исчерпаны.

Если в конце этого процесса в множестве состояний, оставшихся после исчерпания входных данных, останется хотя бы одно состояние принятия, значит, входная строка принадлежит языку, принимаемому НКА. Если ни одного состояния принятия в множестве состояний не осталось или множество возможных состояний стало пустым до исчерпания входных данных, значит, входная строка в язык НКА не входит. Эти действия *детерминированы*: они объясняют, какие вычисления лежат в основе работы любого НКА, и никакие догадки здесь не нужны.

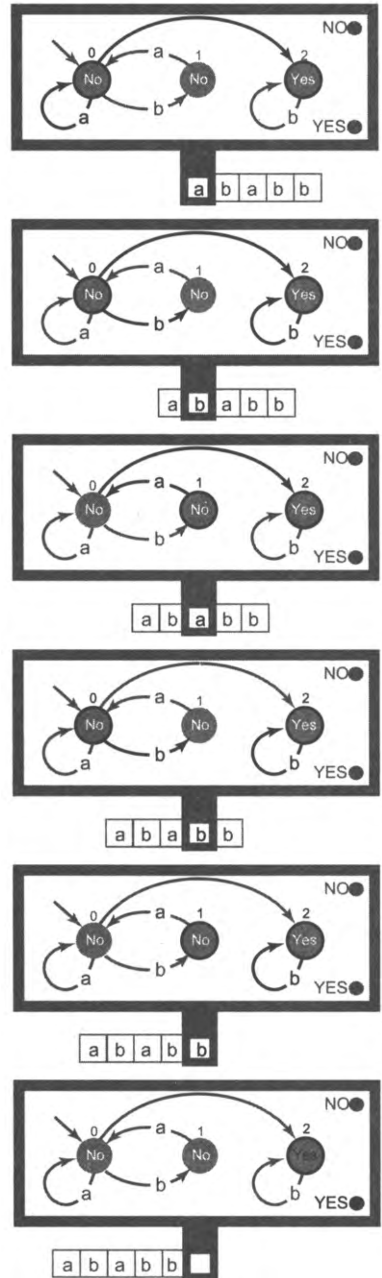


Один шаг моделирования работы НКА

Пример трассировки НКА

Для примера рассмотрим работу автомата НКА для уже упоминавшегося языка *отсутствие bba* при вводе ababb (см. рисунок). Автомат начинает с множества состояний {0, 2}, так как 0 является исходным состоянием и существует нуль-переход в 2. Множество возможных состояний после чтения первого входного бита (a) по-прежнему имеет вид {0, 2}, поскольку возможен только a-переход из 0 в 0 и последующий нуль-переход в 2. После чтения второго входного бита (b) НКА может находиться в состоянии 1 (b-переход из 0) или в состоянии 2 (b-переход из состояния 2 обратно в него же). С третьим битом a единственный a-переход из {1, 2} возвращает автомат в состояние 0, а нуль-переход из 0 в 2 возможен всегда, поэтому после чтения третьего бита множество возможных состояний имеет вид {0, 2}.

После четвертого бита a-переходы из {0, 2}, как и прежде, ведут к {1, 2}, а b-переход из состояния 2 дает множество {2} после пятого бита. Так как НКА может при исчерпании входных данных остаться в состоянии 2 (состояние принятия), мы знаем, что НКА примет строку ababb (и можем привести последовательность переходов состояний, которые для этого потребуются машине: 0-0-1-0-2-2-2). Если бы во введенных данных был еще один дополнительный символ a, то множество возможных состояний после его чтения было бы пустым (так как после пятого бита единственно возможным состоянием является состояние 2, не имеющее a-переходов); следовательно, строка ababba этим НКА будет отклонена.



Трассировка работы НКА

Для ДКА мы разработали решение задачи распознавания (листинг 5.1.3), основанное на моделировании работы ДКА, шаг за шагом. Чтобы сделать то же самое для НКА, мы используем тот же подход, как в только что описанном методе трассировки: вместо того чтобы хранить одиночное состояние в счетчике, программа отслеживает *множество* состояний: множество всех

состояний, которые НКА может достичь после чтения данных до текущего символа. Реализация остается читателю для самостоятельной работы. Идея машины, которая может угадать правильный ответ, кажется фантастикой, но такая программа позволяет изучать и использовать недетерминированность так, словно она вполне реальна. И снова эту программу на Java можно рассматривать как «компьютер», а каждый конкретный НКА — как «программу» для этого компьютера. Возникает закономерный вопрос — будет ли компьютер, наделенный способностью угадывать, более мощным, чем «компьютер для ДКА» — универсальный виртуальный ДКА из листинга 5.1.3?

Теорема Клини

Между регулярными выражениями, ДКА и НКА обнаруживается неожиданная связь, которая имеет серьезные практические и теоретические последствия. Эта связь известна под названием *теоремы Клини* — по имени логика Стивена Клини (Stephen Kleene), установившего в 1951 году взаимосвязь между автоматами и регулярными языками.

Теорема (Клини, 1951). Регулярные выражения, ДКА и НКА являются эквивалентными моделями в том смысле, что все они характеризуют регулярные языки.

Идея доказательства: см. ниже.

Теорема Клини — первый серьезный теоретический результат, который мы рассматриваем, поэтому стоит сказать несколько слов о концепции математического доказательства. В этом вводном материале мы не доказываем теоремы в стандартном математическом смысле. Представляя формулировку теоремы, мы стремимся сообщить вам о важном факте, а набросок доказательства приводится для того, чтобы убедить вас в истинности теоремы. Мы приводим общую идею доказательства и такое количество подробностей, чтобы вы поняли суть, а читатели, искушенные в доказательствах, могли самостоятельно заполнить все пробелы. Наши первые доказательства являются *конструктивными* и, пожалуй, проверяются проще, чем другие виды доказательств (которые, например, могут зависеть от логических принципов), потому что мы включаем примеры для пояснения конструкций. И хотя примеры не являются доказательством (как и тестовые примеры лишь демонстрируют правильность программы, но не доказывают, что она правильна *всегда*), они становятся необходимым первым шагом. Если вы прочитаете доказательство, проанализируете пример, а потом вернетесь к доказательству, вам будет проще убедиться в истинности доказываемого утверждения. Конечно, вас проще убедить, чем опытного математика, но поверьте: мы вас не обманываем!

Стратегия доказательства

По определению язык является регулярным, если вы можете написать описывающее его регулярное выражение, а любое регулярное выражение описывает ре-

гулярный язык. Чтобы доказать теорему Клини, нужно показать, что то же самое относится и к ДКА с НКА. Это будет сделано за два этапа: сначала мы докажем эквивалентность ДКА и НКА, а потом докажем эквивалентность НКА и регулярных выражений.

Эквивалентность НКА и ДКА

*НКА, распознающий выражение (a|b)*a(a|b)*

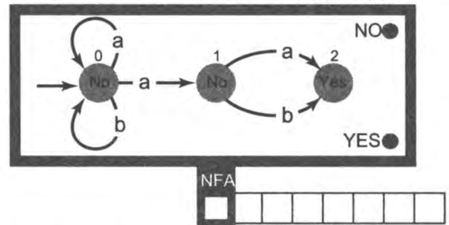
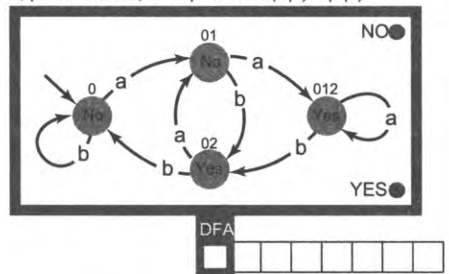


Таблица преобразования НКА в ДКА

имя	принимается?	a	b
ϕ	No	ϕ	ϕ
0	No	01	0
1	No	2	2
2	Yes	ϕ	ϕ
01	No	012	02
02	Yes	01	0
12	Yes	2	2
012	Yes	012	02

*ДКА, распознающий выражение (a|b)*a(a|b)*



Преобразование НКА в ДКА

Каждый ДКА также является НКА, поэтому для установления эквивалентности можно описать способ построения ДКА, распознающего тот же язык, что и любой заданный НКА. Первоначально этот ДКА содержит одно состояние, соответствующее каждому возможному множеству состояний НКА. Отсюда следует, что если НКА имеет n состояний, то ДКА имеет 2^n (потенциальных) состояний. Для любого заданного НКА существует однозначно определенное множество возможных состояний после чтения любой части входных данных, поэтому ДКА может справиться со всеми возникающими возможностями. Чтобы разобраться с переходами, мы воспользуемся тем же подходом, который был описан при обсуждении работы НКА. Для заданного множества состояний НКА и входного символа множество состояний НКА строится объединением всех состояний НКА, доступных из любого состояния исходного множества по одному ребру, помеченному символом, и любому количеству нуль-ребер. Такое множество состояний НКА соответствует одному состоянию того ДКА, который мы строим. Для каждого текущего состояния ДКА и каждого символа алфавита можно однозначно вычислить новое состояние. Для экономии из ДКА можно исключить любые состояния, недоступные из состояния 0. Наконец, мы определяем как состояние принятия каждое состояние ДКА, у которого соответствующее множество состояний НКА содержит состояние принятия.

Пример на диаграмме демонстрирует процесс построения ДКА для НКА, распознающего двоичные строки с предпоследним символом a. НКА имеет три состояния,

поэтому конструируемый ДКА имеет до 8 состояний, обозначенных на диаграмме как подмножества состояний НКА, которым они соответствуют: (пустое состояние), `0`, `1`, `2` (подмножество одиночных состояний), `01`, `02`, `12` (подмножество пар состояний) и `012` (все три состояния НКА.) Для каждого из этих состояний ДКА можно определить состояния-преемники для каждого входного символа. Например, если входной символ `a` читается в состоянии `01`, в НКА может быть выбран любой из переходов `0-0`, `0-1` или `1-2`, поэтому состоянием-преемником для `a`-перехода из состояния `01` в ДКА является состояние `012`; аналогичным образом, если в состоянии `01` будет прочитан входной символ `b`, в НКА может быть выбран любой из переходов `0-0` или `1-2`, поэтому состоянием-преемником для `b`-перехода из состояния `01` в ДКА является состояние `02`. В таблице преобразования приведены восемь возможных вариантов, которые должны быть проанализированы подобным образом. Обратите внимание: состояния `1` и `12` не являются конечными точками перехода по каким-либо ребрам; состояние `2` может быть достигнуто только из этих состояний, а состояние Ф может быть достигнуто только из состояния 2 и самого себя. Значит, ни одно из этих состояний не может быть достигнуто из состояния 0, и из ДКА все эти состояния можно исключить.

Чтобы убедиться в эквивалентности ДКА и НКА, необходимо быть уверенным в том, что теоретически этот процесс может быть выполнен для любого НКА (хотя работа может оказаться нудной из-за большого количества анализируемых вариантов)[1]. Другие примеры имеются в упражнениях. *Примечание*: для НКА с нуль-переходами прежде всего необходимо вычислить исходное состояние (см. упражнение 5.1.14).

Короче говоря, не существует формального языка, который бы распознавался некоторыми НКА, но не распознавался бы ни одним ДКА. Априорно кажется, что недетерминированные конечные автоматы должны быть мощнее детерминированных, но это не так. В отношении вычислительной мощности они эквивалентны.

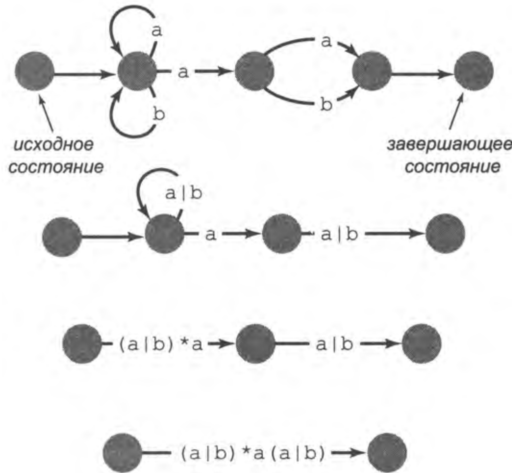
Эквивалентность НКА и регулярных выражений

Чтобы сконструировать регулярное выражение, которое описывает язык, распознаваемый заданным НКА, мы действуем следующим образом.

- Расширяем модель НКА с n состояниями так, чтобы метками ребер служили регулярные выражения.
- Добавляем исходное состояние с нуль-ребром, ведущим в исходное состояние НКА, и завершающее состояние с нуль-ребрами, ведущими из каждого состояния принятия.
- Последовательно исключаем состояния из НКА (добавляя более сложные регулярные выражения к его ребрам), пока не останутся только исходное и конечное

[1] Правильно определить набор состояний — обычно основная трудность при проектировании конечных автоматов, включая и реализацию их в программах. — *Примеч. науч. ред.*

состояния, соединенные одним ребром, помеченным регулярным выражением. Это регулярное выражение и является результатом.



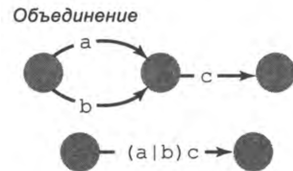
Преобразование НКА в регулярное выражение

Три базовые операции, используемые для исключения состояний, изображены справа. Для простоты они показаны с односимвольными метками ребер; на самом деле могут использоваться любые регулярные выражения. Кроме того, они не учитывают возможность того, что в исключаемую вершину входят и выходят другие ребра. Но любое состояние x всегда может быть удалено из любого расширенного НКА за четыре шага.

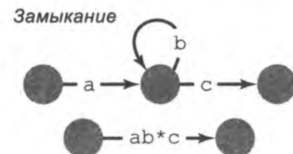
- Объединить ребра, входящие в x из одного источника, используя операцию |.
- Объединить ребра, выходящие из x в один приемник, используя операцию |.
- После выполнения предыдущих операций у x остается не более одной петли (обозначим ее метку R). Для каждой пары из входного/выходного ребра (обозначим их метки S и D соответственно) создать ребро с пометкой SR*D, которое идет из источника входного ребра в приемник выходного ребра (и, таким образом, обходит x).
- Удалить состояние x и все ребра, входящие и выходящие из него.

Конкатенация

Объединение



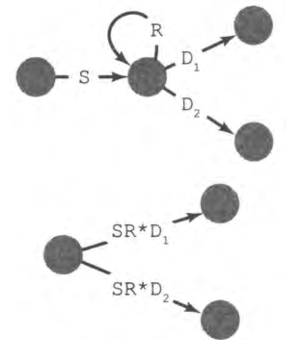
Замыкание



Три способа удаления вершины из расширенного НКА

Пример этой базовой операции изображен ниже справа. Легко убедиться в том, что эти операции дают расширенный НКА, эквивалентный исходному. Новый НКА содержит больше ребер, чем исходный (когда вершина имеет m входных и n выходных ребер, мы заменяем вершину и эти ребра mn ребрами), но на одно состояние меньше. Следовательно, мы можем применять эту операцию до тех пор, пока не останется ни одного состояния (всего одно ребро). Пометка этого ребра представляет собой регулярное выражение, описывающее язык, распознаваемый НКА. Шаги по преобразованию НКА из нашего примера (для двоичных строк с предпоследним символом a) в регулярное выражение представлены на с. 723. Процесс определен достаточно четко, чтобы мы могли написать программу Java для выполнения преобразования, но нас сейчас интересует только доказательство: любой НКА можно преобразовать в регулярное выражение, описывающее язык, распознаваемый НКА.

Применение той же процедуры в обратном направлении позволяет построить НКА, который распознает тот же язык, что и заданное регулярное выражение. Отправной точкой становится тривиальный расширенный НКА с регулярным выражением на одном ребре, а построение НКА осуществляется последовательным добавлением состояний и упрощением регулярных выражений на ребрах до тех пор, пока у каждого ребра не останется метка a, b или метки вообще не будет. Мы начинаем с расширенного НКА с одним состоянием, входным ребром, помеченным заданным регулярным выражением. Каждое ребро с нетривиальной пометкой должно иметь форму (R), RS, R|S или R* (где R и S — регулярные выражения), а следовательно, может быть упрощено одним из следующих способов.

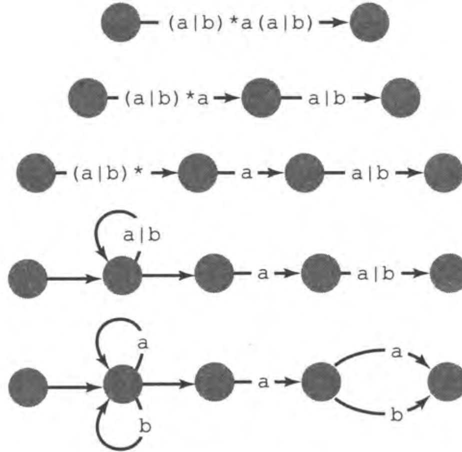


Исключение узла НКА с двумя выходными ребрами

- Если пометка ребра имеет форму (R), круглые скобки убираются.
- Если пометка ребра имеет форму RS, ребро заменяется новым состоянием и двумя новыми ребрами: одно с пометкой R идет от исходного состояния к новому (добавленному), а другое с пометкой S идет от нового состояния к состоянию-преемнику.
- Если пометка ребра имеет форму R|S, ребро заменяется параллельными ребрами: одно имеет пометку R, а другое пометку S.
- Если пометка ребра имеет форму R*, ребро заменяется непомеченным ребром и петлей с пометкой R.

Применяя любую из этих конструкций, мы получаем расширенный НКА, который распознает тот же язык, но его ребра упрощаются. Поэтому мы можем продолжать до тех пор, пока у нас не останется обычный НКА, ребра которого будут иметь метки a или b или будут непомеченными. Пример представлен на с. 725. По сути,

происходит то же самое, что и при преобразовании НКА в регулярное выражение, но в обратном направлении. Как и в предыдущем случае, упражнения в конце этого раздела помогут вам убедиться в том, что этот метод позволяет построить НКА для любого заданного регулярного выражения.



Преобразование регулярного выражения в НКА

Итак, нам удалось установить, что ДКА, НКА и регулярные выражения эквивалентны (теорема Клини). Любой язык, описываемый любым регулярным выражением, ДКА или НКА, является регулярным, а любой регулярный язык может быть описан некоторым регулярным выражением, ДКА или НКА. Мы привели подробное описание этого результата для того, чтобы вы привыкли доказывать факты, относящиеся к абстрактным машинам; чтобы показать, что даже простейшие машины могут привести к неожиданным теоретическим результатам (недетерминированность не делает конечные автоматы более мощными!); чтобы познакомить вас с интересной и функциональной, но при этом хорошо понятной вычислительной моделью и описать базовый метод, который может адаптироваться для поддержки действительно глубоких идей, лежащих в основе теории вычислений.

Во всей этой истории кроется один подвох: мы сосредоточились на *возможности* выполнения этих преобразований; но затраты на их выполнение — совсем другое дело. Например, ДКА, соответствующий НКА с n состояниями, может иметь до 2^n состояний. В самом деле, в нашем примере известно, что ни один ДКА с менее чем 2^{n-1} состояний не сможет распознать язык всех двоичных строк, содержащих a в n битах от конца. Для больших n эти затраты чрезмерно велики и неприемлемы. Таким образом, сейчас мы знаем, что *возможно* сделать с регулярными выражениями, ДКА и НКА, но на практике необходимо принимать во внимание затраты при применении рассмотренных концепций. Мы вернемся к этой теме в разделе 5.5.

Применение теоремы Клини на практике

Теорема Клини является важным теоретическим результатом и одновременно находит прямое практическое применение. Сейчас будут рассмотрены два важных применения этой теоремы, по одному из области теории и области практики.

Распознавание регулярных выражений

Фундаментальное практическое применение этой теории — служить основой решения задачи распознавания для регулярных выражений: если заданы регулярное выражение и строка, то входит ли строка в язык, определяемый регулярным выражением? Наше доказательство теоремы Клини является конструктивным, поэтому оно открывает прямой путь к построению программы, решающей задачу распознавания для регулярных выражений.

- Постройте НКА, соответствующий заданному регулярному выражению.
- Смоделируйте действие НКА для заданной входной строки.

Именно такой подход выбран для реализации метода Java `matches()`, упоминавшегося ранее в этом разделе. Реализация требует особого внимания, чтобы вы могли быть уверены в том, что затраты остаются под контролем. Полное описание и реализация приведены в нашей книге *Algorithms, Fourth Edition*. Перед вами пример парадигмы, имеющей фундаментальное значение для информатики.

- Выбрать промежуточную абстракцию (в данном случае НКА).
- Построить модель, которая конкретизирует эту абстракцию.
- Построить компилятор, преобразующий практическую задачу в абстракцию.

Точно такой же базовый процесс используется для компиляции программ на языке Java в программу для виртуальной машины Java (которая сама по себе является абстракцией, моделируемой программами для конкретных компьютеров, написанными на низкоуровневом языке).

Ограничения возможностей ДКА

Теорема Клини также помогает пролить свет на один фундаментальный теоретический вопрос: какие формальные языки можно описать регулярным выражением, а какие нельзя?

Чтобы хотя бы приступить к ответу на этот вопрос, рассмотрим язык всех двоичных строк с равным количеством `a` и `b`. Пожалуй, этот язык по крайней мере не проще большинства языков, рассматривавшихся в этом разделе. Можно ли написать регулярное выражение, описывающее этот язык? Безусловно, такая задача вполне может встретиться на практике. Однако в действительности данный язык *не является* регулярным, поэтому описать его регулярным выражением невозможно.

Как доказать этот результат? Теорема Клини позволяет избежать прямых рассуждений о выражении чего-либо с помощью регулярных выражений; вместо этого мы

работаем с абстрактными машинами. Эквивалентность регулярных выражений, ДКА и НКА, установленная теоремой Клини, означает, что мы можем взять любую из этих форм для представления концепции регулярного языка. В данном случае мы воспользуемся ДКА.

Утверждение. Не все формальные языки являются регулярными.

Доказательство. Воспользуемся доказательством от противного: сначала предположим, что язык *равное количество a и b* является регулярным, следовательно, мы можем записать регулярное выражение, которое его описывает. Согласно теореме Клини, мы также можем построить ДКА, распознающий этот язык. Пусть *n* — количество состояний в этом ДКА, который принимает любую строку с равным количеством символов a и b. В частности, он распознает строку, состоящую из *n* символов a, за которыми следуют *n* символов b. Теперь рассмотрим трассировку состояний, посещаемых ДКА в процессе принятия этой строки. Так как ДКА имеет только *n* состояний, принцип Дирихле утверждает, что ДКА должен заново посетить одно (или более) из состояний во время чтения a, потому что трассировка содержит *n + 1* состояний, включая исходное. Например, если *n* = 10, трассировка могла бы выглядеть так:

```
 a a a a a a a a a a b b b b b b b b b b
0-3-5-2-9-7-3-5-2-9-7-1-5-4-2-9-6-8-7-8-7
```

В этом примере состояние 3 посещается заново после шестого перехода. Теперь мы можем построить другую входную строку, исключив из исходной строки a для каждого перехода между повторениями. В приведенном примере будут исключены пять a, соответствующих переходам 3-5-2-9-7-3; также можно сделать вывод, что трассировка этой строки должна иметь вид:

```
 a a a a a b b b b b b b b b b
0-3-5-2-9-7-1-5-4-2-9-6-8-7-8-7
```

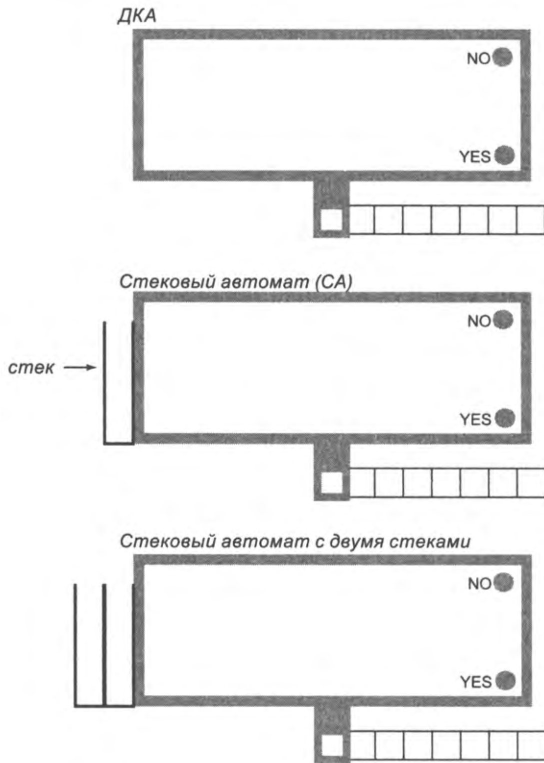
Ключевой момент заключается в том, что со вторым набором входных символов ДКА должен закончить работу в том же завершающем состоянии, как и с первым набором. Следовательно, он примет второй набор. Однако этот набор содержит меньше a, чем b, что противоречит предположению о распознании языка ДКА, которое следовало из предположения о существовании описывающего его регулярного выражения. Следовательно, мы доказали от противного, что никакое регулярное выражение не может описывать язык *равное количество a и b*.

В этом доказательстве упоминаются два простых приема, заслуживающих вашего внимания. Первый — *доказательство от противного*: чтобы доказать, что утверждение истинно, мы сначала предполагаем, что оно ложно. Затем мы шаг за шагом делаем логические заключения и приходим к заведомо ложному результату. Чтобы этот результат имел смысл, исходное предположение должно быть ложным; иначе говоря, утверждение должно быть истинным. Второй прием — *принцип Дирихле*: предположим, что $n + 1$ голубей прилетают в место с n и менее гнездами и каждый голубь размещается в гнезде. В этом случае в каком-то гнезде должны сидеть два и более голубя. Если у вас нет опыта математических доказательств, возможно, стоит перечитать доказательство с учетом этих приемов.

Казалось бы, доказать существование хотя бы одного нерегулярного языка — не такое уж достижение. Однако этот прием можно применить ко множеству других языков, и он поможет понять, какие характеристики присущи регулярным языкам. Многие простые языки, имеющие практическое значение, не являются регулярными.

Более мощные машины

Еще важнее другое: если вам известно, что язык не является регулярным, появляется критически важный вопрос. Как проще всего расширить модель абстрактной машины, чтобы получить машину, способную распознать этот простой язык? Согласно теореме Клини, даже недетерминированность в этом не поможет.



Три абстрактные машины

А что поможет? Когда мы найдем такую модель, нужно будет понять множество языков, распознаваемых ее машинами, и затем проанализировать ограничения возможностей этих машин. Фундаментальное ограничение конечных автоматов заключается в том, что их количество состояний ограниченно, поэтому ограниченно и количество отслеживаемых ими аспектов. В только что рассмотренном примере автомат не может определить разницу между входной строкой, начинающейся с 10 символов a,

и строкой, начинающейся только с 5 символов. Один из простых способов преодоления этого недостатка основан на добавлении в ДКА стека; в результате получается машина, называемая *стековым* (или *магазинным*) *автоматом* (СА). Нетрудно создать стековый автомат, способный распознавать равные количества a и b за счет хранения лишних символов в стеке (см. упражнение 5.1.44). Кроме того, простой, хорошо изученный и очень полезный класс языков — так называемые *бесконтекстные (контекстно-свободные) языки* — эквивалентен языкам, распознаваемым стековыми автоматами. Например, к контекстно-свободным языкам относится множество всех возможных регулярных выражений, как и базовое подмножество Java.

Существуют ли языки, которые не распознаются СА? Существуют ли автоматы более мощные, чем СА? Да, такие языки и машины существуют. Возможно, вы ожидаете, что мы сейчас начнем рассматривать длинный список моделей машин, каждая из которых чуть мощнее предыдущей, но список на самом деле получается довольно коротким. Как будет показано в следующем разделе, достаточно добавить второй стек для получения машины, которая по своей функциональности не уступает любой, которую только можно себе представить!

Выводы

Регулярные выражения, ДКА и НКА представляют собой эквивалентные модели для описания множества языков, называемых регулярными языками. Эта связь позволяет нам доказывать различные положения, относящиеся к регулярным языкам, и разрабатывать программы, которые используют их свойства для решения широкого круга базовых вычислительных задач.

Существует много языков, которые представляют интерес, но не являются регулярными, — какие формальные механизмы существуют для их определения? Существуют ли языки, которые не могут быть описаны при помощи этих более общих механизмов? Подобные вопросы и отношения между языками, формальные методы их описания и абстрактные машины, упоминаемые в теореме Клини, уточняют контекст для вопросов, которые упоминались в начале этой главы.

- Являются ли некоторые компьютеры по своей природе более мощными и функциональными, чем другие?
- Какие задачи можно решать с помощью компьютера?
- Существуют ли пределы тому, на что способны компьютеры?

В этом разделе мы проанализировали эти вопросы для конечных автоматов — простейших абстрактных компьютеров, которые можно определить точно. При этом были упомянуты многочисленные практические применения этих простейших абстрактных моделей. Более того, в список были добавлены два фундаментальных вопроса.

- Какая связь существует между компьютерами и языками?
- Расширяет ли недетерминированность возможности этих машин?

Далее эти вопросы будут рассмотрены для более функциональных вычислительных моделей. Те модели, которые необходимо рассмотреть, лишь немногим сложнее ДКА и НКА, но они распространяются на все известные вычислительные устройства. Замечательная история этих вычислительных моделей занимает центральное место в информатике.

Вопросы и ответы

В. Что такое символ?

О. Символы являются основными абстрактными структурными элементами формальных языков. С математической точки зрения определение символов несущественно: при изучении формальных языков изучаются множества последовательностей символов. С точки зрения ученого или инженера, символы важны, потому что они образуют очень узкий «мостик» между абстрактным миром формальных языков и реальным миром, в котором они используются. Символом может быть переключатель с двумя состояниями, ген, нейрон, молекула или абстрактное понятие: бит, цифра, буква нижнего регистра или символ Юникода.

В. Пересечение также относится к числу фундаментальных операций с множествами. Почему бы не включить оператор пересечения в базовое определение регулярных выражений?

О. Хороший вопрос. Так как языки — всего лишь множества, многие понятия переходят из теории множеств в регулярные выражения. Для языков можно определить пересечение, причем пересечение двух регулярных языков является регулярным (см. упражнение 5.1.37).

В. Экранирующие символы для определения метасимволов только запутывают дело. Нет ли записи попроще?

О. Напротив. Когда вы задаете регулярное выражение в строковом литерале Java, вам нужен еще один уровень экранирования для метасимволов строковых литералов. Например, для представления пропускного символа следует ввести `"\\s"`, а для представления одиночного символа слеша — `"\\\\"`. Также иногда приходится экранировать специальные символы в командной строке. Например, в аргументе командной строки на с. 704, представляющем регулярное выражение для денежных сумм в долларах, запись `"\\\$"` используется для обозначения знака доллара, а запись `"\\."` для обозначения десятичной точки.

В. Для чего изучать НКА? Они намного сложнее ДКА, а преимущества особо не заметны.

О. Мы использовали их для доказательства теоремы Клини, и их можно использовать для реализации распознавания регулярных выражений. Другая причина заключается в том, что они имеют гораздо меньше состояний, чем соответствующие ДКА. Попробуйте построить ДКА для множества всех строк, у которых десятым

символом с конца является 1. Сделать это менее чем с 512 состояниями не удастся, а НКА для этого языка имеет всего 10 состояний.

В. Как написать регулярное выражение для определения одноэлементного множества, содержащего пустую строку?

О. Мы используем запись ∈ для обозначения одноэлементного множества, содержащего строку длины 0. Запись ∅ обозначает пустое множество, не содержащее строк. Формально обе записи должны быть включены в наше формальное определение регулярных выражений.

В. Что является замыканием пустого множества?

О. Пустая строка: `{}* = ∈`.

Упражнения

5.1.1. Напишите программу Java, решающую задачу распознавания для палиндромов. Программа должна получать последовательность входных строк из стандартного потока ввода. Для простоты следует считать, что используется латинский алфавит. Для каждой строки из стандартного потока ввода программа должна выводить строку и сообщение «`is a palindrome`», если строка является палиндромом, или «`is not a palindrome`», если строка палиндромом не является.

5.1.2. По образцу упражнения 5.1.1 напишите программу на Java, решающую проблему распознавания для языка *равное количество a и b*. Предполагается, что во входных данных присутствуют только `a` и `b`.

5.1.3. Докажите, что ДКА на с. 710 распознает язык *количество b кратно 3*.

5.1.4. Приведите краткое неформальное описание языков, задаваемых каждым из следующих регулярных выражений над алфавитом `{a, b}`:

a. `.*`

b. `a.*a|a`

c. `.*abbabba.*`

d. `.*a.*a.*a.*a.*`

5.1.5. Приведите регулярное выражение, которое определяет каждый из следующих языков над алфавитом `{a, b}`.

a. Все строки, кроме пустой строки.

b. Содержит не менее трех смежных символов `b`.

c. Начинается с символа `a` и имеет нечетную длину или начинается с `b` и имеет четную длину.

d. Не содержит смежных символов `b`.

e. Любая строка, кроме bb или bbb.

f. Начинается и заканчивается одним и тем же символом.

g. Содержит не менее двух символов a и не более одного b.

5.1.6. Напишите регулярное выражение Java для всех слов, содержащих пять гласных (a, e, i, o, u) в указанном порядке и не содержащих других гласных (например, *abstemious* и *facetious*.)

5.1.7. Напишите регулярное выражение Java для дат в форме *July 4, 1776*, которое по возможности включает все даты, которые вы считаете допустимыми, и никакие другие.

5.1.8. Напишите регулярное выражение Java для десятичных чисел с фиксированной точкой (конечных десятичных дробей). Такое число представляет собой последовательность десятичных цифр, за которой следует точка и еще одна последовательность цифр. По крайней мере одна из последовательностей цифр должна быть непустой. Если первая последовательность цифр содержит более одной цифры, она не должна начинаться с 0.

5.1.9. Напишите регулярное выражение Java для (упрощенных) литералов с плавающей точкой в Java. Литерал с плавающей точкой представляет собой конечную десятичную дробь (см. предыдущее упражнение), за которой может следовать буква e или E, далее необязательный знак + или - и целочисленная мантисса.

5.1.10. Пусть L — язык {aaaba, aabaa, abbba, ababa, aaaaa}. Идентифицируйте каждое из приведенных ниже регулярных выражений как определяющее язык, который (i) не содержит строк из L, (ii) содержит только некоторые строки из L и некоторые другие строки, (iii) содержит все строки из L и некоторые другие строки или (iv) совпадает с L.

a. `a(a|b)*abb(a|b)*`

b. `a(a|b)*a`

c. `a*b*aba`

d. `a((a*|b*)|(b*aba*))a`

e. `a*b*aa*b*ba*a*a*b*b*a*a*b*`

f. `(a|b)(a|b)(a|b)(a|b)a`

g. `(a|aa|aaa)(ba|aa|bbb)a`

5.1.11. Изобразите ДКА для распознавания языка всех строк с нечетным количеством a и четным количеством b. Создайте файл в формате, который ожидает получить класс DFA (листинг 5.1.3), и протестируйте свое решение.

5.1.12. Изобразите ДКА для распознавания номеров социального страхования. Создайте файл в формате, который ожидает получить класс DFA (листинг 5.1.3), и протестируйте свое решение.

5.1.13. Напишите программу Java, которая решает задачу распознавания для языка, заданного регулярным выражением `C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H`, для последовательности строк из стандартного потока ввода без использования поддержки регулярных выражений Java.

5.1.14. Изобразите ДКА для распознавания языка `a.*a|b.*b`. Затем изобразите НКА с меньшим количеством состояний, который распознает тот же язык.

5.1.15. Изобразите ДКА для распознавания языка `.*aabab.*` (множество всех строк, содержащих строку `aabab`). Затем изобразите НКА с меньшим количеством состояний, который распознает тот же язык.

5.1.16. Имеется торговый автомат, который принимает монеты по 5, 10 и 25 центов и продает товары стоимостью 25 центов. Изобразите ДКА с состоянием, соответствующим каждой внесенной сумме. Добавьте переходы, чтобы автомат находился в состоянии `i`, если внесенная сумма равна $5 \times i$.

5.1.17. Опишите, как преобразовать любой ДКА в ДКА, распознающий *дополнение* языка, распознаваемого исходным (множество строк над тем же алфавитом, но не входящих в исходный язык).

5.1.18. Приведите ответ на предыдущий вопрос для НКА.

5.1.19. Создайте ДКА, распознающий язык всех двоичных строк с нечетным количеством `a` и четным количеством `b`.

5.1.20. Преобразуйте ДКА из предыдущего упражнения в регулярное выражение.

5.1.21. Изобразите НКА, соответствующий следующим регулярным выражениям.

a. `a(a|b)*a`

b. `a*b*aba`

c. `(a|b)(a|b)a`

d. `a((a*|b*)|(b*aba*))*a`

e. `ab(a|b)ba | a(a|b)aba | aa(ab|ba|aa)a`

5.1.22. Преобразуйте НКА на с. 717 (который распознает строки, не содержащие подстроку `bba`) в ДКА.

5.1.23. Возможно ли построить регулярное выражение, которое описывает все двоичные строки с одинаковым количеством вхождений подстрок `ab` и `ba`?

Ответ: да: `(a.*a) | (b.*b)| a*|b*`. Действительно, ДКА не умеет «считать», но в данном случае язык эквивалентен всем двоичным строкам, которые начинаются и заканчиваются одним и тем же битом, так что подсчет не нужен.

5.1.24. Покажите, что никакой ДКА не может распознать множество всех палиндромов.

5.1.25. Покажите, что язык, содержащий все двоичные строки, не содержащие подстроку `bba`, но содержащие количество `b`, кратное 3, является регулярным.

Упражнения повышенной сложности

5.1.26. *Сложные регулярные выражения.* Постройте регулярное выражение, которое задает каждый из следующих языков двоичных строк над алфавитом `{0, 1}`.

a. Все двоичные строки, кроме 11 или 111.

b. Двоичные строки с 1 во всех нечетных позициях.

c. Двоичные строки, содержащие не менее двух 0 и не более одной 1.

d. Двоичные строки, не содержащие двух смежных 1.

5.1.27. *Делимость.* Для каждого из указанных случаев постройте регулярное выражение, которое задает все двоичные строки, которые при числовой интерпретации:

a. Делятся на 2;

b. Делятся на 3;

c. Делятся на 6.

5.1.28. *Поиск по регулярному выражению.* Проверьте различные приложения на вашем компьютере (браузер, редактор, медиатека или любое другое приложение, которым вы часто пользуетесь) и выясните, в какой степени их функциональность поиска поддерживает регулярные выражения.

5.1.29. *Делимость на 3.* Разработайте ДКА для распознавания языка всех двоичных строк, представляющих числа, делящиеся на 3. Например, строка `1111` принимается, потому что 1111 соответствует 15 в десятичной записи, но строка `1110` отклоняется. *Подсказка*: используйте три состояния. В зависимости от входных данных, прочитанных до настоящего момента, ДКА должен находиться в одном из трех состояний в зависимости от остатка от деления числа на 3.

5.1.30. *Противодребезговый фильтр. Конечный преобразователь* представляет собой ДКА, выводящий строку символов. Он отличается от ДКА только тем, что каждый переход помечается выходным символом, который должен выводиться при выполнении этого перехода. Разработайте преобразователь, который исключает из введенных данных одиночные (изолированные) символы. Например, каждое вхождение `aaba` должно заменяться на `aaaa`, а каждое вхождение `bbab` — на `bbbb`, но в остальном входные и выходные данные должны оставаться неизменными.

5.1.31. *Harvester.* Объект Java `Pattern` является представлением регулярного выражения. Такой объект может использоваться для построения объекта `Matcher` для заданной строки. Объект `Matcher` можно рассматривать как представление НКА для регулярного выражения. В набор операций объектов `Matcher` входит метод `find()` для нахождения следующего совпадения регулярного выражения в строке и метод `group()` для возвращения символов совпадения. Напишите приложение-клиент, использующее `Pattern` и `Matcher`, которое получает имя файла (или URL) и регулярное выражение в командной строке и выводит все подстроки файла, совпадающие с регулярным выражением.

Ответ:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class Harvester
{
   public static void main(String[] args)
   {
      In in = new In(args[1]);
      String re = args[0];
      String input = in.readAll();
      Pattern pattern = Pattern.compile(re);
      Matcher matcher = pattern.matcher(input);
      while (matcher.find())
         System.out.println(matcher.group());
   }
}
```

5.1.32. *Подсчет совпадений.* Разработайте программу, которая подсчитывает количество подстрок в стандартном потоке ввода, которые совпадают с регулярным выражением в командной строке, игнорируя возможные перекрытия совпадений (используйте классы `Pattern` и `Matcher` из предыдущего упражнения).

5.1.33. *Обход сети.* Измените решение упражнения 5.1.31 и разработайте программу, которая выводит все веб-страницы, доступные из веб-страницы, переданной в аргументе командной строки.

5.1.34. *Одноуровневые регулярные выражения.* Постройте регулярное выражение Java, которое задает множество строк, являющихся допустимыми регулярными выражениями над двоичным алфавитом, но без вложенных круглых скобок. Например, строка `(a.*b)*|(b.*a)*` входит в язык, а строка `(b(a|b)b)*` в него не входит.

5.1.35. *Поиск и замена.* Напишите фильтр SearchAndReplace.java, который получает как аргументы командной строки регулярное выражение и строку `str`, читает строку из стандартного потока ввода, заменяет в ней все совпадения регулярного выражения строкой `str` и отправляет результат в стандартный поток вывода. Сначала решите задачу с использованием метода `replaceAll()` из библиотеки Java `String`; затем решите ее без использования этого метода.

5.1.36. *Проверка пустого языка.* Добавьте в `DFA` (листинг 5.1.3) метод `isEmpty()`, который возвращает `true`, если распознаваемый машиной язык пуст, или `false` в противном случае.

Ответ: код использует поиск в ширину с классом `Queue` для отслеживания состояний, доступных из состояния 0 (см. листинг 4.5.4).

```
public boolean isEmpty()
{
    Queue<Integer> queue = new Queue<Integer>();
    boolean[] reachable = new boolean[n];
    queue.enqueue(0);
    while (!queue.isEmpty())
    {
```

```
        int state = queue.dequeue();
        if (action[state].equals("Yes")) return false;
        reachable[state] = true;
        for (int  i = 0; i < alphabet.length(); i++)
        {
            int st = next[state].get(alphabet.charAt(i));
            if (!reachable[st]) queue.enqueue(st);
        }
    }
    return true;
}
```

5.1.37. *Операции множеств с языками.* Для заданного НКА, распознающего два языка A и B, покажите, как построить НКА, распознающий объединение и пересечение A и B.

5.1.38. *Случайный ввод.* Напишите программу для оценки вероятности того, что случайная двоичная строка длины n будет распознаваться заданным ДКА.

5.1.39. *Преобразование НКА в ДКА.* Напишите программу, которая читает описание НКА и создает ДКА, распознающий тот же язык.

5.1.40. *Минимальные языки.* Для заданного регулярного языка L покажите, что множество минимальных строк в L также является регулярным. «Минимальность» означает, что если `x` принадлежит `L`, то `xy` не принадлежит `L` для любой непустой строки `y`.

Ответ: измените ДКА так, чтобы он, побывав однажды в состоянии принятия и выйдя из него, потом всегда оставался в состоянии отклонения.

5.1.41. *Обратные ссылки.* Покажите, что операция обратной ссылки (back reference) не является регулярным выражением и не может быть сконструирована из базовых операций регулярных выражений. Покажите, что никакой ДКА не распознает язык, состоящий из ww, где w — некоторая строка (например, *beriberi* и *couscous*), но регулярное выражение Java `(.*)\1` описывает этот язык.

5.1.42. *Универсальный виртуальный НКА.* Разработайте программу, аналогичную листингу 5.1.3, которая может моделировать работу произвольного НКА. Реализуйте метод, описанный в тексте, используя представление графа для НКА и `Queue` для хранения множества возможных состояний как в упражнении 5.1.36. Программа должна выводить трассировку множества возможных состояний для НКА перед чтением каждого символа. Протестируйте свой код, запустив его для НКА, описанного в тексте (распознающего множество все строк, у которых четвертый с конца символ равен 1), для входных данных `aaaaababaabbbaababbbb`.

5.1.43. *Универсальный виртуальный стековый автомат.* Стековый автомат добавляет в ДКА стек и возможность при каждом переходе заносить и извлекать символы из стека. А точнее, каждый переход снабжается дополнительной пометкой x/y, которая означает, что из стека нужно извлечь символ и занести в него y, если был извлечен символ x. Пометка $/y$ означает занесение y без извлечения чего-либо, а $x/$ — извлечение из стека без занесения, если был возвращен символ x. Пометка $/$

обозначает простое извлечение из стека. Разработайте программу, аналогичную приведенной в листинге 5.1.3, которая может моделировать работу любого стекового автомата.

5.1.44. *Стековый автомат для нерегулярных языков.* Изобразите стековый автомат (см. предыдущее упражнение), который распознает язык *равное количество a и b*. Затем протестируйте его с использованием решения предыдущего упражнения.

5.2. Машины Тьюринга

В своей статье 1937 года Алан Тьюринг представил одно из самых изящных и интересных интеллектуальных открытий XX века. Это простая модель вычислений, достаточно универсальная для воплощения любой компьютерной программы, которая закладывает основу теории вычислений. Благодаря простоте описания и поведения она хорошо подходит для математического анализа. Этот анализ привел Тьюринга к более глубокому пониманию компьютеров и вычислений и к откровению о том, что все известные вычислительные артефакты в глубоком техническом смысле эквивалентны, а некоторые вычислительные задачи вообще не решаются на компьютере независимо от скорости процессора или объема памяти. Эти идеи будут представлены в следующих двух разделах, но сначала необходимо заложить основу, определяя и рассматривая базовые свойства открытия Тьюринга. Если вы уже читали о машинах Тьюринга в других контекстах или столкнетесь с ними в будущем, вы заметите, что приведенные определения и описания некоторых свойств несколько отличаются от других источников. В действительности эти различия несущественны; один из постулатов этой теории заключается в том, что любая модель вычислений, которую вы когда-либо изобретете, будет эквивалентна любой достаточно мощной модели. Наш подход будет основан на адаптированной модели, разработанной Марвином Мински (Marvin Minsky) в Массачусетском технологическом институте в середине XX века.

Модель машины Тьюринга

Предметом нашего исследования является *машина Тьюринга* (TM, Turing Machine) — модель абстрактной машины, лишь немногим более сложная, чем модель ДКА из предыдущего раздела. В следующем описании отличия от ДКА выделены жирным шрифтом. Каждая машина Тьюринга состоит из следующих компонентов.

- Конечное множество состояний, каждое из которых относится к одной из следующих категорий: **левое состояние**, **правое состояние**, **состояние остановки**, состояние *принятия* или состояние *отклонения*.
- Множество *переходов*, которое определяет, какое состояние является следующим и **какой символ должен быть записан**. Каждое состояние имеет один переход для каждого символа в алфавите.

- Лента со строкой символов.
- *Каретка* (*управляющее устройство*, *головка чтения/записи*), которая может прочитать **или записать** символ и переместиться **влево или** вправо для чтения следующего символа.

Машина Тьюринга начинает работу в состоянии 0 с кареткой, установленной на первый (крайний левый) входной символ. Машина читает данные с ленты **и записывает их на ленту**, а также дискретно изменяет свое состояние по следующей схеме.

- Прочитать символ.
- Определить переход, связанный с этим символом и текущим состоянием.
- **Записать символ в соответствии с переходом.**
- Изменить состояние в соответствии с переходом.
- Переместить ленту на одну позицию вправо, **если новое состояние является правым.**
- **Переместить ленту на одну позицию влево, если новое состояние является левым.**
- **Если новое состояние является состоянием остановки, принятия или отклонения,** прекратить работу.
- Включить индикатор, соответствующий завершающему состоянию.

Каждая машина Тьюринга описывается графом, в котором состояния (принятия, отклонения, **остановки**, **левое**, **правое**) представлены вершинами, помеченными

Машина Тьюринга

Yes, No, H, L и R соответственно. Каждый переход представлен направленным ребром, помеченным парой символов из алфавита.

Пример машины Тьюринга над двоичным алфавитом изображен справа. Как и в случае с ДКА, вершины нумеруются целыми числами, начиная с 0. Каждый переход помечается парой символов, разделенных двоеточием. Первый символ идентифицирует переход как связанный с прочитанным символом; второй задает символ, который должен быть записан на место символа, только что прочитанного с ленты, при выполнении перехода.

Чтобы диаграмма была более компактной, мы обычно работаем с сокращенной версией: рамка, индикаторы и лента обычно на ней не изображаются. Также не изображаются следующие аспекты.

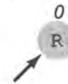
- Петли, не изменяющие ленту.
- Двоеточие и второй символ для переходов, изменяющих состояние, которые не изменяют ленту.

Одна непомеченная стрелка используется для обозначения изменений состояний для любого входного символа, не изменяющего ленту.

Правое и левое состояние называются *состояниями сканирования* — машина сканирует символы в неявных переходах (чтение и последующая запись того же символа), пока не будет прочитан символ, инициирующий переход в другое состояние. Например, в машине, изображенной на иллюстрации, состояние 0 осуществляет сканирование вправо, пока не встретит # во входных данных. На первых порах эти сокращения немного затрудняют чтение диаграмм, но с ними логика машины становятся более понятной; это станет очевидно, когда мы начнем изучать их более подробно.

Различия между моделью машины Тьюринга и моделью ДКА могут показаться второстепенными, но, как вы вскоре увидите, они имеют глубокие последствия. Поведение машин даже с несколькими состояниями может быть достаточно сложным. В частности, два последствия стоит упомянуть еще до того, как мы начнем. Во-первых, не существует никаких ограничений на перемещения каретки вправо или влево. Отсюда следует, что лента бесконечна — количество ячеек, записанных вправо или влево на ленте, не ограничивается. Мы используем метасимвол # для обозначения содержимого ячеек, которые не были достигнуты при перемещении, и считаем, что каждый раз, когда каретка переходит в новую позицию, там имеется ячейка с содержимым #. Во-вторых, количество переходов между состояниями машины Тьюринга не ограничено. Простейший пример обоих качеств — машина Тьюринга, состоящая из единственного правого состояния, которая содержит только неявные переходы (см. иллюстрацию на следующей странице). Машина просто двигается

Машина Тьюринга



Код Java

```
while (!StdIn.isEmpty())
   StdIn.readChar();
```

Бесконечные циклы

вправо, читая все новые и новые ячейки. На первый взгляд пример может показаться искусственным, но он, по сути, не отличается от программы Java, в цикле читающей данные из стандартного потока ввода, так как во всех программах Java предполагается, что размер этого потока не ограничен.

Рассмотрим несколько примеров машин Тьюринга для выполнения интересных и полезных вычислений.

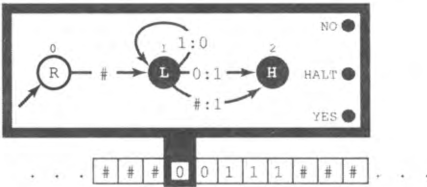
Инкремент двоичного числа

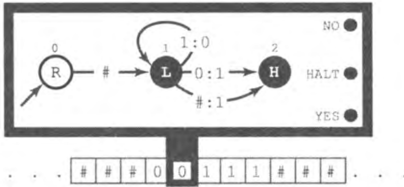
Какие вычисления выполняет машина Тьюринга, представленная в первом примере? Это операция *двоичного инкремента*: если на вход подано двоичное число, то это число увеличивается на 1. Если вы не знаете, как выполняются арифметические вычисления с двоичными числами, прочитайте раздел 6.1; впрочем, операция достаточно простая и знакомая, так что, возможно, делать это и не придется. Рассмотрим процесс увеличения на 1 (инкремента) двоичного числа 10111 (23 в десятичной записи) для получения результата 11000 (24 в десятичной записи). Как бы вам ни объясняли в школе, это тот же процесс, который используется для увеличения 999 на 1 с получением результата 1000 в десятичной записи. Как бы вы объяснили суть происходящего ребенку? Примерно так: мы двигаемся справа налево и заменяем 1 на 0, пока не обнаружим 0; этот 0 заменяется на 1. Наша машина Тьюринга реализует этот процесс, как подробно показано в трассировке на следующей странице. Начиная с состояния 0, машина сканирует входные символы вправо, пока не обнаружит первый знак # в правой позиции данных; с этого момента она начинает двигаться влево и переходит в состояние 1. Находясь в состоянии 1, она сканирует ячейки влево, пока читаются 1; при этом каждый символ 1 заменяется на 0. При достижении 0 машина заменяет его на 1, после чего переходит в состояние остановки. Если машина достигнет # до обнаружения 0 в этом сканировании справа налево, значит, исходная строка состояла только из 1 и все они были заменены 0, поэтому машина должна заменить начальный знак #, приведший к остановке, на 1 (например, результат инкремента #1111# равен #10000# — знак # в начале остается, потому что их запас не ограничен). Наш пример также показывает, что машина работает даже при наличии лидирующих (незначащих) 0.

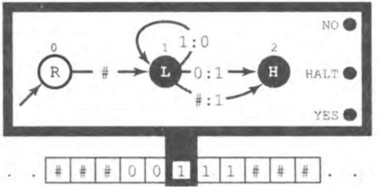
Компактный формат трассировки

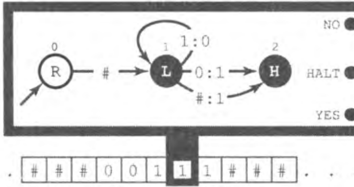
Вместо того чтобы воспроизводить всю машину Тьюринга на каждом шаге, мы в дальнейшем будем использовать гораздо более компактный формат трассировки. В нем отражаются текущее состояние, позиция каретки и содержимое ленты, а символы, записываемые на ленту, выделяются жирным шрифтом. В каждой строке показаны изменения в содержимом ленты и позиции каретки, происходящие в каждом состоянии; каждое изменение состояния соответствует переходу на новую строку. В каждой строке каретка находится в позиции символа, вызвавшего изменение состояния (который мог быть перезаписан).

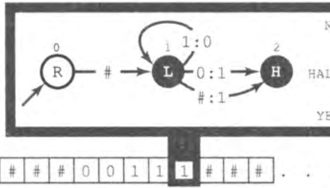
сканирование вправо до #

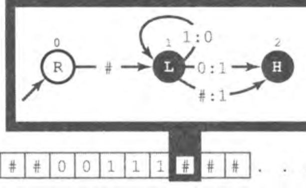




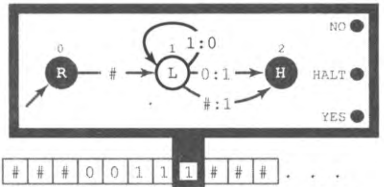


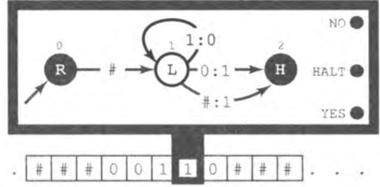


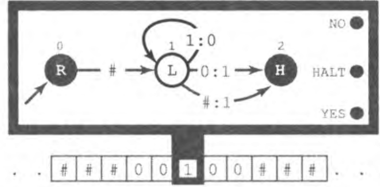


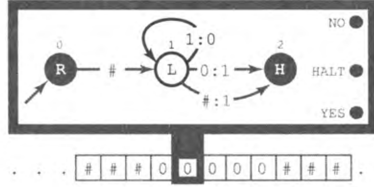


сканирование влево до 0, с заменой 1 на 0

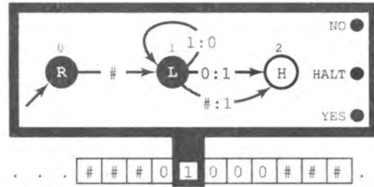


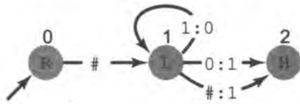






замена 0 на 1 и остановка





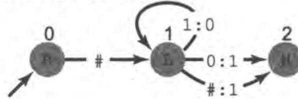
состояние	лента							описание
	#	0	0	1	1	1	#	начало
0	#	0	0	1	1	1	#	сканирование вправо до #
1	#	0	1	0	0	0	#	сканирование влево до 0 с инвертированием битов
2	#	0	1	0	0	0	#	остановка

Компактная трассировка машины Тьюринга для двоичного инкремента

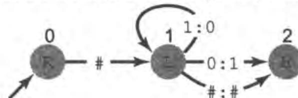
Похожие машины

У машины Тьюринга для двоичного инкремента есть одна особенность: она работает для целых чисел произвольной длины. Если на ленте записано двоичное целое число из миллиарда бит, оно тоже будет увеличено (и если число состояло из одних единиц, его длина возрастет до миллиарда и 1 бита). Возможность обработки таких больших чисел такой простой машиной выглядит впечатляюще. Если же заменить пометку `#:1` на `#`-переходе из состояния остановки на `#:#`, то мы получим машину для выполнения двоичного инкремента, которая не изменяет длину числа, но игнорирует переполнение (как многие реальные компьютеры). Еще одна разновидность — машина для *двоичного декремента с фиксированной длиной*, которая работает практически так же, если не считать изменения ролей 0 и 1: двигаясь справа налево, она заменяет 0 на 1, пока не обнаружит 1, которая будет заменена на 0. Это правило работает во всех случаях, кроме чисел, состоящих из одних 0; в этом случае все 0 будут заменены на 1 (двоичный декремент, который является настоящей обратной операцией по отношению к инкременту, должен удалять начальные 0 — см. упражнение 5.2.10). Обратите внимание: все три описанные машины Тьюринга практически одинаковы, если не считать небольших различий в переходах из состояния 1.

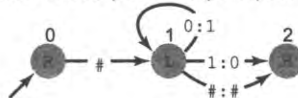
двоичный инкремент



двоичный инкремент с фиксированной длиной



двоичный декремент с фиксированной длиной

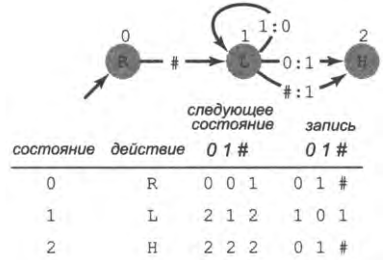


Три взаимосвязанные машины Тьюринга

На данный момент вам будет полезно проверить свое понимание базовых характеристик машин Тьюринга и нашего формата трассировки — попробуйте написать трассировку этих машин для разных вариантов входных данных.

Таблицы состояний

Как и в случае с ДКА, важно заметить, что любая машина Тьюринга легко задается таблицей, в которой одна строка представляет одно состояние. В ней указывается тип состояния (правое, левое, принятие, отклонение, остановка), а также приводятся следующие состояния и заменяющие символы для всех возможных входных символов. Это представление изображено справа для машины двоичного инкремента, которую мы рассматривали ранее. Конечно, неявные переходы в этой таблице указываются явно.



состояние	действие	следующее состояние 0 1 #	запись 0 1 #
0	R	0 0 1	0 1 #
1	L	2 1 2	1 0 1
2	H	2 2 2	0 1 #

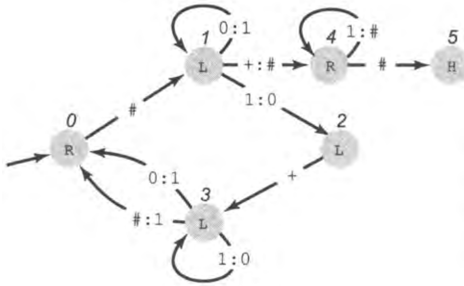
Представление машины Тьюринга в виде таблицы состояний

Двоичный сумматор

Машина Тьюринга на диаграмме на с. 744 (вверху) работает как *сумматор*: она заменяет введенное a+b на входной ленте суммой a и b (предполагается, что a, b и их сумма — положительные целые числа, представленные в двоичной записи). Например, если при запуске машины лента содержит **#1011+1010#**, то она вычислит сумму $11_{10} + 10_{10} = 21_{10}$, и при остановке на ленте останется строка **#10101#**. Как и в случае с двоичным инкрементом, размер чисел может быть произвольным — машина Тьюринга может вычислить сумму независимо от количества битов на входе. Стратегия таких вычислений заключается в уменьшении b с увеличением a до тех пор, пока b не уменьшится до 0. Для выполнения этой работы машине потребуются шесть состояний.

- Состояние 0 проводит сканирование вправо до конца данных.
- Состояние 1 уменьшает на 1 число справа от знака +.
- Состояние 2 проводит сканирование влево до правого края числа слева от знака +.
- Состояние 3 увеличивает на 1 число слева от знака +.
- Состояние 4 достигается при попытке уменьшения 0 — все 0 заменяются на 1, а при поиске 0 обнаруживается знак +. К этому моменту число слева от знака + равно a+b, поэтому машине остается только заменить знак + и все 1 справа от него на #.
- Состояние 5 — состояние остановки.

На с. 744 (внизу) приведена трассировка для простого примера. Анализ этой трассировки убедит вас, что машина действительно суммирует два произвольных



	действие	следующее состояние 0	1	#	запись 0	1	#	
0	R	0	0	1	0	1	#	*сканирование вправо до #*
1	L	2	1	2	1	0	1	*декремент*
2	L	2	2	2	0	1	#	*сканирование влево до +*
3	L	0	0	1	0	1	#	*инкремент*
4	R	2	1	2	1	0	1	*подготовка к завершению*
5	H	2	2	2	0	1	#	*остановка*

Машина Тьюринга для двоичного суммирования

двоичных числа. Эта машина также работает с двоичными числами любой длины; как и в предыдущем случае, впечатляет то, что такая простая машина способна выполнять такие вычисления.

состояние	лента	описание
0	# 1 0 1 + 1 1 #	начало
0	# 1 0 1 + 1 1 [#]	сканирование вправо до #
1	# 1 0 1 + 1 [0] #	декремент
2	# 1 0 1 [+] 1 0 #	сканирование влево до +
3	# 1 [1] 0 + 1 0 #	инкремент
0	# 1 1 0 + 1 0 [#]	сканирование вправо до #
1	# 1 1 0 + [0] 1 #	декремент
2	# 1 1 0 [+] 0 1 #	сканирование влево до +
3	# 1 1 [1] + 0 1 #	инкремент
0	# 1 1 1 + 0 1 [#]	сканирование вправо до #
1	# 1 1 1 + 0 [0] #	декремент
2	# 1 1 1 [+] 0 0 #	сканирование влево до +
3	# [1] 0 0 0 + 0 0 #	инкремент
0	# 1 0 0 0 + 0 0 [#]	сканирование вправо до #
1	# 1 0 0 0 [#] 1 1 #	декремент
4	# 1 0 0 0 # # # [#]	подготовка к завершению
	# 1 0 0 0 # # # #	остановка

Вычисление 5 + 3 = 8

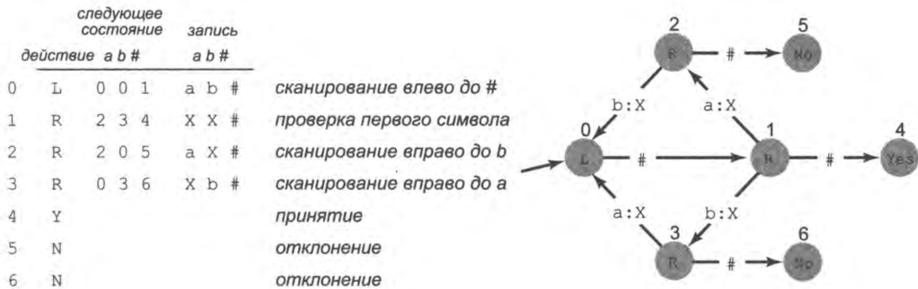
Эффективность

В этом и следующем разделах мы попробуем выяснить, какие вычисления возможны в границах модели машины Тьюринга; скорость таких вычислений нас не интересует (эта тема будет рассматриваться в разделе 5.5). Возможно, у вас уже возник вопрос — для чего нужна такая медленная машина? Человек вычисляет

суммы за время, пропорциональное их длине, а не значениям. Вообще говоря, для этой задачи можно построить машину Тьюринга с квадратичным временем (см. упражнение 5.2.22), но для текущего обсуждения эти факты несущественны.

Проверка равенства количества символов

В следующем примере выполняются вычисления, которые, как было доказано ранее, невозможны для ДКА: на вход подается двоичная строка, требуется решить, равны ли количества вхождений двух символов. Например, для строк `aabaabbbab` и `aaabbb` машина должна входить в состояние `Yes`, а для строк `aaa` и `aabbbab` — в состояние `No`. Машина Тьюринга, показанная на следующей диаграмме, выполняет эти вычисления.



	действие	следующее состояние a	b	#	запись a	b	#	
0	L	0	0	1	a	b	#	*сканирование влево до #*
1	R	2	3	4	X	X	#	*проверка первого символа*
2	R	2	0	5	a	X	#	*сканирование вправо до b*
3	R	0	3	6	X	b	#	*сканирование вправо до a*
4	Y							*принятие*
5	N							*отклонение*
6	N							*отклонение*

Машина Тьюринга для проверки равенства частот символов в двоичной строке

Чтобы понять, как работает машина, проанализируйте трассировку на с. 746. Машина находит крайний левый символ (`a` или `b`), заменяет его на `X` и ищет вхождение другого символа (`b` или `a`). Если совпадение не найдено, машина переходит в состояние `No`. В противном случае совпадающий символ заменяется на `X`, а машина возвращается к левой границе входных данных и ищет другую совпадающую пару. Каждый раз, когда машина входит в состояние 0, мы знаем, что она заменила равное количество `a` и `b` на `X`, поэтому если сканирование вправо в состоянии 1 не находит ни одного символа `a` или `b`, значит, исходная строка содержала равное количество `a` и `b`, поэтому она принимается. Как и для программ на Java, мы не будем приводить полное доказательство правильности; читатели со склонностью к математике могут попробовать сформулировать доказательство для такой машины Тьюринга.

Примеры машин Тьюринга также приведены в упражнениях в конце этого раздела; другие примеры будут рассмотрены позднее в этой главе. Как показывают упражнения, машины Тьюринга могут выполнять самые разнообразные вычислительные задачи, от умножения и деления до вычисления инструкций для построения кривой дракона. Построение машины Тьюринга, как и написание программы на Java, может быть интересным и захватывающим интеллектуальным занятием. В этих примерах и упражнениях мы стараемся убедить вас в том, что машина Тьюринга способна выполнить любые вычисления, которые только можно себе представить. Именно к этому удивительному выводу пришел Тьюринг в своей статье в 1937 году.

состояние	лента								описание
	#	a	a	b	b	b	a	b	# начало
0	[#]	a	a	b	b	b	a	b	# сканирование влево до #
1	#	[X]	a	b	b	b	a	b	# проверка первого символа
2	#	X	a	[X]	b	b	a	b	# сканирование вправо до b
0	[#]	X	a	X	b	b	a	b	# сканирование влево до #
1	#	X	[X]	X	b	b	a	b	# проверка первого символа
2	#	X	X	X	[X]	b	a	b	# сканирование вправо до b
0	[#]	X	X	X	X	b	a	b	# сканирование влево до #
1	#	X	X	X	X	[X]	a	b	# проверка первого символа
3	#	X	X	X	X	X	[X]	b	# сканирование вправо до b
0	[#]	X	X	X	X	X	X	b	# сканирование влево до #
1	#	X	X	X	X	X	X	[X]	# проверка первого символа
	#	X	X	X	X	X	X	X	[#] отклонение в состоянии 6

Трассировка принятия решения о равенстве

Как и в случае с ДКА, для проектировании и разработки машин Тьюринга той сложности, которая задействована в наших упражнениях, вам определенно понадобится программная среда, упрощающая отладку и трассировку. Впрочем, создать программу Java, моделирующую произвольную машину Тьюринга, не так уж сложно. И мы воспользуемся такой программой для тестирования машин Тьюринга и получения трассировки. Давайте рассмотрим код этой программы.

Универсальная виртуальная машина Тьюринга

В листинге 5.1.3 была приведена программа Java, предназначенная для анализа свойств ДКА, — для этого она моделирует их работу и выдает трассировку для разных вариантов входных данных. Теперь мы сделаем то же самое для машин Тьюринга.

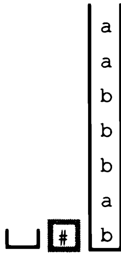
Лента

Первая проблема — как смоделировать бесконечную ленту? Возможно, вам будет нелегко найти решение самостоятельно, но ответ прост: нужно хранить текущий символ в переменной `current` типа `char` и использовать два стека, один для символов слева от каретки (в таком порядке, как если бы они заносились в стек слева направо), а другой для символов справа (в таком порядке, как если бы они заносились в стек справа налево). После того как вы разберетесь с этим представлением, дальнейшее особого труда не представляет. Например, как показано на левой нижней диаграмме на следующей странице, если каретка находится в любой позиции,

у левой границы входных данных

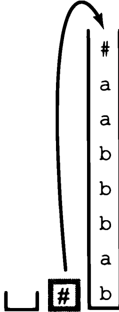
до

[#] a a b b b a b #



после

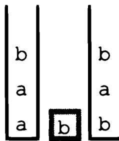
[#] # a a b b b a b #



в любой другой позиции

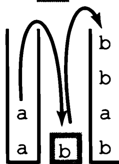
до

a a b [b] b a b



после

a a [b] b b a b

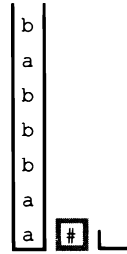


Перемещение влево на ленте

у правой границы входных данных

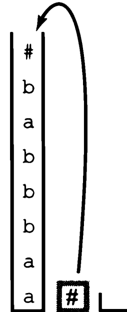
до

a a b b b a b [#]



после

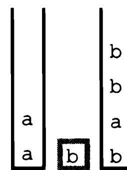
a a b b b a b # [#]



в любой другой позиции

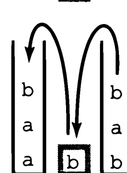
до

a a [b] b b a b



после

a a b [b] b a b



Перемещение вправо на ленте

кроме левой границы входных данных, мы перемещаем ее влево, просто занося current в правый стек и извлекая элемент из левого стека в current. Если каретка находится на левой границе входных данных, мы просто заносим # в правый стек, как показано на левой верхней диаграмме на предыдущей странице. Два случая для перемещения вправо реализуются аналогично. При такой структуре программы реализация в листинге 5.2.1 получается достаточно прозрачной. Этот код хорошо выражает абстракцию бесконечной ленты: чтобы предоставить клиенту иллюзию бесконечной ленты, мы просто «производим» новый символ # каждый раз, когда клиент пытается выйти за левую или правую границу данных, встречавшихся до настоящего момента. Лента на самом деле не бесконечна, но, с точки зрения клиента, ее длина не ограничивается. Конечно, стеки обладают тем же свойством; именно поэтому они так хорошо подходят для моделирования бесконечной ленты[1].

Машина

Программа TuringMachine (листинг 5.2.2) является клиентом класса Tape и реализует универсальную виртуальную машину Тьюринга, способную моделировать работу любой машины Тьюринга. По аналогии с классом DFA из листинга 5.1.3 она получает описание машины (из файла, имя которого передается как аргумент командной строки) и последовательность строк из стандартного потока ввода и выводит результат работы машины Тьюринга для заданных входных строк.

В файле хранится количество состояний, за ним следует алфавит и описания состояний. Каждая строка содержит один из вариантов L, R, H, Yes или No. За L или R следует индекс состояния для каждого символа в алфавите; i-й индекс определяет переход из этого состояния в том случае, если из позиции каретки был прочитан i-й символ алфавита. Далее указывается символ для каждого символа в алфавите, при этом i-й символ определяет заменяющий символ, если был прочитан i-й символ алфавита. После листинга 5.2.2 приведено описание машины Тьюринга для сумматора (файл addTM.txt).

Конструктор создает структуры данных, необходимые для хранения внутреннего представления машины Тьюринга, по данным в файле, переданном в аргументе командной строки. Для этого он:

- читает количество состояний и алфавит;
- создает массив строк для действий состояний, а для состояний L и R — два массива символических таблиц: для переходов между состояниями и для заменяющих символов;
- заполняет эти структуры данных, читая информацию каждого состояния из заданного файла.

---

[1] Для этой цели подошел бы также двусвязный список с возможностью произвольного добавления и изъятия элементов, но, несомненно, стек ближе к низкоуровневым реализациям. — *Примеч. науч. ред.*

Листинг 5.2.1. Класс ленты для виртуальной машины Тьюринга

```
public class Tape
{
   private Stack<Character> left  = new Stack<Character>();
   private Stack<Character> right = new Stack<Character>();
   private char current;

   public Tape(String input)
   {
      right.push('#');
      for (int i = input.length() - 1; i >= 0; i--)
         right.push(input.charAt(i));
      current = right.pop();
   }

   public char read()
   {  return current;  }

   public void write(char symbol)
   {  current = symbol;  }

   public void moveLeft()
   {
      right.push(current);
      if (left.isEmpty()) left.push('#');
      current = left.pop();
   }

   public void moveRight()
   {
      left.push(current);
      if (right.isEmpty()) right.push('#');
      current = right.pop();
   }

   public String toString()
   {  /* См. упражнение 5.2.7 */  }
}
```

Программа использует два стека для эмуляции бесконечной ленты, необходимой для работы машин Тьюринга. Символ, находящийся в позиции каретки, хранится в переменной current, символы слева от каретки хранятся в стеке left, а символы справа от каретки — в стеке right.

Код реализации конструктора достаточно прост:

```
public TuringMachine(String filename)
{
   In in = new In(filename);
   int n = in.readInt();
   String alphabet = in.readString();
   action = new String[n];
```

```
      next = (ST<Character, Integer>[])  new ST[n];
      out = (ST<Character, Character>[]) new ST[n];
      for (int st = 0; st < n; st++)
      {
         action[st] = in.readString();
         if (action[st].equals("Halt")) continue;
         if (action[st].equals("Yes"))  continue;
         if (action[st].equals("No"))   continue;

         next[st] = new ST<Character, Integer>();
         for (int i = 0; i < alphabet.length(); i++)
         {
            int state = in.readInt();
            next[st].put(alphabet.charAt(i), state);
         }

         out[st] = new ST<Character, Character>();
         for (int i = 0; i < alphabet.length(); i++)
         {
            char symbol = in.readString().charAt(0);
            out[st].put(alphabet.charAt(i), symbol);
         }
      }
   }
```

Другие методы листинга 5.1.3 тоже просты. Метод `simulate()` моделирует работу машины Тьюринга, а метод `main()` класса `TuringMachine` вызывает `simulate()` для каждой строки стандартного потока ввода.

Программа, приведенная в листинге 5.1.3, является одновременно как исчерпывающим руководством по построению машины Тьюринга, так и незаменимым инструментом для изучения свойств конкретных машин. Пользователь предоставляет алфавит, табличное описание машины Тьюринга и последовательность входных строк, а программа моделирует работу машины для каждой строки. Это еще один простой пример концепции виртуальной машины: не реального вычислительного устройства, а полного определения того, как такое устройство должно работать. Класс `TuringMachine` можно рассматривать как «компьютер», который вы «программируете», задавая множество вершин и переходов, которые подчиняются правилам для допустимых машин Тьюринга. Каждый экземпляр машины Тьюринга (файл с ее описанием) представляет собой «программу» для этого универсального компьютера.

Мы описываем новую реализацию практически теми же словами, что и реализацию универсального виртуального ДКА в разделе 5.1, но между ДКА и машинами Тьюринга существует принципиальное различие: *машина Тьюринга может не остановиться.* Если заданная машина Тьюринга входит в бесконечный цикл при заданных входных данных, то класс `TuringMachine` тоже зациклится. Более того, как вы узнаете в разделе 5.4, предотвратить зацикливание в принципе невозможно.

Модель машины Тьюринга чрезвычайно проста. Может, это просто игрушка? Вовсе нет! В главах 6 и 7 будет показано, что компьютер, на котором вы работаете, основан на вычислительной модели, которая ближе к модели машины Тьюринга,

Листинг 5.2.2. Универсальная виртуальная машина Тьюринга

```
public class TuringMachine
{
   private String[] action;
   private ST<Character, Integer>[] next;
   private ST<Character, Character>[] out;

   public TuringMachine(String filename)
   {  /* См. текст. */  }

   public String simulate(String input)
   {
      Tape tape = new Tape(input);
      int state = 0;
      while (action[state].equals("L")
          || action[state].equals("R"))
      {
         if (action[state].equals("R")) tape.moveRight();
         if (action[state].equals("L")) tape.moveLeft();
         char c = tape.read();
         tape.write(out[state].get(c));
         state = next[state].get(c);
      }
      return action[state] + " " + tape;
   }

   public static void main(String[] args)
   {
      TuringMachine tm = new TuringMachine(args[0]);
      while (StdIn.hasNextLine())
         StdOut.println(tm.simulate(StdIn.readLine()));
   }
}
```

Программа моделирует работу машины Тьюринга, заданной первым аргументом командной строки, для каждой строки из стандартного потока ввода. При остановке машины Тьюринга программа выводит сообщение Yes, No или Halt, за которым следует содержимое ленты. Также возможен переход программы в бесконечный цикл.

```
% more addTM.txt
6 01+#
R  0 0 0 1  0 1 + #
L  1 2 4 1  1 0 # #
L  2 2 3 2  0 1 + #
L  0 3 3 0  1 0 + 1
R  4 4 4 5  # # # #
Halt
```

```
% java TuringMachine addTM.txt
101+11
Halt 1000
10000000011011+11000001
Halt 10000011011100
```

чем к знакомой вам среде Java. А еще важнее то, что модель машины Тьюринга позволяет получить ответы на ряд принципиальных вопросов о природе вычислений. Этой темой мы сейчас и займемся.

Вопросы и ответы

В. Где я могу больше узнать о машинах Тьюринга?

О. Эта тема рассматривается во многих, многих книгах. В тексте уже упоминался классический труд М. Мински *Computation: Finite and Ininite Machines*. Более современная точка зрения представлена в книгах *Introduction to the Theory of Computation* Майкла Сипсера (Michael Sipser) и *Computers Ltd.: What They Really Can't Do* Дэвида Харела (David Harel).

В. Где я могу больше узнать об Алане Тьюринге?

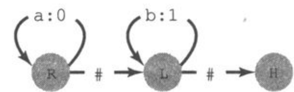
О. Существует несколько достойных биографий, в которых описана история жизни и наследие Алана Тьюринга, в том числе *Alan Turing: The Enigma* Эндрю Ходжеса (Andrew Hodges). Известный фильм «Игра в имитацию» с Бенедиктом Камбербэтчем поставлен по мотивам книги Ходжеса.

В. Так ли необходимо уметь строить машины Тьюринга с нуля?

О. Как и многие другие навыки, этот осваивается на практике. Нет, вряд ли вам придется создавать машины Тьюринга на вашей будущей работе. Но если вы будете понимать базовые свойства машины Тьюринга, описанные в этом разделе, вам будет намного проще понять удивительные вычислительные концепции, которыми мы займемся позднее. И бесспорно, разработка машины Тьюринга — интеллектуальное упражнение, которое обогащает ум. Выполнять все упражнения подряд не обязательно — выбирайте те, которые вам интересны (или те, которые вам назначил преподаватель!).

Упражнения

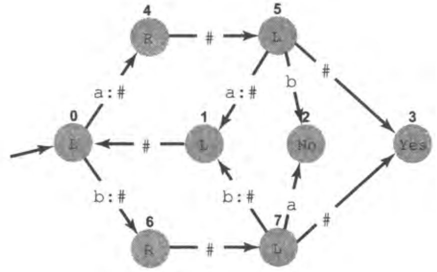
5.2.1. Приведите трассировку работы машины Тьюринга на диаграмме справа для исходных данных aabaabaabb. Затем приведите краткое неформальное описание вычислений, выполняемых машиной.



5.2.2. Загрузите файлы TuringMachine.java и Tape.java с сайта книги. Создайте текстовые файлы incrementerTM.txt (инкремент) и addTM.txt (сумматор), запустите эти машины для разных вариантов входных данных и убедитесь в том, что машины работают в соответствии с описанием.

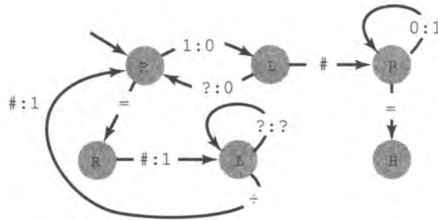
5.2.3. Создайте текстовый файл с табличным представлением машины Тьюринга, описанной в упражнении 5.2.1. Загрузите файлы TuringMachine.java и Tape.java с сайта книги и протестируйте свое решение, запустив машину для разных вариантов входных данных.

5.2.4. Приведите трассировку работы машины Тьюринга на диаграмме справа для исходных данных aabaabaa. Затем приведите краткое неформальное описание вычислений, выполняемых машиной.



5.2.5. Создайте текстовый файл с табличным представлением машины Тьюринга, изображенной на диаграмме справа. Загрузите файлы TuringMachine.java и Tape.java с сайта книги и протестируйте свое решение упражнения 5.2.4, запустив машину для разных вариантов входных данных.

5.2.6. Машина Тьюринга, изображенная на диаграмме внизу, должна выполнять унарное деление — для двух чисел, выраженных в унарной (унитарной, единичной) системе счисления: для двух чисел, записанных в унарной системе и разделенных символом ÷, она должна выводить на ленту знак равенства и результат деления после двух исходных строк. Например, при вводе **1111111111÷11** результат на ленте должен иметь вид **1111111111÷11=11111**.



Чтобы реализовать эту функциональность, замените символы **?** на ленте нужными символами.

5.2.7. Добавьте в класс `Tape` код реализации метода `toString()`. Чтобы выводимый результат соответствовал результатам программы 5.2.2, блокируйте метасимвол **#**.

5.2.8. Добавьте в класс `TuringMachine` код, который выводит строку с именем состояния, позицией каретки и содержимым ленты при каждом изменении состояния машины. Измените метод `toString()` в классе `Tape`, чтобы он оставлял метасимвол **#** и помечал позицию каретки, выводя пару квадратных скобок до и после символа в позиции каретки.

В каждом из следующих упражнений вам предлагается «спроектировать машину Тьюринга». Правильное решение состоит из диаграммы в стиле, приведенном в тексте, текстового файла с табличным представлением и трассировки для типичных

входных данных, созданной классом TuringMachine и измененной в соответствии с упражнением 5.2.8.

5.2.9. Спроектируйте машину Тьюринга для разрешения[1] языка, состоящего из всех строк с равным количеством символов A, B и C.

5.2.10. Спроектируйте машину Тьюринга для операции двоичного декремента, которая является полноценной обратной операцией по отношению к нашему двоичному инкременту (то есть удаляет начальные 0). Например, результат применения декремента к **#10000#** должен быть равен **#0111#**.

5.2.11. Спроектируйте машину Тьюринга для разрешения языка, состоящего из всех вариантов входных данных, у которых количество символов равно степени 2.

5.2.12. Спроектируйте машину Тьюринга для разрешения языка, состоящего из всех двоичных строк нечетной длины, у которых средним символом является символ |.

5.2.13. Спроектируйте машину Тьюринга для разрешения языка, состоящего из строк, в которых за n символами a следуют n символов b для некоторого целого n.

5.2.14. Спроектируйте машину Тьюринга для разрешения языка, состоящего из всех двоичных палиндромов.

5.2.15. Спроектируйте машину Тьюринга для разрешения языка, состоящего из всех строк круглых скобок, имеющих корректную парную структуру: (), ()(), (()()), ((()(()))) и т. д.

5.2.16. Спроектируйте машину Тьюринга для разрешения языка, состоящего из двух идентичных десятичных чисел, разделенных символом |.

5.2.17. Спроектируйте машину Тьюринга, которая создает вторую копию своих входных данных. Например, если лента инициализирована строкой abcd, на ленте должна остаться строка abcd#abcd.

5.2.18. Спроектируйте машину Тьюринга, которая получает две унарные строки, разделенные символом x, перемножает их и записывает на ленту после двух строк знак равенства и результат. Например, для входных данных **11x11111** на ленте должен остаться результат **11x11111=1111111111**.

5.2.19. Спроектируйте машину Тьюринга, которая ведет отсчет в двоичной системе. В исходном состоянии лента должна быть пустой. В процессе работы машины на ней должны появляться строки **1**, **10**, **11**, **100**, **101**, **110** и т. д. без остановки.

5.2.20. Спроектируйте машину Тьюринга, которая получает две двоичные строки равной длины, разделенные символом ^, и оставляет на ленте результат операции поразрядного исключающего ИЛИ с двумя строками.

---

[1] Разрешение языка — способность машины корректно обрабатывать программы на этом языке (не только корректные). Подробнее о разрешимости и выполнимости языка см. раздел 5.3. — *Примеч. науч. ред.*

5.2.21. Спроектируйте машину Тьюринга, которая получает двоичное целое число с начальным битом 1 и оставляет на ленте двоичное представление разрядности этого целого числа. Эта функция называется *дискретным двоичным логарифмом.*

Упражнения повышенной сложности

5.2.22. *Эффективный сумматор.* Спроектируйте машину Тьюринга наподобие описанной в тексте; машина получает две двоичные строки, разделенные знаком +, интерпретирует их как двоичные числа и складывает, оставляя результат на ленте. В отличие от машины Тьюринга в тексте, ваша машина должна выполняться за время, ограниченное полиномиальной зависимостью от длины чисел, а не их значений.

5.2.23. *Эффективное сравнение.* Спроектируйте машину Тьюринга наподобие описанной в тексте; машина получает две двоичные строки, разделенные знаком ?, интерпретирует их как двоичные числа и переходит в состояние Yes, если первое число меньше второго, и в состояние No в противном случае. Убедитесь в том, что машина выполняется за время, ограниченное полиномиальной зависимостью от длины чисел, а не их значений.

5.2.24. *Кривая дракона.* Спроектируйте машину Тьюринга, которая при запуске с входной лентой, содержащей последовательность из $2^n - 1$ нулей, оставляет на ленте инструкции для построения кривой дракона порядка n (см. упражнение 1.2.35). Используйте следующий алгоритм: заменить каждый второй 0 попеременно на L или R; повторить.

5.2.25. *Машина Тьюринга для функции Коллаца.* Спроектируйте машину Тьюринга, которая получает на входе двоичное представление целого числа и многократно делит его на 2 (для четного числа) или умножает на 3 и прибавляет 1 (для нечетного числа), пока результат не будет равен 1. Эта функция называется *функцией Коллаца*, а вопрос о том, завершится ли работа машины для всех вариантов входных данных, до сих пор остается открытым (см. упражнение 2.3.29).

5.3. Универсальность

Модель машины Тьюринга — математический артефакт, который упрощает базовые концепции, связанные с вычислениями, до такой степени, что о них можно сделать точные математические утверждения. База теории была в основном заложена Тьюрингом в исходной статье, однако исследователи продолжают работать над развитием нашего понимания этих идей. Такие утверждения позволяют сделать глубокие выводы, которые оказывают непосредственное влияние на окружающие нас вычислительные инфраструктуры. Из-за колоссальных масштабов своего влияния они должны быть понятны любому специалисту, серьезно занимающемуся вычислениями (в том числе и каждому читателю этой книги). В этом разделе мы

исследуем две фундаментальные идеи, представленные в исходной статье Тьюринга: концепция того, что каждый отдельно взятый компьютер общего назначения может выполнить любое вычисление, и гипотеза о том, что любые вычислительные устройства на фундаментальном уровне функционально эквивалентны.

Алгоритмы

Работая над компьютерной программой, мы обычно реализуем метод, предназначенный для решения некоторой задачи. Этот метод не зависит от используемой среды программирования — скорее всего, он будет подходить для использования во многих средах. Действия, выполняемые для решения задачи, определяются методом, а не компьютерной программой. Конечный, детерминированный и эффективный (результативный) метод решения задачи, который может быть реализован в виде компьютерной программы, называется *алгоритмом*. Изучением алгоритмов занимается информатика; они являются центральным объектом изучения в этой области.

В книге мы уже использовали этот термин и рассматривали многочисленные примеры алгоритмов, от метода Ньютона и алгоритма Евклида до сортировки слиянием и поиска в ширину. И хотя приведенное определение является неформальным, оно хорошо описывает концепцию. Модель машины Тьюринга позволяет математикам разработать формальное определение, которое может быть использовано в математических доказательствах. Как обычно, мы воздержимся от тщательного, формального изложения определений и доказательств, но приведем обзор важнейших концепций.

Разрешимость

Допустим, вы поместили на ленту входную строку и запустили машину Тьюринга из ее исходного состояния. Существуют четыре возможных результата; машина может:

- остановиться в состоянии с пометкой `Yes` (принять входную строку);
- остановиться в состоянии с пометкой `No` (отклонить входную строку);
- остановиться в состоянии с пометкой `H` (остановиться без принятия или отклонения);
- не выбрать ни один из этих вариантов (войти в бесконечный цикл).

Временно забудем о содержимом ленты на момент остановки машины (или во время зацикливания). Мы говорим, что машина Тьюринга распознает язык, состоящий из всех входных строк, переводящих машину в состояние принятия. Если вдобавок машина Тьюринга остановится (в состоянии с пометкой `No` или `H`) для всех входных строк, не входящих в распознаваемый язык, мы говорим, что машина Тьюринга *разрешает* этот язык (принимает решение о входных данных). Машина должна всегда давать правильный ответ и всегда останавливаться для каждого варианта входных

данных. Например, наша программа для проверки равенства числа появлений символов в конце предыдущего раздела разрешает (и распознает) язык всех двоичных строк с равным количеством символов a и b. *Примечание*: с ДКА различать разрешение и распознавание было не нужно, так как ДКА останавливается всегда.

Вычислимость

С учетом вывода результатов на ленту мы можем расширить концепцию машины Тьюринга и разрешения до концепции вычисления функции. Например, машина Тьюринга из упражнения 5.2.21 вычисляет функцию дискретного двоичного логарифма. Функция $f(x)$ называется *вычислимой*, если существует некоторая машина Тьюринга, которая при инициализации ленты значением x оставляет на ленте строку $f(x)$. Целые числа могут представляться двоичными строками или цепочками десятичных цифр, или для них может использоваться любое другое разумное представление. Все обычные операции с целыми числами (инкремент, сложение, вычитание, умножение, целочисленное деление, возведение в степень) являются вычислимыми функциями.

Понятия разрешимости и вычислимости помогают точно сформулировать концепцию того, что мы подразумеваем под термином «алгоритм»; машина Тьюринга, которая разрешает некоторый язык или вычисляет некоторую функцию, представляет алгоритм для этой задачи. У любой задачи может быть много разных машин Тьюринга, поэтому часто мы рассматриваем несколько разных алгоритмов для решения задачи программирования. В теории вычислений термины «*машина Тьюринга*» и «*алгоритм*» могут считаться синонимами, что позволяет нам делать общие утверждения обо «всех алгоритмах» или обо «всех алгоритмах для конкретной задачи» с математической точностью.

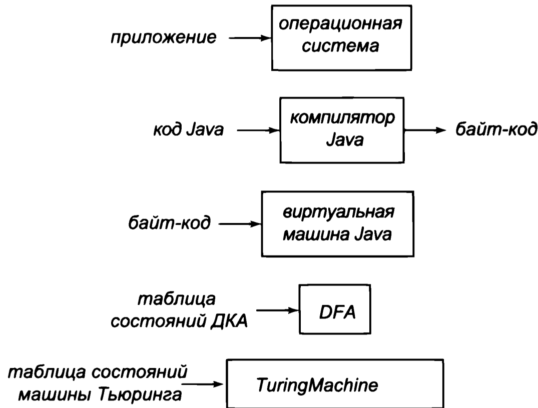
Программы, обрабатывающие программы

Чтобы подвести вас к правильному образу мыслей для последующих теоретических рассуждений, мы ненадолго отвлечемся для обсуждения концепции программы, которая получает на вход другую программу (и возможно, выдает на выходе еще одну программу). После некоторых размышлений вы наверняка осознаете, что вы регулярно используете программы, которые обрабатывают программы. Например, каждое приложение на вашем мобильном устройстве является программой: загружая новую программу, вы используете программу на хосте, которая получает вашу программу на входе и упаковывает ее для отправки, после чего другая программа на вашем устройстве получает эту программу и устанавливает ее, а третья программа (операционная система) запускает приложение.

Среда разработки Java

Подробности будут приведены в разделе 6.4, но в разделе 1.1 мы уже описывали два примера программ, обрабатывающих программы, из среды разработки Java. С этого момента вы неоднократно пользовались этими программами. *Компилятор* Java

переводит код Java на другой язык программирования, называемый байт-кодом Java, а (универсальная) виртуальная машина Java (JVM, Java Virtual Machine) получает любую программу на байт-коде и запускает ее на выполнение, для чего в конечном итоге используется еще один язык программирования — машинный язык вашего компьютера. В действительности JVM также является программой, написанной на языке программирования C — более старом и низкоуровневом по сравнению с Java, но доступном на большинстве компьютеров.



Программы, которые обрабатывают программы

Универсальный виртуальный ДКА (реализация на Java)

Наша программа DFA (листинг 5.1.3) получает описание ДКА в виде таблицы состояний, после чего моделирует работу машины. Как обсуждалось в конце раздела 5.1, правила составления таблицы состояний можно рассматривать как язык программирования, а каждое описание таблицы состояний ДКА — как программу, написанную на этом языке. Таким образом, DFA представляет собой программу на языке Java, которая получает на вход «программу» (описание конкретного ДКА). Кроме того, реализация процедуры преобразования произвольного НКА в ДКА, описанной в конце раздела 5.1 (см. упражнение 5.1.39), также требует программы на языке Java, которая получает программу (таблицу переходов НКА) и выдает программу (таблицу состояний ДКА).

Универсальный виртуальный ДКА (реализация в виде машины Тьюринга)

Конечно, не каждая программа нуждается в Java для своей реализации! В самом деле, можно даже построить машину Тьюринга, которая обрабатывает другую программу. Например, машина Тьюринга из упражнения 5.3.2 моделирует работу любого ДКА с тремя состояниями, который использует двоичный алфавит (архитектура элементарно расширяется для поддержки большего количества состояний и больших

алфавитов). На вход (на ленту) подается таблица состояний для произвольного ДКА с тремя состояниями и двоичная входная строка для этого ДКА, машина моделирует работу ДКА с полученными данными и завершает ее в состоянии `Yes`, если входная строка входит в язык, принимаемый ДКА, и в состоянии `No` в противном случае (см. упражнение 5.3.4). Получается, что программа на языке Java (`TuringMachine`) получает на вход другую программу (таблицу состояний машины Тьюринга), которая получает на входе третью программу (таблицу состояний ДКА)!

Подведем итог: программы, обрабатывающие другие программы, имеют фундаментальное значение в информатике, и ничто не мешает нам создавать машины Тьюринга для обработки программ. А самое поразительное, что Тьюринг осознал последствия тех фактов, о которых мы сейчас расскажем, еще до появления компьютеров!

Универсальная машина Тьюринга

Итак, мы можем создать машину Тьюринга, моделирующую работу произвольного ДКА для любых заданных входных данных. А можно ли создать машину Тьюринга, которая моделирует работу произвольной машины Тьюринга для любых заданных входных данных? Ответ на этот вопрос — да и еще раз ДА! Это был самый фундаментальный вывод из статьи Тьюринга. Такая машина называется *универсальной машиной Тьюринга*.

В книге мы не станем разрабатывать полную реализацию универсальной машины Тьюринга, но после изучения упражнения 5.3.2 вы поймете, как следует подходить к решению этой задачи.

- Расширьте формат ввода для включения в него символов замены.
- Разработайте аналог `Tape` (листинг 5.2.2) для эмуляции ленты моделируемой машины.
- Добавьте операции с лентой в машину Тьюринга для универсального виртуального ДКА из упражнения 5.3.2.
- Добавьте механизм отслеживания текущего состояния (вместо повторения «кодов» состояний, для чего необходимо знать количество состояний).

В текущем контексте мы не предлагаем прорабатывать все детали (даже исходная универсальная машина Тьюринга, созданная самим Тьюрингом, содержала ошибки!). Просто убедитесь в том, что построение такой машины — вполне реальное дело, хотя и выходящее за рамки книги. А когда вы поймете, что универсальная машина Тьюринга существует, вы будете готовы к пониманию тех выводов, о которых мы расскажем далее.

Если вам *захочется* изучить все подробности, на нашем сайте книги имеется реализация универсальной машины Тьюринга с 24 состояниями и 7 символами, а также графическая виртуальная универсальная машина Тьюринга, обеспечивающая визуальную трассировку ее работы.

Компьютеры общего назначения

Так как на вашем компьютере установлена среда Java и на нем можно выполнить `TuringMachine`, ваш компьютер является универсальной виртуальной машиной Тьюринга: он может выполнять разные алгоритмы, не требуя изменений аппаратной части. Подробности будут приведены в главах 6 и 7, но стоит заметить, что это стало возможно благодаря тому, что современные процессоры базируются на *архитектуре фон Неймана*, при которой компьютерные программы и данные хранятся в общей основной памяти. Это означает, что в зависимости от контекста содержимое памяти может интерпретироваться и как машинные команды, и как данные. Такая структура полностью аналогична содержимому ленты универсальной машины Тьюринга, которое состоит как из программы (моделируемая машина Тьюринга), так и из ее данных (содержимое ленты для моделируемой машины).

Таким образом, работа Тьюринга над универсальной машиной Тьюринга предвосхитила разработку компьютеров общего назначения и может с полным основанием рассматриваться как изобретение программного обеспечения! Вместо того чтобы проектировать разные машины для разных задач, вы проектируете одну машину и программируете ее для выполнения разнообразных задач. Например, на устройстве, используемом для анализа экспериментальных данных, можно подготавливать тексты статей; обрабатывать графику, музыку и фильмы; заниматься веб-серфингом; общаться в социальных сетях и играть в шахматы.

Тезис Чёрча — Тьюринга

Тьюринг был глубоко убежден, что его модель воплощает концепцию вычислений, выполняемых условным математиком по четко определенной процедуре[1]. Приблизительно в то же время (а на самом деле чуть ранее) Алонзо Чёрч (Alonso Church) изобрел совершенно иную математическую модель для изучения понятия вычислимости. Как оказалось, модель Чёрча, называемая *лямбда-исчислением*, вела прямиком к разработке современных функциональных языков программирования. В отношении вычислений модели кажутся совершенно различными, но Тьюринг позднее продемонстрировал их эквивалентность в том отношении, что они характеризуют полностью совпадающие классы математических функций. И хотя исследования были чисто теоретическими, эта эквивалентность привела Чёрча и Тьюринга к одному выводу относительно вычислений. Внимание Тьюринга было сосредоточено на условном математике, вычисляющем значения функций для целых чисел, а подход Чёрча был чисто функциональным, но со временем они и другие ученые осознали, что их идеи сообщают нечто важное о реальном мире: что все физически реализуемые вычислительные устройства могут быть смоделированы машиной Тьюринга. Таким образом, изучение вычислений в реальном мире сводится к изучению машин Тьюринга вместо бесконечного набора потенциально

[1] Такая модель (или *парадигма*) вычислений получила название *императивной*. Она изучена наиболее подробно и обычно более проста для понимания. — *Примеч. науч. ред.*

возможных вычислительных устройств. В более формальном виде концепция универсальной машины Тьюринга используется для выражения следующей идеи.

Тезис Чёрча — Тьюринга. Универсальная машина Тьюринга может выполнить любые вычисления (разрешение языка или вычисление функции), которые могут быть выполнены любым физически реализуемым вычислительным устройством.

Комментарий: в форме, рассматривавшейся Тьюрингом и Чёрчем, тезис относился к умозрительным свойствам вычислений, выполняемых «условным математиком» по четко определенной процедуре. В сильной форме, представленной здесь, тезис содержит утверждение о законах природы и о том, какие вычисления могут выполняться в нашем мироздании.

Тезис Чёрча — Тьюринга не является математической теоремой и не имеет формального доказательства. Дело в том, что мы не можем дать математическое определение того, что подразумевается под «физически реализуемым вычислительным устройством».

Как будет показано далее, за годы, прошедшие с момента первоначальной формулировки тезиса, был накоплен огромный объем свидетельств, указывающих на его истинность. Тем не менее теоретически тезис может быть опровергнут (то есть приведено доказательство его ложности). Если кто-то обнаружит более мощную модель вычислений, имеющую физическую реализацию, нам придется отказаться от тезиса или дополнить его.

Контрапозиция тезиса тоже представляет интерес. Она гласит, что если какие-то вычисления не могут быть выполнены на машине Тьюринга, они не могут быть выполнены ни на каком ином физически реализуемом вычислительном устройстве. Как будет показано в разделе 5.4, существуют языки, которые машина Тьюринга разрешить не может. Следовательно, если считать тезис Чёрча — Тьюринга истинным, то из этого следует, что для вычислений, которые могут быть выполнены на физически реализуемых вычислительных устройствах, существуют ограничения.

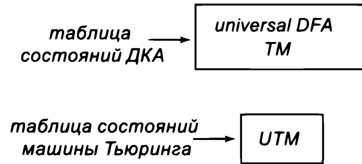
Разновидности модели машины Тьюринга

Одно из доказательств в пользу тезиса Чёрча — Тьюринга связано с анализом последствий различных изменений в модели машины Тьюринга. В одном направлении исследователи изучали вопрос о том, приведет ли добавление различных механизмов в модель к появлению машин, способных распознавать некоторые языки, которые не распознаются машиной Тьюринга, или вычислять функции, которые ею не вычисляются; в другом направлении исследователи изучали, до какой степени можно упростить модель без ее ослабления. Учтите, что в текущем контексте нас не интересует, сколько состояний или символов может использовать машина или сколько переходов между состояниями может потребоваться для вычисления, — нас интересует *только* вычислительная мощь, определяемая множеством распознаваемых языков или множеством вычислимых функций.

Эквивалентные модели

Различные авторы обычно используют незначительные модификации исходной модели Тьюринга (вроде модели, которую рассматривали мы, — ее разработал Мински) без дополнительных комментариев, если преобразование между моделями не требует пояснений. Некоторые решения принимаются исключительно для удобства. Мы считаем, что наша версия приводит к простым компактным схемам, максимально доступным для понимания. Авторы учебника более высокого уровня по теории вычислений могут предпочесть другую версию, которая упрощает доказательства. Например, перемещения каретки обычно связываются с каждым переходом, а не с целевым состоянием. Другая эквивалентная модель, упоминавшаяся в разделе 5.2, — стековый автомат с двумя стеками; наше использование двух стеков в классе `TuringMachine` подтверждает этот интуитивно понятный факт.

Расширения



Еще две программы, которые обрабатывают программы

Ученые проанализировали многочисленные возможные усовершенствования модели машины Тьюринга. Например, станет ли модель машины Тьюринга более мощной, если добавить еще одну ленту с независимой кареткой? Ответ на этот вопрос отрицательный, потому что такую машину можно моделировать на базе машины Тьюринга, используя нечетные позиции для одной ленты и четные для другой. Другой пример — включение недетерминированности. Вычислительные возможности при этом не возрастают, потому что можно построить детерминированную машину Тьюринга, которая распознает тот же язык или вычисляет ту же функцию, что и любая недетерминированная машина Тьюринга, аналогично тому, как это было сделано для НКА. Другие примеры показаны в верхней части таблицы на следующей странице. Мы опустим дальнейшие обсуждения и доказательства, но заметим, что за десятки лет работ *никому не удалось найти более мощную физически реализуемую модель.*

Ограничения

Исследователи также искали возможность упростить модель машины Тьюринга. Некоторые примеры приведены в нижней части таблицы на следующей странице. Например, использование ленты, бесконечной только в одном направлении, ограничением не является, потому что мы можем использовать нечетные позиции для одного направления и четные позиции для другого. Или другой пример: использование двоичного алфавита также не является ограничением, потому что символы на ленте могут быть закодированы в двоичном виде. Все подробности, дальнейшие обсуждения и доказательства снова опускаются. Конечно, у ограничений есть свои пределы: как вы уже видели, модель ДКА является однозначно менее мощной, чем модель машины Тьюринга, — как и модель машины Тьюринга,

которая не может записывать символы на ленту, или машина Тьюринга с кареткой, способной перемещаться только в одном направлении. Задача поиска простейшей модели машины Тьюринга привлекала многих исследователей; вскоре мы ненадолго вернемся к этой теме.

модификация машины Тьюринга	описание
эквивалентные модели	
перемещения при переходах	Перемещения L и R связываются с каждым переходом, а не с целевым состоянием
стековый автомат с двумя стеками	Стековый автомат с двумя стеками
расширения машины Тьюринга	
многоленточная	Добавление конечного числа независимых лент с независимыми каретками
многомерная	Использование многомерной ленты (каретка может перемещаться в любом направлении)
недетерминированная	Возможность нескольких переходов для каждого входного символа
вероятностная	Переходы выбираются случайным образом (принимается, если большинство исходов ведет в состояние принятия)
забывчивая	Переходы не зависят от входных данных
ограничения	
односторонне-бесконечная	Лента бесконечна только в одном направлении
двоичная	Разрешены только два символа
с двумя состояниями	Разрешены только два состояния
без стирания	Нет возможности перезаписи
последовательная	Состояния циклически упорядочены (переходы из текущего состояния разрешены только в следующее за ним)

Разновидности модели машины Тьюринга, не влияющие на ее вычислительную мощь

О разновидностях модели Тьюринга, распознающих то же множество языков и вычисляющих то же множество функций, написаны сотни, если не тысячи, статей и книг. Этот факт однозначно показывает, что модель по крайней мере является поворотной точкой для нашего понимания вычислений — шаг от стековой машины с одним стеком к стековой машине с двумя стеками, эквивалентной машине Тьюринга, является воистину огромным.

Универсальные модели

Модель называется *полной по Тьюрингу* или *универсальной по Тьюрингу*, если она эквивалентна модели машины Тьюринга (то есть она распознает то же множество языков и вычисляет то же множество функций.) Тезис Чёрча — Тьюринга предполагает, что машина Тьюринга является фундаментальным объектом окружающей природы. Существуют ли другие естественные модели вычислений, которые могут выполнить любую программу для любых входных данных, как машина Тьюринга? И снова ответом будет да и еще раз ДА! Около века математики, специалисты по информатике, физики, лингвисты или биологи рассматривали многочисленные модели вычислений, которые, как было показано, обладают полнотой по Тьюрингу. Многие из них перечислены в таблице на с. 766, а отдельные представители кратко упоминаются ниже.

Лямбда-исчисление

Как упоминалось ранее, в то же время, когда Тьюринг готовил к печати свою статью в Принстонском университете, Чёрч завершал свою работу о *лямбда-исчислении* — формальной системе, которая является основой для современного функционального программирования. Понимание того, что лямбда-исчисление Чёрча и модель машины Тьюринга являются эквивалентными, привело к формулированию тезиса Чёрча — Тьюринга.

Счетчики

Еще более простая по сравнению с машиной Тьюринга модель была популяризирована Мински — это машина-*счетчик*, у которой лента заменяется небольшим множеством счетчиков, способных хранить произвольное целое число, а состояния и переходы заменяются последовательностью инструкций из набора операций типа «инкремент», «декремент» и «переход по нулю».

Клеточные автоматы

Возможно, вам знакома игра «Жизнь» — вычислительная модель, разработанная математиком Джоном Конуэем (за подробностями обращайтесь к упражнениям 2.4.20 и 5.3.19). Эта игра является примером *клеточного автомата* — дискретной системы, в которой клетки взаимодействуют со своими соседями. Изучение клеточных автоматов было начато в 1940 году Джоном фон Нейманом, важной фигурой в истории информатики (см. главу 6).

Ваш компьютер

Как вы уже заметили, программа `TuringMachine` (листинг 5.2.2) доказывает, что ваш компьютер как минимум не уступает по мощи машине Тьюринга, потому что он может моделировать поведение любой машины Тьюринга. Но, возможно, вас удивит другой факт: существует машина Тьюринга, способная моделировать работу вашего компьютера. Как будет показано в главах 6 и 7, ваш компьютер построен на

базе модели машины с инструкциями, работающими с двоичными числами, которая намного ближе к машине Тьюринга, чем к знакомой вам среде Java. Разработать машину Тьюринга, которая может моделировать такую машину, на концептуальном уровне несложно. Следовательно, если вы можете разработать на своем компьютере программу, которая может разрешить некоторый язык или вычислить некоторую функцию, то существует машина Тьюринга, которая может выполнить ту же задачу (также это может сделать и любая универсальная машина Тьюринга).

Языки программирования

Почти каждый язык программирования, используемый в наши дни, является полным по Тьюрингу. К этой категории относятся процедурные языки (такие, как C, Fortran и Basic), объектно-ориентированные языки (такие, как Java и Smalltalk), функциональные языки (такие, как Lisp и Haskell), мультипарадигменные языки (такие, как C++ и Python), специализированные языки (такие, как Matlab и R) и логические языки (такие, как Prolog). И хотя некоторые языки программирования могут показаться более мощными, чем другие, по своей сути они эквивалентны (в отношении того, какие функции они могут вычислить и какие языки могут разрешить.) Языки программирования отличаются по другим важным характеристикам (удобству сопровождения, переносимости, доступности, надежности и эффективности), поэтому при выборе языка мы руководствуемся удобством и эффективностью, а не потенциальными возможностями.

Системы замены строк

Многие модели вычислений предполагают создание наборов правил для замены подстрок в множествах строк. Подобные модели (системы) бывают достаточно просты, чем и объясняется их популярность. Некоторые примеры таких систем рассмотрены в упражнениях повышенной сложности в конце раздела.

Компьютеры и ДНК

Современная молекулярная биология помогла понять дискретные изменения в ДНК, управляющие биологическими процессами. В 1994 году Леонард Адельман (Leonard Adelman) выдвинул идею использования этих изменений для проведения вычислений и моделирования машины Тьюринга. Лабораторные эксперименты подтвердили обоснованность этого подхода: на базе ДНК можно создать компьютер! Конечно, вопрос о том, ведет ли себя подобным образом какой-нибудь биологический процесс, — совсем другое дело.

Концепция универсальности ведет к радикальному сдвигу точки зрения. По сути, утверждается, что любой феномен, наблюдаемый в реальном мире и похожий на компьютер, действительно *является* компьютером. Несмотря на многолетние попытки, было доказано, что каждая разумная модель вычислений, которая была разработана со времен Тьюринга и по меньшей мере не уступает по мощи машине Тьюринга, может моделировать машину Тьюринга. Все адекватные модели вычислений, созданные за десятилетия интенсивных работ со времен Тьюринга

и по крайней мере не уступающие по мощи машине Тьюринга (то есть могут ее моделировать), также и не превосходят машину Тьюринга (потому что могут быть смоделированы на машине Тьюринга).

модель	описание
начало XX века	
полусистемы Туэ (Туэ, 1910)	Правила замены строк, применяемые в произвольном порядке
формальные системы Поста (Пост, 1920)	Правила замены строк, спроектированные для доказательства математических утверждений по набору аксиом
середина XX века	
лямбда-исчисление (Чёрч, 1936)	Метод определения и выполнения операций с функциями (основа функциональных языков программирования, таких как Lisp и ML)
Машина Тьюринга (Тьюринг, 1936)	Конечный автомат, который читает и записывает данные на бесконечную ленту
Машина Поста (Пост, 1936)	Машина Тьюринга с одной очередью
Рекурсивные функции (Гёдель и др., 1930-е)	Функции, определенные для вычислений с натуральными числами
Неограниченные грамматики (Хомский, 1950)	Правила замены строк, предназначенные для описания естественных языков
Двумерные клеточные автоматы (фон Нейман, 1952)	Двумерный массив (матрица) двоичных значений, которые изменяются в соответствии с заданными правилами и в зависимости от значений их соседей
Системы Маркова (Марков, 1960)	Правила замены строк, применяемые в заранее определенном порядке
Логика дизъюнктов Хорна (Horn, 1961)	Система доказательства теорем на базе логики (основа языка программирования Prolog)
Двухрегистровый ДКА (Мински, 1961)	ДКА с двумя счетчиками (каждый счетчик хранит целое число, к которому машина может применять операции инкремента, декремента и проверки на нуль)
Двухстековая машина Тьюринга (Шеферсон/Стерджес, 1963)	Машина Тьюринга с двумя стеками
Игра «Жизнь» (Конуэй, 1960-е)	Специализированный двумерный клеточный автомат
Машина с указателем	Конечный набор регистров и память, с которой можно работать как со связным списком
Машина с произвольным доступом	Конечный набор регистров и память, с которой можно работать посредством индексирования

Универсальные модели вычислений

модель	описание
конец XX века	
языки программирования	Java, C, C++, Python, Matlab, R, Fortran, Lisp, Haskell, Basic, Prolog
системы Линденмайера (Линденмайер, 1976)	Правила замены строк, применяемые параллельно (используются для моделирования роста растений)
«бильярдный» компьютер (Фредкин и Тоффоли, 1982)	Неразличимые (одинаковые) бильярдные шары перемещаются на плоскости с упругими столкновениями между шарами и с внутренними препятствиями
компьютер на базе частиц	Частицы передают информацию в пространстве (вычисления происходят при столкновении частиц)
одномерный клеточный автомат (Кук, 1983)	Одномерный массив (вектор) двоичных значений, изменяющихся по заданным правилам и с учетом значений их соседей
квантовый компьютер (Дейч, 1985)	Вычисления, основанные на суперпозиции квантовых состояний (на базе работ Фейнмана 1950-х годов)
карты обобщенного сдвига (Мур, 1990)	Одна классическая частица перемещается в трехмерном поле, формируемом параболическими заркалами
компьютер на базе ДНК (Адельман, 1994)	Вычисления посредством биологических операций над цепями ДНК

Универсальные модели вычислений (окончание)

Это важное знание, потому что машина Тьюринга при всей ее простоте может использоваться как универсальная модель для доказательства различных фактов, относящихся к вычислениям. Свойства машин Тьюринга, которые могут быть доказаны с математической точностью, также относятся и к компьютерам, которыми мы регулярно пользуемся. В следующем разделе мы рассмотрим одно из самых важных таких свойств.

В какой мере знания о машинах Тьюринга относятся к самому реальному миру? Это вопрос скорее философский, но и над ним стоит поразмыслить.

Вопросы и ответы

В. Значит, действительно можно построить машину Тьюринга, которая моделирует обычный компьютер, такой как микропроцессор в моем ноутбуке?

О. Да. Именно это и утверждает тезис Чёрча — Тьюринга. В классической книге Мински приведен прообраз такой машины.

В. Разве модель Тьюринга не является однозначно более мощной, чем реальные компьютеры, потому что ее лента бесконечна, а память реальных компьютеров ограниченна?

О. В техническом смысле — да. Реальный компьютер можно смоделировать очень большим ДКА. Однако при этом для моделирования компьютера с 1 Гбайт памяти потребуется более 21 000 000 000 состояний! И хотя реальный компьютер может обращаться только к ограниченному объему памяти, на практике эта память становится почти безграничной, если добавить Интернет. Кроме того, если вы соглашаетесь, что количество доступных битов во Вселенной ограниченно, придется признать, что машина Тьюринга является чисто воображаемой конструкцией.

В. Машина Тьюринга может моделировать любые вычисления?

О. Машины Тьюринга предназначены для разрешения языков и вычисления функций. Другие виды алгоритмов не лучшим образом вписываются в эту модель — например, генерирование случайных чисел, программирование автомобиля на автономном управлении или приготовление суфле. Чтобы смоделировать такие вычисления, необходимо подключить машину Тьюринга к соответствующим периферийным устройствам.

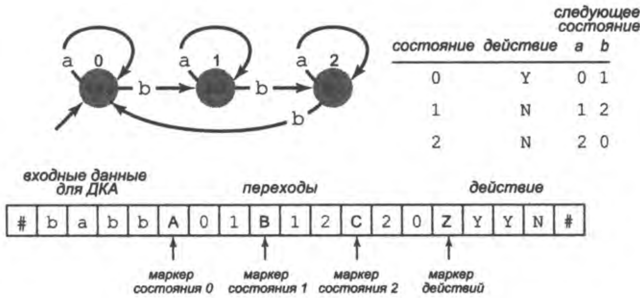
В. Почему бы просто не представить себе машину более мощную, чем машина Тьюринга?

О. Такие попытки уже делались. Сверхуниверсальные вычислительные устройства являются абстрактными моделями, которые теоретически могут выполнять вычисления, недоступные для машины Тьюринга. Один из принципов построения таких моделей заключается в хранении непрерывных значений вместо дискретных символов. Однако неизвестно, существуют ли непрерывные значения в природе, а если существуют — существуют ли природные процессы, способные использовать эту возможность.

Упражнения повышенной сложности

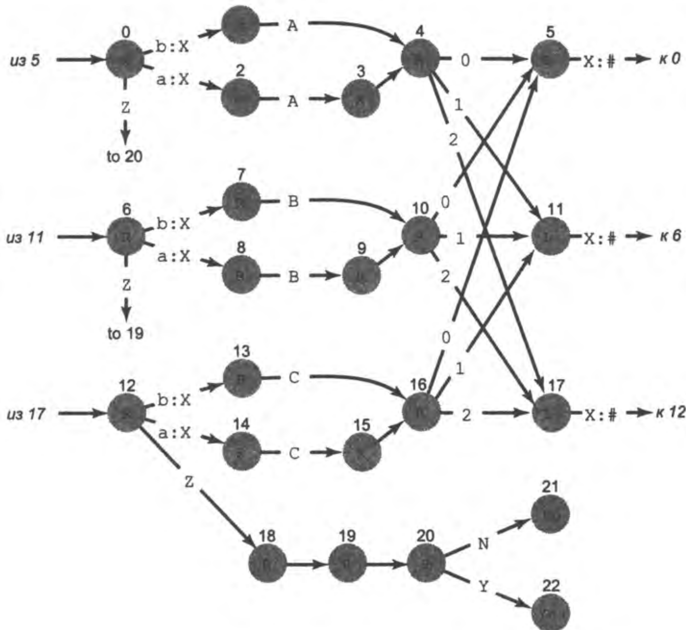
5.3.1. *Универсальный виртуальный ДКА (представление).* Разработайте представление ДКА, подходящее для использования на ленте машины Тьюринга.

Решение: начните с входных данных для ДКА, за которым следуют четыре строки таблицы состояний, помеченные символами-маркерами `A` (для состояния 0), `B` (для состояния 1) и `C` (для состояния 2). За каждым из этих символов следуют две цифры (каждая из которых — 0, 1 или 2), задающие следующее состояние для двух возможных входных символов ДКА. За таблицей состояния следует символ-маркер `Z`, а за ним идут три символа, определяющие действие `Y` или `N` для каждого состояния.



5.3.2. *Универсальный виртуальный ДКА.* Разработайте машину Тьюринга, моделирующую работу любого заданного ДКА с тремя состояниями при любых заданных входных данных.

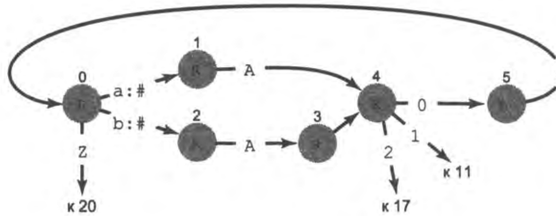
Решение.



Чтобы увидеть, как работает эта машина Тьюринга, рассмотрите первые шесть состояний справа. Эти состояния соответствуют состоянию 0 ДКА (для каждого состояния ДКА существуют шесть аналогичных состояний).

- Состояние 0 выполняет сканирование вправо в поисках входного символа ДКА (или маркера Z, если такого символа нет).

- Состояния 1 и 2 сканируют ленту в поисках маркера A; за ними следуют переходы для состояния ДКА 0. Если прочитан входной символ b, то состояние 3 пропускает следующий символ.



- Состояние 4 выполняет переход либо в состояние 5, либо в соответствующие состояния частей машины, соответствующих состояниям ДКА 1 и 2 (11 и 17).
- Состояние 5 сканирует ленту до левой границы входных данных и приводит машину в готовность к чтению следующего символа.

При обнаружении маркера Z, указывающего на завершение ввода данных для ДКА, состояние 20 читает действие для состояния 0 и входит в нужное состояние (Y или N).

Части машины, соответствующие состояниям ДКА 1 и 2, идентичны шести состояниям, соответствующим состоянию ДКА 0, за исключением того, что они сканируют ленту в поисках маркеров B и C соответственно, чтобы найти свои строки в таблице состояний, а затем перейти к подходящему символу действия после сканирования маркера Z в состояниях 19 и 18 соответственно. Обратите внимание: у этой машины Тьюринга нет состояния H и она не может войти в бесконечный цикл, потому что каждый ДКА либо принимает свою входную строку, либо отклоняет ее. Таким образом, для каждого ДКА и любой входной строки она решает, принимает ли заданный ДКА эти входные данные.

5.3.3. *Универсальный виртуальный ДКА (трассировка).* Приведите трассировку машины Тьюринга из упражнения 5.3.2 для нашего ДКА для количества b, кратного 3 (см. упражнение 5.3.1), с входными данными babb. Затем объясните, как расширить трассировку для babbb.

Частичное решение. См. следующую страницу.

5.3.4. *Универсальный виртуальный ДКА (моделирование).* Создайте текстовый файл с табличным представлением машины Тьюринга для универсального ДКА из упражнения 5.3.2, загрузите файлы TuringMachine.java и Tape.java с сайта книги. Измените их так, как описано в упражнении 5.2.7 и 5.2.8, и проверьте свое решение упражнения 5.3.3, запустив машину для заданных входных данных.

```
[#] b  a  b  b  A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #      исходное состояние 0

 #  # [a] b  b  A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #   0-2    сканирование вправо до b
 #  #  a  b  b  A [0] 1  B  1  2  C  2  0  Z  Y  Y  N  #   2-3    сканирование вправо до A
 #  #  a  b  b  A  0 [1] B  1  2  C  2  0  Z  Y  Y  N  #   3-4    пропуск
 #  #  a  b  b  A [0] 1  B  1  2  C  2  0  Z  Y  Y  N  #   4-11   переход в состояние ДКА 1
 #  # [a] b  b  A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #   11-6   сканирование влево до #
 #  #  # [b] b  A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #    6-7   сканирование вправо до a
 #  #  #  b  b  A  0  1  B [1] 2  C  2  0  Z  Y  Y  N  #   7-10   сканирование вправо до B
 #  #  #  b  b  A  0  1 [B] 1  2  C  2  0  Z  Y  Y  N  #   10-11  переход в состояние ДКА 1
 #  #  # [b] b  A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #   11-6   сканирование влево до #
 #  #  #  # [b] A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #    6-8   сканирование вправо до b
 #  #  #  #  b  A  0  1  B [1] 2  C  2  0  Z  Y  Y  N  #    8-9   сканирование вправо до B
 #  #  #  #  b  A  0  1  B  1 [2] C  2  0  Z  Y  Y  N  #   9-10   пропуск
 #  #  #  #  b  A  0  1 [B] 1  2  C  2  0  Z  Y  Y  N  #   10-17  переход в состояние ДКА 2
 #  #  #  # [b] A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #   17-12  сканирование влево до #
 #  #  #  #  #  A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #   12-14  сканирование вправо до b
 #  #  #  #  #  A  0  1  B  1  2  C  2  0  Z  Y  Y  N  #   14-15  сканирование вправо до C
 #  #  #  #  #  A  0  1  B  1  2  C  2 [0] Z  Y  Y  N  #   15-16  пропуск
 #  #  #  #  #  A  0  1  B  1  2  C [2] 0  Z  Y  Y  N  #   16-5   переход в состояние ДКА 0
 #  #  #  #  # [A] 0  1  B  1  2  C  2  0  Z  Y  Y  N  #    5-0   сканирование влево до #
 #  #  #  #  #  A  0  1  B  1  2  C  2  0  Z [Y] Y  N  #   0-20   сканирование вправо до Z

 #  #  #  #  #  A  0  1  B  1  2  C  2  0  Z [Y] Y  N  #   0-20   принятие в состоянии 22
```

5.3.5. *Ограниченная машина Тьюринга.* Докажите, что машина Тьюринга с кареткой, которая может двигаться только в одном направлении, не является универсальной. Для этого найдите язык, который не может быть разрешен такой машиной.

5.3.6. *Полусистемы Туэ.* Рассмотрим следующее множество правил замены строк:

```
 a -> c
aa -> b
ab -> abc
```

Правила должны применяться по одному в произвольном порядке для преобразования одной строки в другую. Возможно ли преобразовать aababca в bbccbcc?

Решение: да. aababca -> aabcabca -> bbcabca -> bbccbca -> bbccbcc

5.3.7. *Системы Туэ.* Рассмотрим следующее множество правил замены строк:

```
ac <-> ca
ad <-> da
bc <-> cb
bd <-> db
eca <-> ce
edb <-> de
cdca <-> cdcae
aaa <-> aaa
daa <-> aaa
```

Как и для полусистем Туэ, правила должны применяться по одному в произвольном порядке для преобразования одной строки в другую, но симметричны ли они (могут ли они применяться в любом направлении)? Возможно ли преобразовать `abcaccddaa` в `aaa`?

5.3.8. *Системы Маркова.* В системе Маркова вы начинаете с некоторой строки и применяете правила замены строк в заданном порядке, пока продолжает действовать хотя бы одно правило. Если в итоге получится 1, то исходная строка принимается; в противном случае она отклоняется. Для примера рассмотрим следующую систему Маркова:

```
 ab -> 1
a1b -> 1
```

Строка `aaabbb` будет принята системой, потому что сначала применяется первое правило для получения `aa1bb`, после чего дважды применяется второе правило для получения 1. С другой стороны, строка `aabbabb` отклоняется, потому что после двукратного применения первого правила мы получим `a1b1b`, после чего второе правило дает `11b`, и на этом все кончается. Спроектируйте систему Маркова, которая распознает язык всех палиндромов над алфавитом `{a, b}`.

5.3.9. *Системы Поста.* Для заданного списка аксиом и правил замены, состоящих из переменных (буквы верхнего регистра) и символов (остальные символы), примените в произвольном порядке правила замены для получения строк. Например, если начать с аксиомы `1+1=11` и правил замены

```
X+Y=Z -> X1+Y=Z1
X+Y=Z -> X+Y1=Z1
```

можно применить первое правило три раза подряд для получения `1111+1=11111`, затем применить второе правило дважды для получения `1111+111=1111111` и, наконец, снова применить первое правило для получения `11111+111=11111111`. Опишите язык, генерируемый этой системой Поста.

5.3.10. *Парные круглые скобки.* Спроектируйте систему Поста, которая генерирует все строки из круглых скобок, имеющие корректную парную структуру: `()`, `()()`, `(()())`, `((()(())))` и т. д.

5.3.11. *Системы Линденмайера.* Система Линденмайера (*L*-система) начинает с исходной строки и применяет правила замены параллельно — допустим, заменяя все вхождения `F` на `FLFRRFLF`. Если исходная строка содержала символы `FRRFRRF`, то после одной итерации мы получим `FLFRRFLFRRFLFRRFLFRRFLFRRFLF`. Используйте эти правила для разработки программы-клиента для `Turtle` (листинг 3.2.4), строящей «снежинку Коха» (см. раздел 3.2). Интерпретируйте `F` как движение вперед с рисованием, `L` как поворот против часовой стрелки на 60° и `R` как поворот по часовой стрелке на 60°. Строка после `n`-й итерации содержит инструкции для вычерчивания снежинки Коха порядка `n`. Напишите программу Java с именем `Lindenmayer`, которая получает аргумент командной строки `n` и выводит инструкции для построения снежинки Коха порядка `n`. *Подсказка*: используйте метод `String.replaceAll()`.

5.3.12. *Квадратичный остров Коха.* Используйте следующую систему Линденмайера для создания квадратичного фрактала острова Коха: начните со строки `F` и выполняйте многократную замену всех вхождений `F` на `FLFRFRFFLFLFRF`.

5.3.13. *L-системы с квадратными скобками.* Исследуйте систему Линденмайера, которая начинается со строки `F` и многократно выполняет параллельную замену `F` на `FFR[RFLFLF]L[LFRFRF]`. Интерпретируйте `L` и `R` как повороты на 22°, а символы `[` и `]` — как занесение и извлечение из стека текущего состояния «черепашки» (позиция и угол). Квадратные скобки позволяют избежать случая, при котором фигура состоит из одной линии.

5.3.14. *Кривая Гильберта.* Используйте следующую *L*-систему для создания кривой Гильберта: начните со строки `A`, затем выполняйте многократную замену `A` строкой `LBFRAFARFBL` и `B` строкой `RAFLBFBLFAR`, применяя оба правила замены одновременно. Первая итерация дает `LBFRAFARFBL`, а вторая — `LRAFLBFBLFARFRLBFRAFARFBLFLBFRAFARFBLRFRAFLBFBLFARL`.

Когда потребуется построить кривую, интерпретируйте `L` как поворот налево на 90°, `R` как поворот направо на 90° и `F` как движение вперед (`A` и `B` игнорируются).

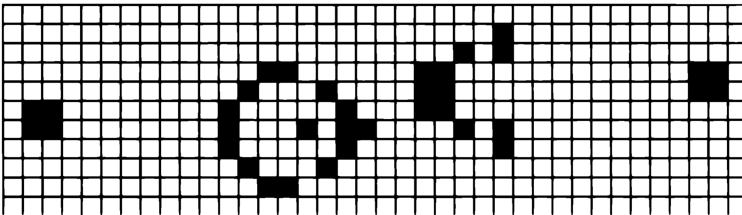
5.3.15. *Кривая дракона.* Используйте следующую *L*-систему для создания кривой дракона (см. упражнение 1.2.35): начните со строки `FA`. Затем выполняйте многократную замену `A` строкой `ALBFL` и `B` строкой `RFARB`. Интерпретируйте буквы так, как описано в упражнении 5.3.14. После исключения `A` и `B` первые 3 итерации выглядят так: `FLFL`, `FLFLLRFRFL` и `FLFLLRFRFLLRFLFLRRFRFL`.

5.3.16. *Таг-системы.* Напишите программу, которая читает двоичную строку и применяет следующую таг-систему `(00, 1101)`: если первый бит равен 0, удалить первые 3 бита и присоединить `00`; если первый бит равен `1`, удалить первые три бита и присоединить `1101`. Повторять, пока строка содержит не менее 3 бит. Попробуйте определить, приведет ли ввод следующих данных к остановке или зацикливанию: `10010`, `100100100100100100`. Рассмотрите возможность применения очереди.

5.3.17. *Машина Поста.* Предполагая, что таг-системы универсальны, покажите, что машина Поста (ДКА с очередью) также является универсальной; для этого опишите способ моделирования любой таг-системы при помощи машины Поста.

5.3.18. *2-стековый ДКА*. Предполагая, что машина Поста универсальна, покажите, что ДКА с двумя стеками также универсальна: опишите, как смоделировать очередь с двумя стеками.

5.3.19. *Синтез «планеров»*. Инициализируйте игру «Жизнь» Конуэя (см. упражнение 2.4.20) конфигурацией, показанной ниже, в левом верхнем углу сетки 50 × 50. Эта конфигурация, называемая «планерной пушкой», генерирует новые планеры. Она была изобретена Р. Госпером (R. Gosper) в 1970 году в ответ на опубликованную Конуэем задачу — найти конфигурацию, способную к бесконечному росту. Генератор «планеров» можно рассматривать как средство передачи информации. В частности, он использовался Полом Ренделлом (Paul Rendell) в качестве одного из структурных элементов в реализации модели универсальной машины Тьюринга в 2011 году.



5.4. Вычислимость

Теперь, когда мы четко определились с тем, что такое алгоритм (машина Тьюринга), можно заняться исследованием природы вычислений на основе доказательства фактов, относящихся к машине Тьюринга. Из тезиса Чёрча — Тьюринга следует, что если вы располагаете конечным, детерминированным и эффективным методом для решения задачи, то существует машина Тьюринга, которая решает эту задачу. Впрочем, обратное утверждение получается намного более сильным: *если удается доказать, что не существует машины Тьюринга для разрешения языка или вычисления функции, то предполагается, что для этой задачи не существует ни одного физически реализуемого процесса*. Отделение задач, которые решаются некоторой машиной Тьюринга, от задач, которые не решаются никакой машиной Тьюринга, было центральной темой статьи Тьюринга. В этом разделе мы опишем исключительно важное следствие из модели машины Тьюринга: ему удалось доказать, что существуют неразрешаемые языки и невычислимые функции — те, для которых не существует машин Тьюринга, решающих эти задачи. Такие задачи называются *алгоритмически неразрешимыми*, или просто *нерешаемыми*, потому что мы предполагаем, что ни один компьютер из числа существующих или возможных в будущем не сможет решить эту задачу: для такой задачи не существует *алгоритма*.

Неразрешимость — очень сильное утверждение о задаче. Оно означает не только то, что ученые еще не обнаружили алгоритм для задачи, но и то, что его найти

невозможно. Это важное знание: если вам известно, что вы пытаетесь решить нерешаемую задачу, вы можете сэкономить свое время и перейти на более простой вариант задачи. В прошлом веке многие люди усердно работали над решением задач, которые позднее оказались неразрешимыми. Люди, не знающие о теории Тьюринга, работают «вслепую» и, с большой вероятностью, тратят значительные усилия на попытки решений, которые невозможны.

Контекст: программа Гильберта

В начале XX века Давид Гильберт, выдающийся математик своего времени, запустил смелую программу для решения самых фундаментальных задач из области логики и математики. Он предложил своим коллегам доказать формальную версию следующих трех утверждений.

- *Непротиворечивость математики*: невозможно доказать одновременно и утверждение, и обратное ему утверждение (например, 1 = 2).
- *Полнота математики*: если математическое утверждение истинно, то возможно доказать его истинность.
- *Разрешимость математики*: для любой математической теоремы существует пошаговая последовательность применения аксиом, которая приводит к доказательству.

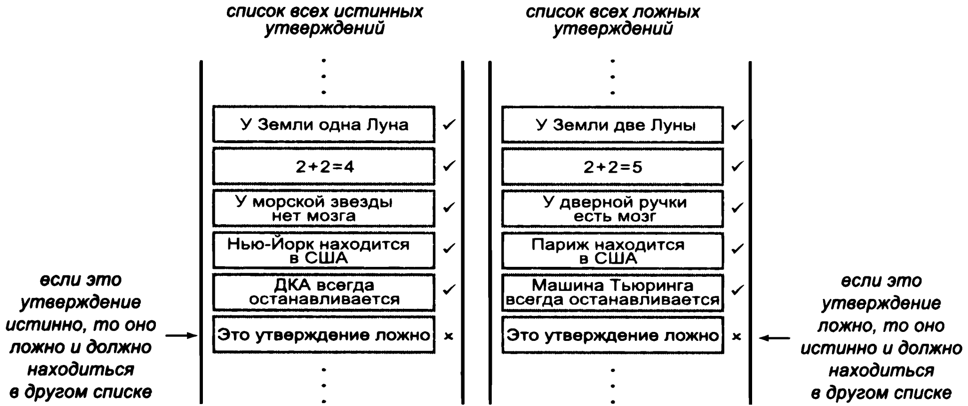
Одно из самых глубоких научных достижений XX века произошло в 1930 году, когда Курт Гёдель разрешил первые из этих двух утверждений самым неожиданным способом: он доказал, что любая система аксиом (способная моделировать арифметику) не может быть одновременно целостной и полной. Это открытие потрясло основы интеллектуального мира, вызвало замешательство в области устоявшихся представлений и вдохновило разносторонние исследования в области математики.

Как выглядит точное определение «пошаговой процедуры»? Как выглядит точное определение «математической теоремы»? Что такое «математика»? Машина Тьюринга успешно ответила на первый из этих вопросов и заложила основу для окончательного вывода по программе Гильберта: математика не может быть одновременно целостной и полной, и она *неразрешима*: *существуют теоремы, которые невозможно доказать*. В данном разделе эта концепция будет рассмотрена в вычислительном контексте.

Разминка: парадокс лжеца

Для разминки рассмотрим *парадокс лжеца*, восходящий к творчеству древнегреческих философов. Предположим, наша цель заключается в классификации всех утверждений на истинные и ложные. Например, можно отнести утверждения «2 + 2 = 4» и «У Земли одна Луна» к истинным, а утверждения «2 + 2 = 5» и «У Земли две Луны» к ложным. На первый взгляд эта задача выглядит разумной,

но возникает непреодолимое препятствие при попытке классифицировать следующее утверждение: «Это утверждение ложно». Если отнести его к истинным, то из истинности утверждения «Это утверждение ложно» следует, что оно ложно. Если же отнести его к ложным, то из ложности утверждения следует его истинность, поэтому его следует отнести к истинным. В любом случае возникает противоречие.



Парадокс лжеца

Единственный способ разрешить это противоречие — признать ложность исходной предпосылки. Этот метод доказательства известен под названием «сведение к абсурду» (*reductio ab absurdum*): если предположение приводит к абсурдному заключению, то оно должно быть ложным. В случае парадокса лжеца исходное предположение гласило, что все утверждения можно разделить на истинные и ложные, и оно приводило к противоречию при любом способе классификации, поэтому исходное предположение должно быть ложным. Иначе говоря, *невозможно* разделить все утверждения на истинные и ложные. На первый взгляд аргумент кажется тривиальным, но в действительности он достаточно глубок, и подобная аргументация закладывает основу для многих интересных доказательств. Обратите внимание: этот результат не является примером нарушения целостности математики и даже парадокса. Это доказательство устанавливает математический факт.

Проблема остановки

Вслед за Тьюрингом мы продемонстрируем факт существования задач, не имеющих решения; для этого мы покажем, что задача (или *проблема*[1]) остановки неразрешима. В неформальном виде проблема остановки описывается просто: для заданной

[1] Для математических, вычислительных и тому подобных задач традиционно используется термин «problem», который чаще переводится как «задача», но некоторые задачи принято называть именно «проблемами». — *Примеч. науч. ред.*

программы и ее входных данных определить, остановится ли эта программа при выполнении на этих данных. Поскольку компьютеры в наши дни редко останавливаются аварийно, мы будем использовать термин «остановится» как синоним «не войдет в бесконечный цикл». Например, функция Java считается «остановившейся», если она возвращает управление на сторону вызова без зацикливания.

Все программисты сталкивались с печальными последствиями зацикливания, и возможность заранее выявить зацикливание перед запуском программы была бы безусловно полезной. Это подтвердит любой преподаватель, которому доводилось ставить оценки многочисленным программам, написанным новичками! Или другой пример проблемы, с которой сталкивается отдел контроля качества фирмы-разработчика: конечно, было бы полезно заранее удостовериться в том, что программный продукт не приведет к зависанию мобильного устройства. Но доказательство Тьюринга (в сочетании с универсальностью) говорит, что разработать программу, которая будет проверять, не войдет ли любая другая программа с заданными входными данными в бесконечный цикл, невозможно.

Универсальная машина Тьюринга — это программа, которая получает на вход описание машины Тьюринга и ее входные данные и моделирует работу этой машины. Решение проблемы остановки требует выяснить, существует ли машина Тьюринга, способная выполнить (вроде бы) более простую задачу — определить, войдет ли заданная другая машина Тьюринга в состояние остановки при заданных входных данных.

Проблема остановки на примере программы на Java

Вследствие универсальности проблему остановки можно исследовать на примере программы на языке Java. И хотя приведенное ниже доказательство без особого труда адаптируется для машин Тьюринга (см. упражнение 5.4.7), пример с Java-программой для человека с опытом программирования кажется более интуитивно понятным. Итак, проблема остановки формулируется следующим образом: существует ли функция `halts(f,x)`, которая получает в аргументах функцию `f` и входные данные `x` (закодированные в строковом виде) и определяет, приведет ли вызов `f(x)` к зацикливанию? Если говорить конкретнее, функция `halts(f,x)` должна иметь следующую форму:

```
public static boolean halts(String f, String x)
{
   if ( /* что-то невероятно умное */ ) return true;
   else return false;
}
```

Чтобы разрешить проблему остановки, сама функция `halts()` не должна зацикливаться: она должна выдавать правильный ответ для каждой функции `f` (с одним аргументом `String`) и для каждого варианта входных данных `x`.

Как и в случае с универсальной машиной Тьюринга, язык Java может рассматриваться как программа, которая получает вашу программу и ее входные данные

(закодированные в строковом виде) в виде двух аргументов, а затем выполняет вашу программу для получения результата вычислений. Существует ли более простая программа, которая получает те же два аргумента и просто сообщает, возникнет ли в этой программе бесконечный цикл?

Поучительный пример

Чтобы вы поняли, почему эта задача считается такой сложной, рассмотрим следующие две функции, которые различаются всего одним символом:

```
public static void f(int x)
{
  while (x != 1)
     if (x % 2 == 0) x = x / 2;
     else x = 2*x + 1;
}
```

```
public static void g(int x)
{
  while (x != 1)
     if (x % 2 == 0) x = x / 2;
     else x = 3*x + 1;
}
```

Левая функция входит в бесконечный цикл в том и только в том случае, если x не является положительной степенью 2, а правая функция реализует последовательность Коллаца, упоминавшуюся в примере 2.3.29; в этом случае ситуация неясна, потому что никто не знает, завершается ли эта функция для всех x. Как долго следует наблюдать за ней для любого заданного x, чтобы решить, что функция вошла в бесконечный цикл? Можно запустить программу и посмотреть, что произойдет. Возможно, программа остановится... но что делать, если она продолжает работать? Не исключено, что если она поработает еще немного, то остановится. В общем случае невозможно определить это с полной уверенностью. Математики доказали, что она завершится для любого значения x, меньшего 10^{300}, но всегда существует большее значение, для которого программа еще не была протестирована (см. упражнение 5.4.7). Это крайний случай, но он подчеркивает тот факт, что нет простого способа определить, завершится ли заданная программа. Оказывается, смоделировать работу программы шаг за шагом проще, чем определить, зацикливается ли она. Более того, второе вообще невозможно, и сейчас мы докажем этот удивительный факт.

Доказательство неразрешимости

Идея того, что задача может в принципе не иметь решения, способна поколебать ваши представления о вычислениях, поэтому мы рекомендуем несколько раз перечитать это доказательство, пока вы не убедитесь в его истинности и не проникнетесь самой идеей. Это одна из важнейших идей XX века.

Теорема (Тьюринг, 1937). Проблема остановки неразрешима.

Идея доказательства. В порядке доказательства от противного предположим, что функция `halts(f,x)`, описанная выше, существует. Первым шагом станет создание новой функции `strange()`, которая получает на вход одну функцию `f` (закодированную в виде строки) и вызывает `halts()` следующим образом:

```
public static boolean strange(String f)
{
   if (halts(f, f))
      while (true) /* бесконечный цикл */ ;
}
```

На первый взгляд идея вызова `halts(f,f)` для проверки того, останавливается ли программа при получении самой себя в качестве входных данных, выглядит странно; здесь это является лишь инструментом доказательства. Но на самом деле идея не такая уж странная: например, представьте, что проектировщик компилятора желает проверить, что компилятор не входит в бесконечный цикл, когда он компилирует сам себя.

Что в действительности делает `strange()`? Вспомните, что `halts(f,x)` возвращает `true`, если `f(x)` останавливается, или `false`, если `f(x)` не останавливается (где под термином «останавливается» следует понимать «не входит в бесконечный цикл»). Теперь проанализируем код.

- Если `f(f)` останавливается, то `strange(f)` не останавливается.
- Если `f(f)` не останавливается, то `strange(f)` останавливается.

А теперь наступает критически важный шаг: что произойдет, если вызвать функцию `strange()`, передав ей в качестве входных данных *саму себя* (закодированную в строковом виде)? Другими словами, мы заменяем в двух предыдущих утверждениях `f` на `strange`, получая следующие два (странных) утверждения.

- Если `strange(strange)` останавливается, то `strange(strange)` не останавливается.
- Если `strange(strange)` не останавливается, то `strange(strange)` останавливается.

Оба утверждения явно противоречивы, из чего мы приходим к заключению, что гипотетическая функция `halts(f, x)` не существует. Таким образом, проблема остановки не имеет решения!

Если вам кажется, что мы проделали какой-то логический трюк, перечитайте доказательство и попробуйте выполнить упражнение 5.4.7. Неразрешимость проблемы остановки — фундаментальный результат, отражающий саму природу вычислений, который имеет множество практических следствий.

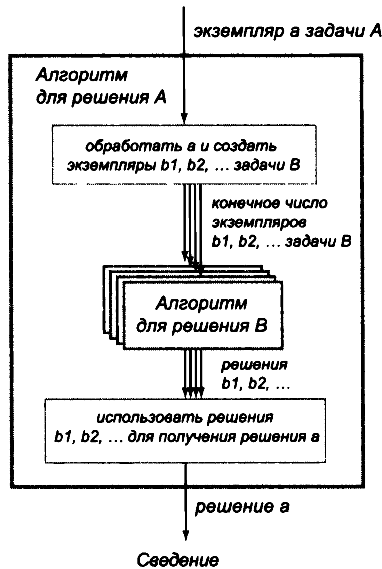
Сведение

Хотя проблема остановки чрезвычайно интересна сама по себе, в действительности еще важнее то, что она может использоваться для доказательства неразрешимости других значимых задач. Прием, который при этом используется, называется сведением задачи.

Определение. Задача A *сводится* к другой задаче B, если, имея соответствующую процедуру решения для B, любой экземпляр a задачи A может быть решен по следующей схеме.

- Представить a как множество экземпляров b_1, b_2, ... задачи B.
- Используя известную процедуру для B, получить решения b_1, b_2, ...
- Использовать решения b_1, b_2, ... для решения a.

Концепция проста, но поначалу она может вызвать замешательство из-за возможного появления нескольких экземпляров B. На самом деле во всех наших примерах используется всего один экземпляр B. Кроме того, для проблем неразрешимости сведение было использовано в контрапозиции. Мы выбрали проблему остановки в качестве задачи A, а затем показали, что решение задачи B может использоваться для решения задачи A, то есть проблемы остановки. Из этого следует, что и задача B не имеет решения, потому что проблема остановки неразрешима.



Сведение

Проблема тотальности

В качестве первого примера рассмотрим так называемую *проблему тотальности*. Можно ли написать программу, которая получает на вход функцию и определяет, зацикливается она или нет при *любых* входных данных? Например, решение этой задачи для функции `g()` на с. 778 разрешило бы вопрос об истинности гипотезы Коллаца. Любая фирма-разработчик не отказалась бы иметь такую программу, чтобы знать, что ее продукты никогда не зацикливаются. Тем не менее сведение позволяет доказать, что эта задача не решается.

Утверждение А. Проблема тотальности неразрешима.

Идея доказательства. Выбираем проблему тотальности как «задачу В»; предположим, имеется функция Java `alwaysHalts()`, которая ее решает: `alwaysHalts()` получает как аргумент произвольную функцию `f` и выводит `Yes`, если `f(x)` останавливается для всех x, или `No`, если `f(x)` входит в бесконечный цикл для некоторого x. Таким образом, появился способ решения задачи остановки с использованием `alwaysHalts()`: для любой функции `f` и аргумента x определяем функцию `g()`, которая вызывается без аргументов и просто вызывает `f(x)`. В этом случае вызов `alwaysHalts(g)` выводит `Yes`, если `f(x)` останавливается; в противном случае выводится `No`. Другими словами, она решает проблему остановки (для которой доказана ее неразрешимость). Возникает противоречие, поэтому исходное предположение о существовании `alwaysHalts()` должно быть ложным. Следовательно, задача тотальности неразрешима.

Короче говоря, мы утверждаем, что *проблема остановки сводится к проблеме тотальности*, поэтому и проблема тотальности должна быть неразрешимой. Если бы проблему тотальности можно было решить, то мы могли бы решить и проблему остановки. Так как проблема остановки неразрешима, то и проблема тотальности тоже должна быть неразрешимой.

Более того, этот аргумент работает для любой неразрешимой задачи A. Тщательно проанализировав этот пример и пример из следующего подраздела, вы получите хорошее представление о том, как работает этот метод. Если вы не искушены в математике, при первом чтении пропустите доказательства и попробуйте понять выводы. Позже доказательства можно будет изучить более подробно, так как они в действительности весьма просты по сравнению с типичными математическими доказательствами.

Проблема эквивалентности программ

Можно ли написать программу, которая получает на входе две функции и определяет, являются ли эти функции эквивалентными (то есть возвращают один и тот же результат для любых заданных входных данных)? И снова любая фирма-разработчик наверняка не отказалась бы от такой программы. И снова мы можем доказать неразрешимость этой задачи (так называемой *проблемы эквивалентности*) посредством сведения.

Утверждение В. Проблема эквивалентности программ неразрешима.

Доказательство. Предположим, имеется функция Java `areEquivalent()`, которая получает в аргументах произвольные функции `f` и `g` и выводит `Yes`, если эти функции эквивалентны, или `No` в противном случае. Для любой функции Java `f()` мы вызываем `areEquivalent(f, h)`, где `h` — функция, которая просто возвращает управление. Тогда вопрос эквивалентен вопросу об определении того, что `f()` никогда не входит в цикл ни для каких входных данных, то есть к проблеме тотальности.

Короче говоря, мы утверждаем, что *проблема тотальности сводится к проблеме эквивалентности*, поэтому задача эквивалентности должна быть неразрешима. Если бы проблему эквивалентности можно было разрешить, то мы могли бы разрешить и проблему тотальности. Так как проблема тотальности неразрешима, то и проблема эквивалентности тоже должна быть неразрешимой.

Теорема Райса

Эти свойства — всего лишь верхушка айсберга. Определим *функциональное свойство* программы как свойство, связывающее вход и выход программы (то есть вычисляемую ею функцию), нетривиальное в том смысле, что оно присуще некоторым, но не всем программам. В своей диссертации 1951 года Генри Райс доказал теорему, из которой следует вот что.

Теорема (Райс, 1951). Задача определения того, обладает ли заданная программа *любым* заданным функциональным свойством, неразрешима.

Эта теорема имеет исключительно широкое практическое применение. Выводит ли программа Java более 10^{1000} символов? Выводит ли она хоть один символ? Входит ли она в бесконечный цикл более чем с одним значением своего аргумента? Останавливается ли она, если все ее аргументы положительны? Останавливается ли программа Java без аргументов? Без особых усилий этот список можно дополнить десятками свойств. Многие, многие естественные вопросы связаны с функциональными свойствами программ.

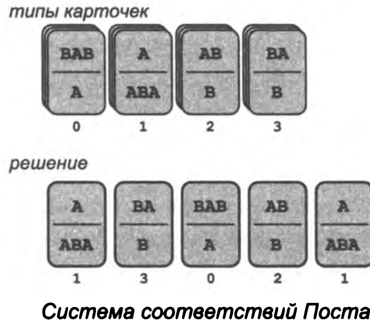
В практическом смысле представления о неразрешимости и теорема Райса помогают понять, почему так сложно добиться надежности в программных системах. Как бы нам ни хотелось написать программы, гарантирующие, что наш продукт обладает любым количеством нужных свойств, *сделать это невозможно*. Люди, которые понимают это, добьются в области компьютерных вычислений значительно большего успеха, чем те, кто этого не понимают.

Другие примеры задач, не имеющих решения

Проблема неразрешимости существует не только для программ, обрабатывающих другие программы (какими бы важными ни были эти применения). За несколько десятилетий, прошедших с момента, когда Тьюринг представил эту концепцию, ученые с помощью метода сведения существенно расширили количество задач, о которых известно, что они принципиально не имеют решения. Практические применения встречаются во многих областях математики, науки и техники. В каждом случае некоторая заведомо неразрешимая задача сводится к новой задаче, что доказывает неразрешимость этой новой задачи, потому что ее решение подразумевало бы решение (в конечном итоге) проблемы остановки. Многие важные и интересные

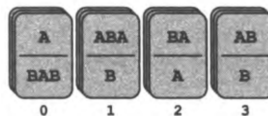
примеры приведены в таблице на с. 784; некоторые из них рассматриваются более подробно.

Проблема соответствий Поста



Система соответствий Поста

Следующая задача со строками, написанными на карточках, впервые была проанализирована Эмилем Постом (Emil Post) в 1940-х годах. Система соответствий Поста представляет собой множество определенных типов карточек; каждый тип определяется двумя строками — в верхней и нижней части карточки. Например, в системе на верхней иллюстрации изображены карточки четырех типов: у первой наверху находится строка BAB, а внизу строка A, у второй — наверху строка A, а внизу строка ABA и т. д. Вопрос в том, возможно ли расположить карточки (используя любое количество карточек каждого типа) в последовательность так, чтобы верхняя и нижняя строки совпадали. В нашем примере ответ положительный, как видно из решений в нижней части диаграммы: сначала идет карточка типа 1, затем карточка типа 3, карточка типа 0, карточка типа 2 и вторая карточка типа 1 — и в верхней, и в нижней части образуется строка ABABABABA. Другой пример, который демонстрирует, что это не всегда возможно, показан внизу. Почему решения в этом случае не существует? Чтобы оно было хотя бы возможно, верхний и нижней левые символы крайней левой карточки в решении должны совпадать, но в этом примере ни у какой карточки левый верхний символ не совпадает с левым нижним, поэтому выстроить карточки не удастся. В общем случае такого простого объяснения может не быть или поиск решения может оказаться нелегкой задачей. Задача, известная как *проблема соответствий Поста*, заключается в разработке алгоритма, который может определить, существует ли решение для любой конкретной системы соответствий карточек. Отметим, что эта задача не имеет решения. Также было показано, что она сводится к другим задачам, в которых задействованы строки; следовательно, и эти задачи тоже являются неразрешимыми.



Другая система соответствий Поста

Оптимальное сжатие данных

Вероятно, вы уже использовали алгоритмы сжатия данных для сокращения размеров фотографии или видеоролика с целью их публикации или хранения. Алгоритм, используемый в вашей системе, является результатом многолетних исследований, но будет естественно спросить, возможно ли дальнейшее уменьшение размера. Формально эта идея преобразуется в задачу оптимального сжатия данных: для заданной строки найти самую короткую (по количеству символов) программу, которая выводит эту строку. Например, множество Мандельброта — изящный пример сложного изображения, которое может быть сгенерировано простой программой (см. листинг 3.2.7). Если вы попытаетесь сжать изображение множества Мандельброта с высоким разрешением в вашей системе, то, вероятно, результат сжатия будет занимать меньше места, чем оригинал, но его не сравнить с несколькими сотнями символов, составляющих программу. Существует ли алгоритм, способный построить программу по изображению? Эта задача может считаться формализованной постановкой задачи *бритвы Оккама* — найти простейшее объяснение, соответствующее факту. И хотя было бы неплохо иметь формальный метод, гарантирующий обнаружение такого точного описания, эта задача решения не имеет.

Оптимизирующие компиляторы

В сообществе исследователей языков программирования оптимальное сжатие данных известно под названием «теоремы полной занятости», потому что она гласит, что ни один компилятор не может гарантировать свою способность оптимизировать каждую программу. Теорема Райса ведет к нескольким теоремам полной занятости — существует много задач, которые нам хотелось бы поручить компиляторам, но которые не имеют решения. Содержит ли программа неинициализированные переменные? Содержит ли программа «мертвый код», который никогда не выполняется? Повлияет ли изменение значения конкретной переменной в конкретной точке на результат вычислений? Может ли программа выдать заданную строку на выходе? Как бы нам ни хотелось, чтобы компиляторы помогали в решении подобных задач, они этого сделать не могут[1].

10-я проблема Гильберта

В 1900 году Давид Гильберт обратился к Международному конгрессу математиков в Париже и предложил 23 кардинальные проблемы для начинающегося века.

---

[1] Утверждение справедливо в рамках фундаментальной теории. Оптимизирующие компиляторы реально существуют, они успешно решают перечисленные (и другие) здачи с требуемым качеством и надежностью, но лишь для некоторых частных (хотя и достаточно многочисленных!) случаев. — *Примеч. науч. ред.*

задача	описание
программы, обрабатывающие программы	
проблема остановки	Входит ли заданная программа в бесконечный цикл для заданных входных данных?
проблема тотальности	Входит ли заданная программа в бесконечный цикл для *каких-либо* входных данных?
проблема эквивалентности программ	Всегда ли две программы приходят к одинаковому результату?
управление памятью	Будет ли программа обращаться к переменной в будущем?
распознавание вирусов	Является ли заданная программа вирусом?
функциональные свойства	Обладает ли программа каким-либо функциональным свойством?
другие примеры	
проблема соответствий Поста	Применим ли заданный набор правил замены строк?
оптимальное сжатие данных	Возможно ли сжать заданную строку[1]?
10-я проблема Гильберта	Имеет ли заданный многовариантный многочлен целочисленные корни?
определенное интегрирование	Имеет ли заданный интеграл решение в замкнутой форме?
теория групп	Является ли конечно-представимая группа простой, конечной, свободной или коммутативной?
динамические системы	Является ли заданная динамическая система хаотичной?

Примеры неразрешимых задач

10-я проблема Гильберта была посвящена разработке процесса, который бы позволял определить за конечное число операций наличие целочисленного корня у заданного многочлена (с несколькими переменными). Иначе говоря, возможно

[1] Напомним, что здесь подразумевается поиск оптимального алгоритма, генерирующего требуемую строку. Если же ограничиться традиционными методами сжатия, то его возможности с практически достаточной точностью описываются формулами для информативности/энтропии сообщений. — *Примеч. науч. ред.*

ли присвоить целочисленные значения переменным многочлена, чтобы он был равен 0? Например, многочлен $f(x,y,z) = 6x^3yz^2 + 3xy^2 - x^3 - 10$ имеет целочисленный корень, так как $f(5,3,0) = 0$, а многочлен $f(x,y) = x^2 + y^2 - 3$ целочисленного корня не имеет. Эта задача, впервые предложенная 2000 лет назад Диофантом, встречается в самых разных областях, включая физику, вычислительную биологию, исследование операций и статистику. В то время формального определения алгоритма не существовало, соответственно вопрос существования неразрешимых задач даже не рассматривался. В 1970-е годы 10-я проблема Гильберта была снята совершенно неожиданным образом: на основании подготовительной работы, проделанной Мартином Дэвисом (Martin Davis), Хилари Путнэм (Hilary Putnam) и Джулией Робинсон (Julia Robinson), Юрий Матиясевич (Yuri Matiyasevich) доказал ее неразрешимость, что привело к появлению класса нерешаемых задач в самых разных практических ситуациях, в которых применялась эта модель. Например, вследствие сведения этой модели неразрешимой оказывается естественная задача планирования путешествий; это означает, что ни один алгоритм не сможет всегда гарантированно находить ответ на любой возможный запрос на построение маршрута (или определять, что такого маршрута не существует) для произвольной базы данных рейсов и тарифов.

Определенные интегралы

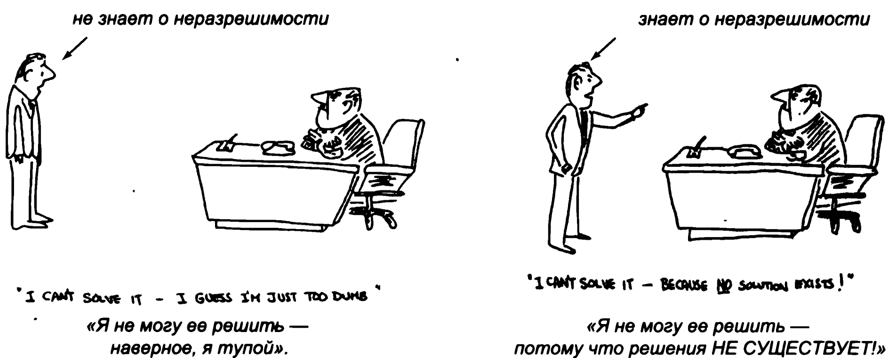
Математики и ученые в настоящее время сильно зависят от компьютерных систем, которые помогают им выполнять сложные операции с символьными данными. Компьютеру поручают «черную работу» по обсчету таких функций, как ряды Тейлора, умножение многочленов, интегрирование, дифференцирование и т. д. Одной из ключевых проблем, с которыми сталкиваются разработчики таких систем, является вычисление определенного интеграла: возможно ли найти для каждого определенного интеграла решение в замкнутой форме, в котором задействованы только полиномиальные и тригонометрические функции? Многие ученые годами искали алгоритм для этой задачи, но теперь была доказана ее неразрешимость вследствие сведения от 10-й проблемы Гилберта.

Следствия

При поверхностном знакомстве с компьютерами может появиться впечатление, что достаточно мощный компьютер способен на все. Как показывают многочисленные примеры этого раздела, это предположение совершенно неправильно. Факт существования неразрешимых задач имеет глубокие последствия — как вычислительные, так и философские. Он означает, что все компьютеры подчиняются внутренним ограничениям в силу фундаментальных свойств вычислений. Мы должны признать, что эти задачи, какими бы важными они ни были, решения не имеют.

Помимо чисто практической важности, неразрешимость (в сочетании с тезисом Чёрча — Тьюринга) приподнимает завесу над вычислительными законами природы и открывает целый ряд захватывающих философских вопросов. Например, если

тезис Чёрча — Тьюринга применим к человеческому мозгу, то человек не сможет решать такие задачи, как проблема остановки. У людей, как у компьютеров, могут быть фундаментальные ограничения. Универсальны ли природные процессы? Если универсальны, то существуют ли условия, которые не могут существовать в реальном мире из-за неразрешимости? Существуют ли природные процессы, нарушающие тезис Чёрча — Тьюринга? Такие вопросы преследовали математиков и философов с того самого момента, когда следствия работы Тьюринга приобрели широкую известность. Широко распространенное мнение о том, что его статья была одной из важнейших научных публикаций XX века, бесспорно справедливо.



Практические следствия теории Тьюринга

Вопросы и ответы

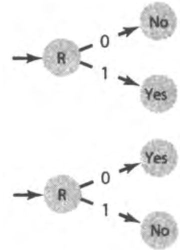
В. Неразрешимость проблемы остановки гласит, что мы не можем написать программу на Java, которая определяет, остановится ли произвольная программа с произвольными данными на входе. Но можно ли написать программу, которая определяет, что одна конкретная программа Java остановится для одного конкретного варианта входных данных?

О. Практик скажет, что это можно сделать для многих программ (таких, как HelloWorld.java). Теоретик скажет, что вы можете написать две программы: одна всегда выводит Yes, а другая всегда выводит No. Несомненно, один из двух вариантов будет правильным. Конечно, нельзя исключать, что никто никогда не узнает правильный ответ, но и доказать, что это в принципе невозможно, тоже нельзя, потому что это приведет к парадоксу. Если программа останавливается, то мы можем запустить ее и получить доказательство того, что она останавливается. Следовательно, если невозможно доказать, останавливается ли она, то она останавливаться не должна, потому что это послужило бы доказательством. Но тогда можно было бы использовать это как доказательство того, что она не останавливается!

В. Разрешим ли вопрос об истинности гипотезы Коллаца?

О. Существует другая версия той же задачи. Сипсер (Sipser) представляет ее следующим образом: пусть L — язык (над двоичным алфавитом), состоящий из единственной строки `1`, если на Марсе есть жизнь, и `0` в противном случае. Является ли язык L разрешимым?

Ответ на этот вопрос будет положительным. Применим правило исключенного среднего: либо жизнь на Марсе есть, либо ее нет. В любом случае одна из машин Тьюринга справа разрешает L, и другой возможности не существует. Тот факт, что мы понятия не имеем, какая именно это машина, несуществен. Очевидный парадокс кроется в простоте *языка*. Тот факт, что гипотеза разрешима, не дает никакой информации о том, как доказать ее истинность или ложность или как найти контрпример.



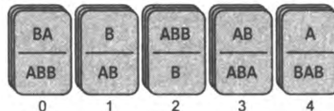
В. Возможно ли написать программу на Java, которая решает задачу остановки для функции Java, не использующей библиотечные функции и не выполняющей ввод/вывод?

О. Может показаться, что это *возможно*, потому что программа может использовать только конечный объем памяти. Но такой аргумент предполагает, что все наши компьютеры являются ДКА, а их производительность определяется некоторой галактической константой, поэтому вся описываемая теория неприменима (так как во всех случаях используется постоянный объем ресурсов). Пожалуй, эффективнее будет принять интуитивное представление о том, что константа настолько близка к бесконечности, что рассматриваемые модели отражают важнейшие свойства машин.

Упражнения

5.4.1. Допустим, в задаче соответствий Поста можно использовать не более одной карточки каждого типа. Останется ли эта задача неразрешимой?

5.4.2. Найдите два решения для следующей системы соответствий Поста или докажите, что решения не существует.



Частичное решение. `34012212`.

5.4.3. Найдите два решения для следующей системы соответствий Поста или докажите, что решения не существует.

5.4.4. Предположим, что алфавит в системе соответствий Поста содержит только одну букву, поэтому нужно найти такую конфигурацию, при которой верхняя и нижняя строки содержат одинаковое количество букв. Разработайте алгоритм решения проблемы соответствий Поста для данного случая.

5.4.5. Существует ли некоторая последовательность замен (в любом порядке) aba на bba, ba на bbb и baa на aa, которая преобразует строку baababbba в ababbbabbba? (Это пример *задачи Туэ о словах*, которая в общем случае неразрешима.)

5.4.6. Измените программу, которая вычисляет функцию Коллаца, приведенную в решении упражнения 2.3.29, с использованием класса Java `BigInteger`, чтобы она могла выполнять свои вычисления с использованием целых чисел произвольной длины.

Упражнения повышенной сложности

5.4.7. *Проблема остановки для машин Тьюринга.* Переработайте доказательство неразрешимости проблемы остановки, приведенное в тексте, для машин Тьюринга.

В каждом из следующих упражнений вам предлагается доказать, что приведенная задача не имеет решения. Задачи предназначены для читателей, сведущих в математике, которым будет интересно разрабатывать такие доказательства посредством сведения. Если вы не разбираетесь в математике, возможно, вам все равно стоит хотя бы прочитать задачи и расширить свои знания в области нерешаемых задач.

5.4.8. *Проблема самоостановки.* Можно ли написать программу, которая решает, завершается ли заданная функция с одним аргументом, в котором передается она сама? Докажите, что эта задача неразрешима, по аналогии с рассуждениями для проблемы остановки.

5.4.9. *Функция Радо. Функция Радо $BB(n)$* определяется как максимальное количество 1, которые может оставить машина Тьюринга с n состояниями над двоичным алфавитом на изначально пустой ленте (при условии, что машина останавливается). Покажите, что функция $BB(n)$ невычислима. *Подсказка*: сначала покажите, как смоделировать машину Тьюринга с n состояниями для входных данных размера m, выполняя машину Тьюринга с $(m + n)$ состояниями с изначально пустыми данными. Затем выполните машину Тьюринга с $(m + n)$ состояниями для $BB(m + n + 1)$ шагов.

5.4.10. *Проблема остановки с пустой лентой.* В доказательстве неразрешимости проблемы остановки из упражнения 5.4.7 используется машина Тьюринга, на вход которой подается описание ее самой. Эта искусственная конструкция, ссылающаяся сама на себя, упрощает доказательство. Покажите, что проблема остановки неразрешима, даже если входная лента изначально пуста. *Подсказка*: для заданного метода решения проблемы остановки для машин Тьюринга с изначально пустой лентой покажите, как вычислить функцию Радо $BB(n)$.

5.4.11. *Непустота.* Существует ли машина Тьюринга, которая может решить, является ли язык, принимаемый заданной машиной Тьюринга, пустым? Докажите, что этот вопрос является неразрешимым.

5.4.12. *Регулярность.* Существует ли машина Тьюринга, которая может решить, является ли язык, принимаемый заданной машиной Тьюринга, регулярным? Докажите, что этот вопрос является неразрешимым.

5.5. Вычислительная сложность

В предыдущем разделе мы классифицировали задачи в зависимости от того, возможно ли в принципе решить их на компьютере. В этом разделе основное внимание будет уделяться задачам, решить которые *возможно* — и в первую очередь на вычислительных ресурсах, необходимых для их решения.

В книге были рассмотрены многочисленные алгоритмы, которые обычно используются для решения практических задач и, следовательно, должны требовать адекватных затрат ресурсов. Практическая ценность большинства алгоритмов очевидна, и во многих задачах доступна такая роскошь, как несколько эффективных алгоритмов, из которых можно выбрать наиболее подходящий. К сожалению, на практике часто встречаются другие задачи, для которых такие эффективные решения отсутствуют. Что еще хуже, для большого класса задач невозможно даже сказать, существует ли такое эффективное решение. Такое состояние дел было в высшей степени неприятным для программистов и разработчиков алгоритмов, которые не могли найти ни одного эффективного алгоритма для широкого круга практических задач, и для теоретиков, которые не могли найти никаких доказательств, что эти задачи действительно сложны.

В этой области были проведены значительные исследования. В результате были разработаны механизмы, благодаря которым новые задачи можно отнести к «трудным для решения» в конкретном техническом смысле. Хотя большая часть этой работы выходит за рамки настоящей книги, основные идеи вполне доступны. Мы рассмотрим их здесь, потому что каждый программист, сталкивающийся с новой задачей, должен хорошо понимать: существуют задачи, для которых не известен ни один алгоритм с гарантированной эффективностью нахождения решения.

В двух предыдущих разделах были представлены следующие две идеи, вдохновленные революционной работой Алана Тьюринга в 1930-е годы.

- *Универсальность.* Машина Тьюринга может выполнить любое вычисление (разрешить язык или вычислить функцию), которое может быть выполнено любым физически реализуемым вычислительным устройством. Эта идея известна под названием *тезиса Чёрча — Тьюринга.* Это утверждение о реальном мире не может быть доказано (хотя не может также и быть опровергнуто). В пользу

тезиса свидетельствует то, что математики и ученые разработали много разных моделей вычислений, но все они оказались эквивалентными машине Тьюринга.

- *Вычислимость.* Существуют задачи, которые не могут быть решены машиной Тьюринга (или любым другим физически реализуемым вычислительным устройством вследствие принципа универсальности). Это утверждение является математически истинным. Примером такой задачи служит знаменитая проблема остановки (ни одна программа не может гарантированно определить, остановится ли заданная программа).

В текущем контексте нас интересует третья идея, относящаяся к эффективности вычислительных устройств.

- *Расширенный тезис Чёрча — Тьюринга.* Машина Тьюринга может *эффективно* выполнить любое вычисление (разрешить язык или вычислить функцию), которое выполнимо любым физически реализуемым вычислительным устройством.

И снова это утверждение относится к реальному миру: оно опирается на идею о том, что все известные физически реализуемые вычислительные устройства могут быть смоделированы машиной Тьюринга, при этом затраты возрастают не более чем с полиномиальной зависимостью. Например, известно, что для любого заданного алгоритма, который можно выполнить за время, пропорциональное $T(n)$, на вашем компьютере (где n — количество входных символов), можно сконструировать машину Тьюринга, которая выполняет те же вычисления за время, пропорциональное $T(n)^2$. И наоборот, наша программа `TuringMachine` предоставляет доказательство того, что любая машина Тьюринга, выполняемая за время, пропорциональное $T(n)$, может быть смоделирована на вашем компьютере за время, пропорциональное $T(n)$. Из расширенного тезиса Чёрча — Тьюринга следует, что, в принципе, для повышения эффективности будущих компьютеров достаточно сосредоточиться только на технологии существующих типов компьютеров, а не на разработке новых типов.

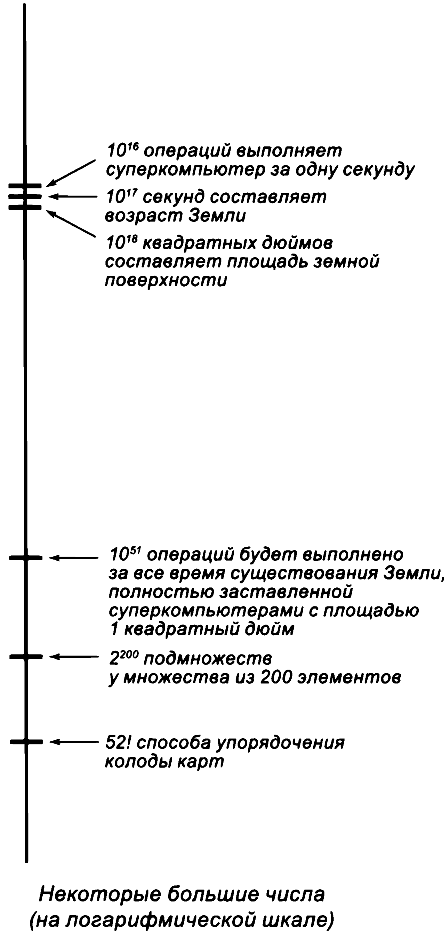
Обзор

Целью теории вычислительной сложности является отделение задач, которые могут быть решены за *полиномиальное время*, от задач, для которых, скорее всего, требуется *экспоненциальное время*. Вскоре мы определим эти термины, но первым шагом для понимания вычислительной сложности должно стать полноценное понимание природы экспоненциального роста.

Для этого понадобится проделать кое-какие вычисления на скорую руку. Не углубляясь в подробности, поработаем со следующими оценками.

- Возраст Земли составляет приблизительно 10^{17} секунд.
- Площадь земной поверхности составляет приблизительно 10^{18} квадратных дюймов.
- Современный суперкомпьютер способен выполнить приблизительно 10^{16} операций в секунду.

Перемножая эти числа, мы видим, что если установить на каждом квадратном дюйме земной поверхности по суперкомпьютеру и если все эти компьютеры будут работать параллельно в течение всего срока жизни Земли, они бы выполнили всего 10^{51} операций. Стоит уточнить, что 10^{51} — намного меньшее число, чем 52! (факториал 52), а также много меньшее 2^{200}.



Некоторые большие числа (на логарифмической шкале)

Представьте, что вы хотите вычислить некоторое свойство случайно перетасованной колоды карт и для этого собираетесь проверить все возможные варианты. Об этой идее можно забыть, потому что существуют 52! возможные перестановки и проверить их все совершенно нереально. Возможно, вас отрезвит понимание того, что доля возможных перестановок колоды карт, которую вы можете проверить, намного меньше 0,000000000000005.

На фоне экспоненциального роста меркнут любые технологические достижения: даже если суперкомпьютер считает в квадриллион раз быстрее, чем счеты, ни то ни

другое устройство и близко не подойдет к решению задачи, требующей 2^{200} шагов. Кроме того, легко можно разработать алгоритмы, которые исчерпают все доступные ресурсы в попытках выполнить такое количество шагов. Примеры в нескольких ближайших разделах убедят вас в этом.

Размер задачи

Теория направлена на то, чтобы сформулировать некоторые общие утверждения относительно широкого класса алгоритмов, а модель машины Тьюринга позволяет достаточно точно формулировать правила. Первое правило состоит в том, что *размер задачи оценивается по количеству битов в заданных входных данных* (с точностью до множителя-константы). Так как мы всегда предполагаем, что алфавит имеет постоянный размер, условие о «постоянном множителе» означает, что в модели машины Тьюринга n интерпретируется как количество символов, находящихся на входной ленте (в программе Java n рассматривается как количество символов в аргументах командной строки или в стандартном потоке ввода).

Худший случай

Вся теория в этом разделе строится на основе *анализа худшего случая*. Иначе говоря, нас интересует гарантированная производительность алгоритма для задач заданного размера независимо от значения данных на входе. Даже если алгоритм быстро работает для абсолютного большинства вариантов входных данных и медленно для немногих, он все равно считается медленным. Для такого пессимистического подхода есть две причины.

- Часто бывает проще определить верхнюю границу времени выполнения как функцию от размера входных данных, чем пытаться характеризовать их варианты каким-либо иным способом. Например, альтернативный вариант с анализом среднего случая выглядит заманчиво, но он потребует разработки вероятностной модели входных данных (что может быть достаточно сложно) и математического анализа поведения алгоритма в этой модели (что может быть еще сложнее).

- Алгоритмы с гарантированным быстродействием худшего случая — достойная цель, часто требуемая (и достигаемая!) на практике. Например, вы наверняка предпочтете, чтобы программа, используемая для управления вашим самолетом, машиной или кардиостимулятором, обеспечивала некоторое гарантированное минимальное быстродействие даже в худшем случае.

Мы подчеркиваем это обстоятельство с самого начала, потому что результаты исследований в теории вычислений часто интерпретируются неправильно. А именно, в типичных практических ситуациях *производительность худшего случая не может использоваться для прогнозирования производительности*. Для этого требуется более сложный анализ, основанный на научном подходе (см. раздел 4.1). Вместо этого мы проявляем особое внимание к худшему случаю для того, чтобы сосредоточиться на фундаментальных вопросах, касающихся вычислений.

Алгоритмы с полиномиальным временем

Вы уже знаете, что существует много задач, для которых известны эффективные алгоритмы. Первым шагом в этой теории станет попытка объединения всех этих задач в одну категорию. Чтобы избежать излишней детализации, мы упростим анализ и установим *верхнюю границу* для времени выполнения в худшем случае, начиная со следующего определения.

Определение. Алгоритм с полиномиальным временем — алгоритм, время выполнения которого как функция от размера входных данных n ограничена сверху значением $a \times n^b$ для всех вариантов данных (где a и b — положительные константы).

В контексте темы этого раздела нас интересуют не конкретные значения констант, а тот факт, что мы знаем *верхнюю границу времени выполнения алгоритма для всех вариантов входных данных*. Например, алгоритмы сортировки, рассмотренные в разделе 4.2, относятся к алгоритмам с полиномиальным временем, потому что мы доказали, что их время выполнения для худшего случая пропорционально $n \log n$ или n^2. Если для задачи существует алгоритм с полиномиальным временем, мы считаем, что она «решается легко». Наша цель — отделить «легко решаемые» задачи от «сложных», которые и рассматриваются ниже.

Алгоритмы с экспоненциальным временем

Также существует много задач, для решения которых ни один эффективный алгоритм не известен. Второй шаг теории — попытка объединения всех этих задач в одну категорию. И снова, чтобы избежать излишней детализации, мы упростим анализ и установим *нижнюю границу* для времени выполнения худшего случая, начиная со следующего определения.

Определение. Алгоритм с экспоненциальным временем — алгоритм, время выполнения которого как функция от размера входных данных n ограничивается снизу значением $2^{a \times n^b}$ для бесконечно большого числа вариантов данных (где a и b — положительные константы).

И снова нас интересуют не конкретные значения констант, а тот факт, что мы знаем *экспоненциальную нижнюю границу времени выполнения алгоритма для некоторого бесконечного класса вариантов входных данных*. Например, решение задачи о ханойских башнях и сопутствующие алгоритмы из раздела 2.3 являются экспоненциальными, так как мы доказали, что время их выполнения пропорционально 2^n. В соответствии с этим определением такое время выполнения, как $1{,}5^n$, $n!$ и $2^{\sqrt{n}}$, также считается экспоненциальным (см. упражнение 5.5.3). Если все алгоритмы, известные для задачи, имеют экспоненциальное время выполнения, то задача считается «сложной».

Если учесть колоссальный разрыв в производительности между решениями для тех задач, которые имеют алгоритмы с полиномиальным временем, и тех, для

которых лучшие известные алгоритмы требуют экспоненциального времени, может показаться, что их легко отличить друг от друга. Главный вывод из всей теории, которая рассматривается в этом разделе:— как ни странно, это определенно не так.

Примеры

Чтобы поместить эти концепции в конкретные условия, мы рассмотрим их в контексте алгоритмических задач, в которых задействовано много разных типов данных. Они закладывают фундамент для более формального обсуждения тех задач, которые решают или пытаются решить ученые, инженеры и прикладные программисты (в том числе и вы). Все эти задачи имеют многочисленные важные применения, но мы преодолеваем искушение описывать их подробно, чтобы не отвлекать вас от фундаментальной, интуитивной природы самих задач.

Числа

Для начала рассмотрим алгоритмы, обрабатывающие целые числа с произвольной точностью. Класс Java `BigInteger` упрощает обработку таких чисел. С классом `BigInteger`, например, можно воспользоваться алгоритмом перемножения двух целых чисел из n цифр с квадратичным временем (причем известны и более быстрые алгоритмы.) Однако следующая задача кажется гораздо более сложной.

Разложение на простые множители (*факторизация*). Найти разложение заданного положительного целого числа из n цифр на простые множители.

Задачу можно решить, модифицировав программу `Factors` (листинг 1.3.9) для использования `BigInteger` (см. упражнение 5.5.36). Но это решение совершенно неприемлемо для больших n, потому что количество итераций цикла для простого числа из n цифр составит приблизительно $10^{n/2}$. Например, для простого числа из 1000 цифр потребуется около 10^{500} итераций. Приемлемый способ решения этой задачи никому не известен. Кстати говоря, протокол RSA, обеспечивающий безопасность интернет-коммерции, основан на предположении о сложности разложения на простые множители.

Подмножества

Другой пример: рассмотрим *задачу суммы подмножеств*, которая обобщает задачу о сумме троек из раздела 4.1.

Сумма подмножеств. Найдите (непустое) подмножество из n заданных целых чисел, сумма которых равна 0 (или сообщите, что такого подмножества не существует).

В принципе, задачу можно решить, написав программу для перебора всех возможных ваиантов (см. упражнение 5.5.4), но эта программа не завершится за приемлемое время для больших n, потому что у множества из n элементов существуют

2^n разных подмножества. Проще всего убедиться в этом, поняв, что любое n-разрядное двоичное число соответствует некоторому подмножеству n элементов, определяемому позициями единиц. Пример приведен справа — справа от каждого двоичного числа выводится сумма соответствующего подмножества. И снова рассмотрение всех возможных вариантов полностью неприемлемо, допустим, для $n = 200$. Никому не известен алгоритм, который может гарантировать решение этой задачи для больших n (несмотря на то что конкретный экземпляр этой задачи может быть решен — например, если подмножество с нулевой суммой будет обнаружено в самом начале поиска).

Чтобы продемонстрировать, что подмножества встречаются в разнообразных ситуациях, в третьем примере мы рассмотрим следующую задачу.

Вершинное покрытие. Для заданного графа G и целого числа m найти подмножество, содержащее не более m вершин G, с которыми соприкасается каждое ребро (или сообщить, что такого подмножества не существует).

На с. 797 показан пример, в котором все возможные подмножества размера 1 и 2 помечены черным цветом и среди них выделены два решения (в которых нет ребер, соединяющих две серые вершины). Чтобы лучше понять природу приложения, представьте, что вершины — это маршрутизаторы, а ребра — каналы связи. Тогда покрытие сообщит атакующему, возможно ли полностью заблокировать передачу данных, выведя из строя m вершин (или специалисту по безопасности — достаточно ли защитить m вершин, чтобы оставить хотя бы один канал связи). В данном случае количество рассматриваемых возможностей является полиномиальным при малых m и экспоненциальным при больших m (см. упражнение 5.5.6). Вскоре мы рассмотрим программу, которая пытается решить задачу перебором всех возможностей, но эта программа *не сможет завершиться для больших n (кроме очень малых m)*.

задача:
для множества из 5 заданных целых чисел найти подмножество, сумма которого равна 0

1,342, −1,991, 231, −351, 1,000

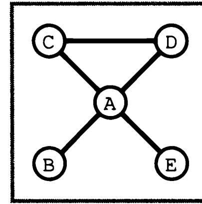
все возможные варианты

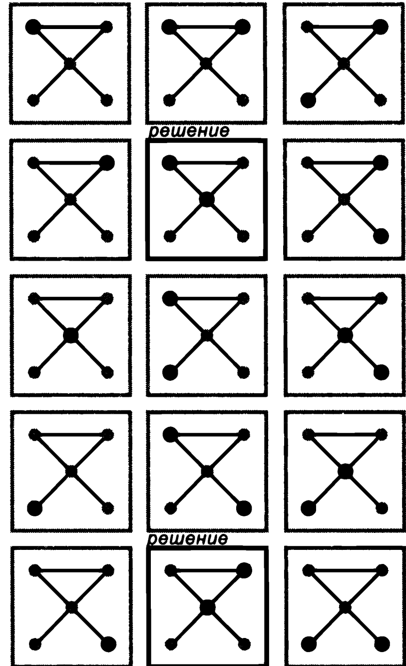
Подмножество	Сумма
0 0 0 0 1	1,000
0 0 0 1 0	−351
0 0 0 1 1	649
0 0 1 0 0	231
0 0 1 0 1	1,231
0 0 1 1 0	−120
0 0 1 1 1	880
0 1 0 0 0	−1,991
0 1 0 0 1	−991
0 1 0 1 0	−2,341
0 1 0 1 1	−1,341
0 1 1 0 0	−1,760
0 1 1 0 1	−760
0 1 1 1 0	−2,111
0 1 1 1 1	−1,111
1 0 0 0 0	1,342
1 0 0 0 1	2,342
1 0 0 1 0	991
1 0 0 1 1	1,991
1 0 1 0 0	1,673
1 0 1 0 1	2,673
1 0 1 1 0	1,322
1 0 1 1 1	2,322
1 1 0 0 0	−649
1 1 0 0 1	351
1 1 0 1 0	−1,000
1 1 0 1 1	**0**
1 1 1 0 0	−418
1 1 1 0 1	682
1 1 1 1 0	−769
1 1 1 1 1	231

решение:
1,342, −1,991, −351, 1,000

Экземпляр задачи о сумме подмножества

«Сложные» задачи





Экземпляр задачи о вершинном покрытии

В каждом из только что описанных примеров упоминалось «решение задачи перебором всех возможных вариантов». Каждый, кто пользуется компьютером для решения задач, должен понять, что этот подход часто оказывается неэффективным из-за экспоненциального роста времени выполнеия. Хорошенько подумайте об этом факте: он противоречит нашим естественным представлениям о том, что компьютеры работают настолько быстро, что они способны решить любую задачу, если у них будет достаточно времени (спросите своих друзей — хватит ли компьютеру скорости, чтобы перебрать все возможные варианты порядка карт в колоде?). Мы считаем задачу «сложной для решения», если все известные алгоритмы ее решения требуют для выполнения экпоненциального времени (см. определение на с. 794). Обычно можно считать, что алгоритм с экспоненциальным временем не решит задачу размера 1000 (допустим) за разумное время, потому что никто не дождется, пока алгоритм выполнит 2^{1000} или 1000! шагов независимо от скорости компьютера.

«Простые» задачи

С другой стороны, многие задачи, с которыми мы сталкиваемся, не требуют перебора всех возможностей (или хотя бы большей их части), что позволяет разработать алгоритмы, эффективные даже при большом размере задачи. Задача считается «простой» для решения, если мы знаем алгоритм с полиномиальным временем для ее решения в соответствии с определением на с. 794. В общем случае наша вычислительная инфраструктура строится на базе таких алгоритмов.

Тонкая грань

Иногда грань между «простыми» и «сложными» задачами оказывается очень тонкой. Вот пример.

Кратчайший путь. Требуется найти простой путь от заданной вершины s до заданной вершины t в заданном графе, содержащем не более m ребер (или сообщить о том, что такого пути не существует).

Наша программа `PathFinder` в разделе 4.1 решает оптимизационную версию этой задачи (находит кратчайший путь) и сразу выдает решение. Но мы еще не изучали алгоритмы для решения следующей задачи, которая на первый взгляд выглядит почти так же.

Самый длинный путь. Требуется найти простой (без повторов и циклов) путь от заданной вершины s до заданной вершины t в заданном графе, содержащем не менее m ребер (или сообщить о том, что такого пути не существует).

Суть проблемы в следующем: насколько нам известно, эти задачи лежат практически на разных концах спектра сложности. Поиск в ширину предоставляет решение первой задачи за *линейное время*, но все известные алгоритмы для второй задачи требуют в худшем случае *экспоненциального времени*, так как им придется проверить все пути.

Обычно когда мы понимаем, что задача относится к классу «простых», мы можем работать над улучшением алгоритмов и ожидать, что технические усовершенствования позволят справляться с экземплярами все большего размера (на практике типичная «простая» задача решается с гарантированным временем выполнения, ограниченным низкостепенной полиномиальной зависимостью от размера входных данных — например, n^2 или n^3). Если же известно, что задача является «сложной», то особенно рассчитывать на технику не стоит. Но какие задачи относятся к «простым», а какие к «сложным»? Теория, которую мы сейчас рассмотрим, поможет нам ответить на такие вопросы. С практической точки зрения она ничуть не менее важна, чем теория вычислимости.

Задачи выполнимости

Четыре конкретные задачи, называемые *задачами выполнимости*, важны для нашего обсуждения вычислительной сложности.

- *Выполнимость линейных уравнений.* Для заданного множества из n линейных уравнений с n переменными найти такой вариант присваивания переменным рациональных значений, с которым выполняются все уравнения (или сообщить, что его не существует).
- *Выполнимость линейных неравенств.* Для заданного множества из n линейных неравенств с n переменными найти такой вариант присваивания переменным рациональных значений, с которым выполняются все неравенства (или сообщить, что его не существует).
- *Выполнимость целочисленных линейных неравенств.* Для заданного множества из m линейных неравенств с n переменными найти такой вариант присваивания переменным целочисленных значений, с которым выполняются все неравенства (или сообщить, что его не существует).

- *Выполнимость логических (булевых) уравнений.* Для заданного множества из m уравнений с n логическими переменными найти такой вариант присваивания переменным значений, с которым выполняются все уравнения (или сообщить, что его не существует).

Все эти задачи часто встречаются на практике; они сыграли важную роль в течение нескольких последних десятилетий, когда компьютеры стали привычным явлением в промышленных и коммерческих областях. Как вы сейчас увидите, несмотря на внешнее сходство, проблемы, возникающие при решении этих задач, на удивление сильно различаются по своей природе.

Выполнимость линейных уравнений

Вероятно, эта задача знакома вам под названием «решения системы уравнений». Алгоритмы для ее решения, обычно называемые *методом исключения Гаусса*, восходят корнями к Древнему Китаю, и их преподают на уроках алгебры с XVIII века. Базовый алгоритм реализуется просто (см. упражнение 5.5.2) и присутствует в стандартных библиотеках численных методов линейной алгебры. Проверка того, что исключение Гаусса работает правильно для всех вариантов входных данных, когда переменные содержат значения типа `double`, создает практические трудности (поэтому рекомендуется использовать библиотечную реализацию). Не все версии исключения Гаусса являются полиномиальными, поскольку объем промежуточных вычислений может расти экспоненциально, но для некоторых стандартных версий полиномиальность была доказана (еще одна причина для использования библиотечной версии). Несмотря на эти технические трудности, эту задачу можно считать «простой».

задача:

$$4x - 2y - z = -1$$

$$4x + 4y + 10z = 9$$

$$12x + 4y + 8z = 21$$

решение:

$$x = 5/4,\ y = 7/2,\ z = -1$$

Экземпляр задачи выполнимости линейных уравнений

Выполнимость линейных неравенств

Теперь предположим, что в системе уравнений могут приступтствовать не только равенства, но и неравенства (причем неравенств может быть больше, чем переменных). Это изменение приводит к разновидности классической задачи, называемой *задачей линейного программирования* (LP, Linear Programming). Теория линейного программирования была разработана в середине XX века для планирования обеспечения войск во время войны, а знаменитый алгоритм для решения задачи, называемый *симплекс-методом*, был изобретен Джорджем Данцигом в 1947 году и с тех

задача:

$$6x - 10y - z \le 0$$

$$2x + 2y + 5z \le 76$$

$$3x + y + 2z \le 54$$

$$x, y, z \ge 0$$

решение:

$$x = 10,\ y = 51/10,\ z = 9$$

Экземпляр задачи выполнимости линейных неравенств (поисковая формулировка ЛП)

пор успешно применяется. Симплекс-метод намного сложнее исключения Гаусса, но после некоторого изучения материала вы поймете идеи, на которых он основан; кроме того, его реализации широко доступны. Однако симплекс-метод может потребовать экспоненциального времени (или хуже), и вопрос о том, существует ли для задач линейного программирования алгоритм с полиномиальным временем, оставался открытым несколько десятилетий, вплоть до середины 1980-х. Формально эта задача сегодня считается «простой». Современные реализации очень широко применяются во многих промышленных и управляющих приложениях. График ваших авиаперелетов или маршрут доставки экспресс-почты, вероятно, был частью решения задачи линейного программирования — в которой, возможно, были задействованы сотни тысяч неравенств.

Выполнимость целочисленных линейных неравенств

Если переменные в задаче линейного программирования представляют пилотов или грузовики (например), появляется необходимое требование о том, чтобы значения в решениях были целыми числами. Эта задача называется задачей *целочисленного линейного программирования* (ЦЛП). В некоторых приложениях переменные могут принимать только значения 0 или 1: эта версия называется 0/1 ЦЛП. Как ни странно, алгоритм с полиномиальным временем для таких задач неизвестен. Один из подходов к ЦЛП основан на решении идентичной задачи ЛП для получения дробного решения с последующим округлением его до ближайшего целого числа — этот прием может стать отправной точкой для поиска решения, но он работает не всегда. Существуют пакеты для решения экземпляров задач ЦЛП, встречающихся на практике; такие пакеты широко используются в различных промышленных и коммерческих приложениях. Однако встречаются экземпляры задачи, на которых эти пакеты работают слишком медленно, поэтому ученые продолжают поиск более быстрых алгоритмов. Различия между ЛП и ЦЛП, насколько нам известно, представляют собой «тонкую грань» между задачей, которую можно решить за полиномиальное время, и задачей, которая, по всей видимости, требует экспоненциального времени. Вопрос о том, является ли задача ЦЛП «простой» или «сложной», оставался открытым десятилетиями[1].

задача:

$$7x - 10y - z \le 1$$
$$2x + 2y + 5z \le 77$$
$$3x + y + 2z \le 54$$
$$x, y, z \ge 0$$

решение:

$$x = 10,\ y = 6,\ z = 9$$

Экземпляр задачи выполнимости целочисленных линейных неравенств (поисковая формулировка ЦЛП).

[1] Экспоненциальность времени решения задачи ЦЛП считается доказанной в общем случае. При этом существуют более эффективные эвристические, а в некоторых частных случаях и точные методы решения. — *Примеч. науч. ред.*

Выполнимость логических (булевых) уравнений

Если система состоит из логических (булевых) уравнений, мы имеем дело с задачей *выполнимости логических уравнений*. Если вы не знакомы с булевой алгеброй или хотите освежить память, обратитесь к началу раздела 7.1. Переменные принимают одно из двух значений — истинное или ложное, и с ними используются всего три операции: x' означает отрицание x (ложно, если значение x истинно, и истинно, если значение x ложно); $x + y$ означает «x или y» (ложно, если оба значения x и y ложны, и истинно во всех остальных случаях); $x\,y$ означает «x и y» (истинно, если оба значения x и y истинны, и ложно во всех остальных случаях). Пример показан внизу страницы. Без потери общности будем считать, что правая часть всегда истинна, а в левой части каждого уравнения не используется операция «и» (см. упражнение 5.5.8). Также можно использовать сокращенную запись: одно уравнение, в котором все левые части объединены операцией «и». Также внизу приведена таблица со всеми возможными значениями переменных и значениями всех выражений в левой части, из которой следуют решения (две строки, в которых все значения истинны).

задача:

$x' + z = true$

$x + y' + z = true$

$x + y = true$

$x' + y' = true$

сокращенная запись:

$s = (x' + z)\,(x + y' + z)\,(x + y)\,(x' + y') = true$

все возможные варианты (T — истина, F — ложь)

x	y	z	(x'+ z)	(x + y' + z)	(x + y)	(x'+ y')	s
F	F	F	T	T	F	T	F
F	F	T	T	T	F	T	F
F	T	F	T	F	T	T	F
F	T	T	T	T	T	T	T
T	F	F	F	T	T	T	F
T	F	T	T	T	T	T	T
T	T	F	F	T	T	F	F
T	T	T	T	T	T	F	F

решения:

x = false	x = true
y = true	y = false
z = true	z = true

Экземпляр задачи логической выполнимости

Задача выполнимости логических уравнений может показаться абстрактной математической задачей, но она находит много важных применений. Например, она может моделировать поведение электронных схем и потому играет важную роль в проектировании современных компьютеров. Как и в случае с ЦЛП, были разработаны алгоритмы, которые достаточно хорошо работают для экземпляров задач, встречающихся на практике. По оценке Дональда Кнута, системы решения задачи выполнимости логических уравнений формируют отрасль с миллиардными оборотами. Но, как и в случае с ЦЛП, алгоритм с гарантированным полиномиальным временем неизвестен — каждая из систем решения для некоторого класса экземпляров задачи выполняется за экспоненциальное время.

Задачи выполнимости напрямую связаны с множеством разнообразных практических применений. Задачи легко формулируются и часто применяются, но определить грань между «простыми» и «сложными» задачами чрезвычайно сложно. Об этой проблеме было известно с первых дней вычислительных технологий, и она стимулировала развитие теоретической базы, которую мы сейчас опишем.

Задачи поиска

Серьезные различия между «простыми» задачами (вроде тех, что решаются в программах этой книги) и «сложными» (для которых решение приходится искать среди потенциально огромного количества вариантов) позволяют переход от одних к другим с помощью простой формальной модели. Первым шагом должно стать описание типа задачи, которую вы изучаете.

Определение. *Задача поиска* — задача, решения которой обладают следующим свойством: существует алгоритм с полиномиальным временем, который может *проверить*, что заданное решение действительно является решением заданного экземпляра задачи. Мы говорим, что алгоритм *решает* задачу поиска, если для любых входных данных он либо выдает решение, либо сообщает, что такого решения не существует.

Все задачи, упоминавшиеся до настоящего момента в этом разделе (сортировка, разложение на множители, умножение, кратчайший путь, самый длинный путь, выполнимость логических уравнений и т. д.), являются задачами поиска. Чтобы установить, что задача является задачей поиска, достаточно показать, что любое решение обладает достаточно четкими характеристиками, чтобы мы могли эффективно проверить его корректность. Решение задачи поиска напоминает поиск «иголки в стоге сена» с одним условием: вы должны опознать иголку, когда она вам попадется. Например, если вам дается потенциальное решение в задаче выполнимости логических уравнений, выполнимость каждого равенства или неравенства[1]

[1] Ввиду особенностей булевой алгебры логические неравенства обычно не рассматриваются. Вероятно, авторы имели в виду категорию задач выполнимости с целочисленными или рациональными значениями. — *Примеч. науч. ред.*

проверяется легко, хотя, как вы вскоре увидите, поиск (или определение факта существования) такого решения — совсем другая задача.

Для описания задач поиска часто используется сокращение **NP**. Многие полагают, что оно происходит от «Not Polynomial» («не полиномиальный»), но это не так — причины для выбора названия описаны на с. 794.

Определение. NP — множество всех задач поиска.

NP представляет собой не что иное, как точное описание всех задач, которые ученые, инженеры и прикладные программисты могут рассчитывать решить с помощью программ, выполняемых за разумное время. Примеры **NP**-задач, упоминавшихся ранее, представлены на с. 796. Затем некоторые примеры будут рассмотрены более подробно.

Задача о сумме подмножеств в NP

i	values[i]	inSubset[i]
0	1342	true
1	-1991	true
2	231	false
3	-351	true
4	1000	true

Чтобы доказать, что задача является задачей поиска, достаточно привести код Java, который может проверить (за полиномиальное время), является ли предполагаемое решение корректным решением задачи для заданных входных данных. Например, представьте, что входные данные для задачи о сумме подмножеств хранятся в целочисленном массиве `values[]`, а в массиве `inSubset[]` типа `boolean` элементы `true` соответствуют значениям из рассматриваемого подмножества. Значения из примера на с. 796> содержатся в верхней таблице. При таком представлении код Java для проверки того, равна ли сумма подмножеств 0, выглядит просто:

```
public static boolean check(int[] values, boolean[] inSubset)
{
   int sum = 0;
   for (int i = 0; i < n; i++)
      if (inSubset[i])
         sum += values[i];
   return sum == 0;
}
```

В данном случае проверка решения выполняется за один проход по данным с линейным временем. Это типично для задач, входящих в **NP**, и показывает, что задачи

из **NP** — это те задачи, для которых решение можно распознать, когда вы его уже увидели!

Задача о вершинном покрытии принадлежит NP

С помощью аналогичного механизма можно доказать, что задача о вершинном покрытии также входит в NP. Предположим, на вход подается граф `G` (с использованием типа данных `Graph` из раздела 4.1) и целое число `m`, задающее верхнюю границу размера подмножества. Решение задачи о вершинном покрытии может быть представлено массивом `inSubset[]` с элементами `boolean`, в котором элементы `true` соответствуют вершинам из покрытия. Чтобы проверить, является ли подмножество вершин покрытием, необходимо:

- убедиться в том, что количество вершин в подмножестве (количество элементов `inSubset[]`, равных `true`) меньше либо равно `m`;
- проверить все *ребра* графа и убедиться в том, что ни одно ребро не соединяет две вершины, не входящие в подмножество.

Такие проверки реализуются просто (см. упражнение 5.5.7). И снова затраты на них линейно зависят от размера входных данных.

Задача 0/1 ЦЛП принадлежит NP

Теперь установим, что задача 0/1 ЦЛП входит в **NP**. Предположим, на вход подается матрица $m \times n$ `a[][]` и вектор-столбец свободных членов `b[]` длины `m`. Решение будет представлено вектором `x[]` длины `n`. Чтобы проверить, действительно ли вектор предполагаемого решения `x[]` является решением, необходимо:

- убедиться в том, что каждый элемент `x[j]` равен 0 или 1;
- проверить, что для каждого неравенства `i` выполняется условие

```
a[i][0]*x[0] + a[i][1]*x[1] + … + a[i][n-1]x[n-1] <= b[i]
```

Эти проверки легко выполняются за время, линейно зависящее от размера входных данных (см. упражнение 5.5.11). Очень похожие рассуждения позволяют доказать, что задача выполнимости логических уравнений тоже принадлежит **NP**.

Тем не менее доказательство принадлежности задачи ЦЛП к **NP** требует более сложных рассуждений (выходящих за рамки книги), потому что значения в решении задачи ЦЛП (без ограничения 0/1) теоретически могут быть экспоненциально большими.

Нахождение решения

Конечно, основные вычислительные затраты при решении задачи поиска связаны с тем, чтобы обнаружить само решение. Как уже упоминалось ранее, для многих задач не известно ничего лучше простого перебора всех возможных решений. Пример

такого рода для задачи выполнимости будет рассмотрен позднее в этом разделе (см. листинг 5.5.1). Фактически программе нужно перебрать все 2^n возможных набора значений размера n, а это требует экспоненциального времени.

Использование задач поиска для определения **NP** — всего лишь один из трех часто встречающихся в литературе способов описания множества задач, закладывающего основу для изучения вычислительной сложности. Два других способа связаны с задачами принятия решения (существует ли решение задачи?) и задачами оптимизации (какое решение является лучшим?) Например, задачу вершинного покрытия можно представить тремя способами. Для заданного графа G имеются следующие три задачи.

- *Оптимизация*: найти вершинное покрытие G с наименьшим количеством вершин.
- *Принятие решения*: имеет ли граф G вершинное покрытие, состоящее не более чем из m вершин?
- *Поиск*: найти вершинное покрытие G, содержащее не более m вершин.

Хотя эти три подхода не являются технически эквивалентными, соотношения между ними в контексте определения NP хорошо изучены, и основные выводы применимы ко всем трем типам задач. Чтобы избежать недоразумений, мы сосредоточимся исключительно на задачах поиска.

Используя название «ЛП», мы в действительности имеем в виду «поисковую формулировку ЛП». С этого момента мы будем использовать термины *«принадлежит NP»* и *«задача поиска»* как синонимы без дополнительных комментариев, поэтому очень важно, чтобы вы тщательно усвоили эту идею сейчас.

Недетерминированность

N в «**NP**» означает *«недетерминированность»*. Речь идет о том, что один (теоретический) путь расширения возможностей компьютера связан с использованием недетерминированности: мы предполагаем, что, когда алгоритм сталкивается с выбором одного из нескольких вариантов, компьютер может «угадать» правильный вариант. Возможно, недетерминированность — всего лишь математическая абстракция, но эта идея полезна (например, в разделе 5.1 было показано, что недетерминированность может пригодиться при проектировании алгоритмов — задача распознавания регулярных выражений может эффективно решаться моделированием недетерминированной машины).

Чтобы кто-либо согласился с тем, что найденное решение действительно является корректным, мы должны иметь возможность проверить его. В контексте обсуждения можно представить алгоритм недетерминированной машины как «угадывание» решения с последующей проверкой его правильности. В машине Тьюринга недетерминированность представляет собой простое определение двух последующих состояний для заданного состояния и заданных входных данных и описание решений

как всех действительных путей к желательному результату.

Например, мы можем построить недетерминированную машину Тьюринга, которая решает задачу выполнимости логических уравнений, подключая изображенную справа машину в начало системы проверки выполнимости и запуская ее для ленты, инициализированной экземпляром задачи, с установкой каретки в левый конец. Недетерминированная часть машины просто помещает решение на ленту, угадывая правильное значение каждой переменной. Затем (детерминированная) система проверки выполнимости вроде той, что описана в упражнении 5.5.38, проверяет ответ. Если идея угадывающей машины кажется вам сомнительной, недетерминированность можно рассматривать иначе: недетерминированная машина принимает входную строку в том и только в том случае, если в машине *существует некоторый путь от исходного состояния в состояние Yes*. В таком случае ответ ясен: если уравнение выполнимо, то существует путь через недетерминированную часть, который записывает на ленту подходящие значения, и путь через детерминированную часть, который завершается в состоянии Yes.

Недетерминированная машина Тьюринга для решения задачи выполнимости логических уравнений

На первый взгляд идея о том, что машина может угадать ответ, кажется фантастической, но она перестает казаться чем-то невероятным, если задуматься о преобразовании недетерминированной машины Тьюринга в детерминированную, как это делалось для НКА на с. 721. Конечным результатом является детерминированная машина, которая опробует и проверяет все 2^n возможных входных значения. Любая задача из **NP** может быть решена за экспоненциальное время (по аналогии). Критическим фактором является не сама возможность нахождения решения, а *затраты*[1]. Существует ли детерминированная машина Тьюринга, которая может найти и проверить решение за *полиномиальное время*?

Следующее определение эквивалентно приведенному ранее: «**NP** — множество всех задач, которые могут быть решены (и проверены) за полиномиальное время

[1] Упрощенно можно сказать так: недетерминированная машина способна анализировать несколько (множество) потенциальных решений одновременно; следовательно, ее можно представить как множество «обычных» детерминированных машин, действующих параллельно. — *Примеч. науч. ред.*

на недетерминированной машине Тьюринга». Как вы увидите, эта характеристика необходима для доказательства некоторых свойств **NP**.

«Простые» задачи поиска

Определение **NP** ничего не говорит о трудности нахождения решения; оно только проверяет, что предполагаемое решение верно. Второе из двух множеств задач, лежащих в основе изучения вычислительной сложности, называется **P**; оно связано со сложностью поиска решения.

Определение. P — множество всех задач поиска, которые могут быть решены за полиномиальное время.

Чтобы задача принадлежала **P**, должен существовать алгоритм с полиномиальным временем для ее решения. С математической точки зрения под «алгоритмом» мы понимаем «детерминированную машину Тьюринга», а под «полиномиальным временем» — «ограничена полиномиальной функцией от количества битов на входной ленте». Параметры полиномиальности вообще не задаются — мы только пытаемся отделить ее от экспоненциального времени (с. 794).

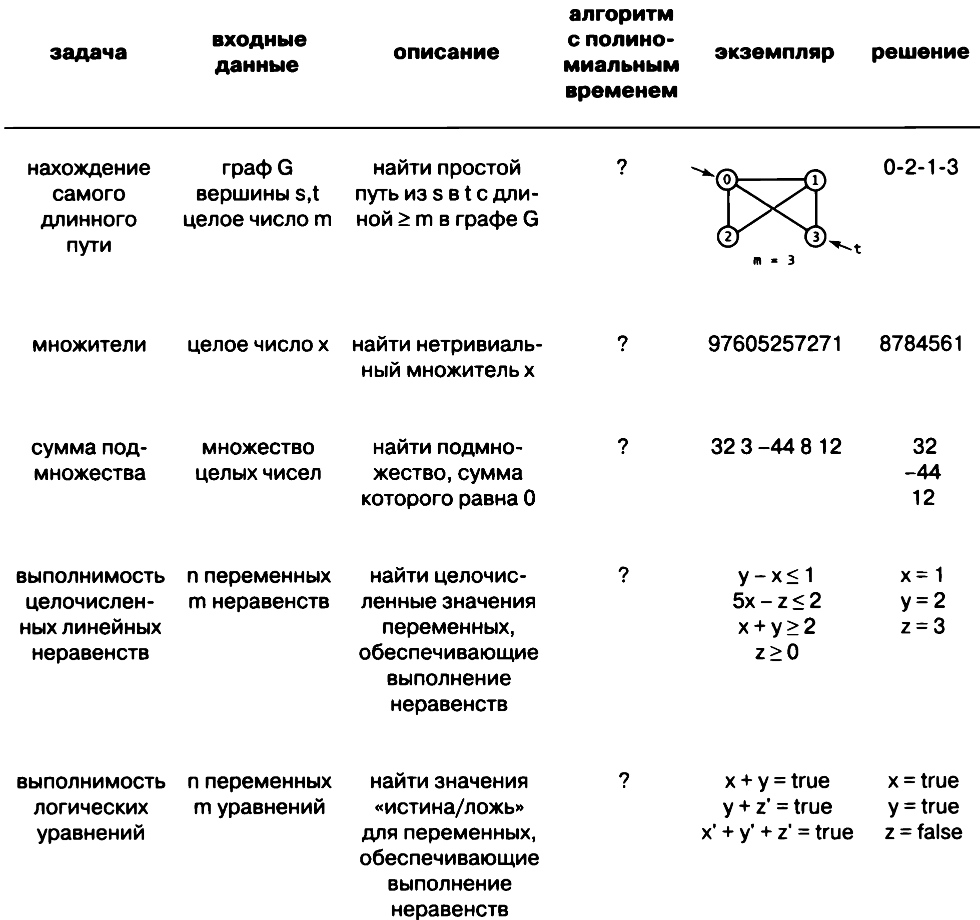
Чтобы доказать, что задача принадлежит **P**, достаточно предоставить программу Java, которая решает эту задачу и выполняется за полиномиальное время. Сортировка относится к множеству **P**, потому что (например) сортировка методом вставки выполняется за время, пропорциональное n^2 (существование более быстрых алгоритмов в данном контексте несущественно); то же самое можно сказать о задаче поиска кратчайшего пути, задаче выполнимости линейных уравнений и многих других. Линейные, линейно-логарифмические, квадратичные и кубические алгоритмы относятся к категории алгоритмов с полиномиальным временем, так что это определение, безусловно, распространяется на классические алгоритмы, рассматривавшиеся ранее.

Наличие эффективного алгоритма для решения задачи поиска — доказательство того, что задача принадлежит к множеству **P**. Другими словами, **P** — не что иное, как точная характеристика всех задач, которые решаются учеными, инженерами и прикладными программистами при помощи программ, выполняемых за приемлемое время. Примеры задач из **P**, рассмотренных ранее, приведены на с. 808.

«Сложные» задачи поиска

Если задача поиска не принадлежит **P**, это значит, что для ее решения не существует алгоритма с полиномиальным временем. Такие задачи мы будем называть «вычислительно сложными».

Определение. Задача называется *вычислительно сложной*, если для ее решения не существует алгоритма с полиномиальным временем.

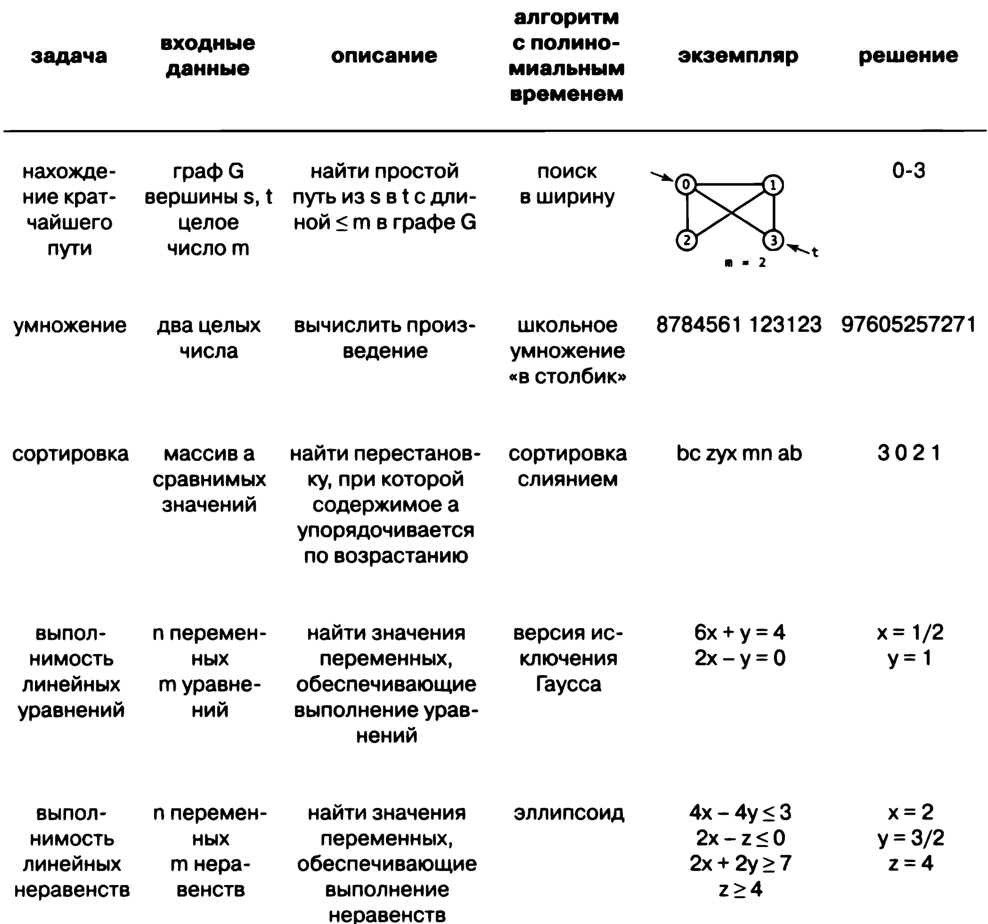


задача	входные данные	описание	алгоритм с полиномиальным временем	экземпляр	решение
нахождение самого длинного пути	граф G вершины s,t целое число m	найти простой путь из s в t с длиной ≥ m в графе G	?	0 1 2 3 t m = 3	0-2-1-3
множители	целое число x	найти нетривиальный множитель x	?	97605257271	8784561
сумма подмножества	множество целых чисел	найти подмножество, сумма которого равна 0	?	32 3 −44 8 12	32 −44 12
выполнимость целочисленных линейных неравенств	n переменных m неравенств	найти целочисленные значения переменных, обеспечивающие выполнение неравенств	?	$y - x \leq 1$ $5x - z \leq 2$ $x + y \geq 2$ $z \geq 0$	x = 1 y = 2 z = 3
выполнимость логических уравнений	n переменных m уравнений	найти значения «истина/ложь» для переменных, обеспечивающие выполнение неравенств	?	x + y = true y + z' = true x' + y' + z' = true	x = true y = true z = false

Примеры задач из множества NP

Если задача является вычислительно сложной, мы не можем гарантировать, что она будет решена за приемлемое время (при сколько-нибудь значительном размере входных данных). В этом разделе мы представим несколько знаменитых задач поиска, которые считаются вычислительно сложными.

Нетрудно представить, как математики середины XX века (и немногочисленные специалисты по информатике, жившие в то время) формулировали эти классы задач в надежде, что это поможет им выявлять вычислительно сложные задачи и избегать их (как в случае с нерешаемыми задачами). Скорее всего, они и не подозревали, что спустя 50 лет мы не только не научимся это делать, но даже не будем знать, возможно ли это.



задача	входные данные	описание	алгоритм с полиномиальным временем	экземпляр	решение
нахождение кратчайшего пути	граф G вершины s, t целое число m	найти простой путь из s в t с длиной ≤ m в графе G	поиск в ширину	m = 2	0-3
умножение	два целых числа	вычислить произведение	школьное умножение «в столбик»	8784561 123123	97605257271
сортировка	массив a сравнимых значений	найти перестановку, при которой содержимое a упорядочивается по возрастанию	сортировка слиянием	bc zyx mn ab	3 0 2 1
выполнимость линейных уравнений	n переменных m уравнений	найти значения переменных, обеспечивающие выполнение уравнений	версия исключения Гаусса	$6x + y = 4$ $2x - y = 0$	$x = 1/2$ $y = 1$
выполнимость линейных неравенств	n переменных m неравенств	найти значения переменных, обеспечивающие выполнение неравенств	эллипсоид	$4x - 4y \leq 3$ $2x - z \leq 0$ $2x + 2y \geq 7$ $z \geq 4$	$x = 2$ $y = 3/2$ $z = 4$

Примеры задач из множества P

Главный вопрос

Недетерминированность кажется фантазией, недостижимой в реальном мире, и рассматривать ее серьезно на первый взгляд кажется абсурдным. Зачем рассматривать воображаемый инструмент, с которым сложные задачи кажутся тривиальными? Дело в том, что, какие бы выгоды ни обещала недетерминированность, никому не удалось доказать, что недетерминированная машина будет значительно быстрее детерминированной для любой конкретной задачи поиска! Иначе говоря, нам, безусловно, хотелось бы знать, какие задачи поиска принадлежат **P**, а какие являются вычислительно сложными, но никому еще не удалось найти задачу, которая

бы гарантированно принадлежала **NP**, но не **P** (или даже доказать, что такая задача существует), поэтому следующий фундаментальный вопрос остается открытым:

P = NP?

Этот вопрос был точно сформулирован в такой форме С. Куком в 1971 году, хотя некоторые ученые довольно близко подходили к нему — в частности, в знаменитом «потерянном письме» К. Гёделя к Джону фон Нейману в 1956 году (а также в письме Джона Нэша в Агентство национальной безопасности США от 1955 года, рассекреченном в 2012 году). С того времени этот вопрос ставит в тупик математиков и специалистов по информатике. Другие формулировки того же вопроса проливают свет на его фундаментальную природу.

- Существуют ли вычислительно сложные задачи поиска?
- Является ли решение методом «грубой силы» лучшим возможным для некоторых задач поиска?
- Является ли поиск решения задачи поиска фундаментально более сложным, чем проверка действительности предполагаемого решения?
- Будет ли решение некоторых задач поиска на недетерминированном вычислительном устройстве принципиально более эффективным, чем на детерминированном вычислительном устройстве?

Не знать ответы на эти вопросы крайне досадно, потому что многие важные практические задачи принадлежат **NP**, но могут принадлежать или не принадлежать **P** (лучшие известные алгоритмы являются экспоненциальными). Если можно доказать, что задача поиска является вычислительно сложной (не принадлежит **P**), поиски эффективного решения для нее можно прекратить. При отсутствии такого

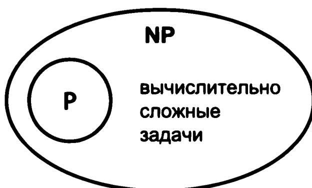
P ≠ NP

Некоторые задачи поиска являются вычислительно сложными.

Решение методом «грубой силы» может быть лучшим из возможных для некоторых задач поиска.

Найти решение некоторых задач поиска труднее, чем проверить предложенное решение.

Недетерминированность поможет нам более эффективно решать некоторые задачи поиска.



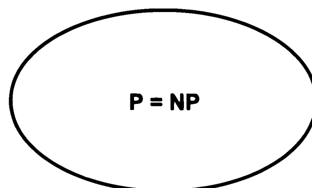
P = NP

Все задачи поиска не являются вычислительно сложными.

Эффективные алгоритмы существуют для задач ЦЛП, разложения на множители, выполнимости логических уравнений и для всех задач из NP

Найти решение задачи поиска не сложнее, чем проверить предложенное решение.

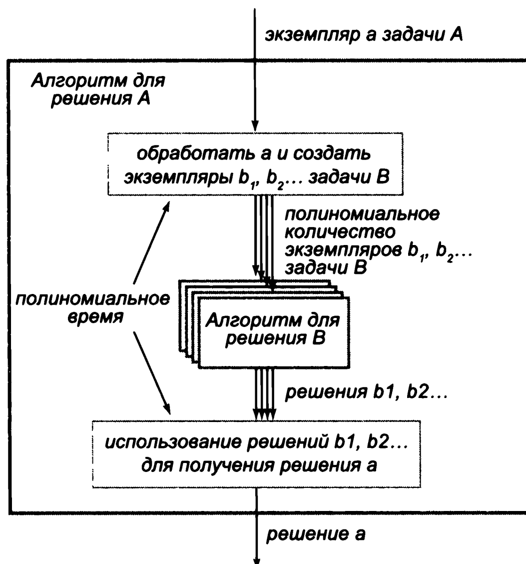
Недетерминированность не принесет особой пользы в решении задач поиска.



доказательства есть вероятность того, что эффективный алгоритм существует, но еще не найден. Собственно, при текущем состоянии наших знаний эффективный алгоритм может существовать у *каждой* задачи из **NP**; это означало бы, что многие эффективные алгоритмы еще не найдены. Мы живем в одной из двух возможных Вселенных. Но мы не имеем представления, в какой именно! Есть те, кто полагают, что P = NP; в то же время значительные усилия были приложены для доказательства обратного. Эта задача до сих пор остается открытой.

Сведение к полиномиальному времени

Ключом к прогрессу в понимании вопроса **P = NP**? является идея сведения к полиномиальному времени. Вспомните, о чем говорилось в разделе 5.4 (с. 779): мы говорим, что задача *A* сводится к задаче *B*, если любой экземпляр *A* можно решить выполнением некоторого числа стандартных вычислительных операций и некоторым количеством вызовов подпрограммы для решения экземпляров *B*. В контексте этого раздела количество вызовов подпрограммы и время, проводимое за ее пределами, ограничиваются полиномиальной зависимостью от размера входных данных. Соответственно если задача *A* сводится за полиномиальное время к задаче *B* и задачу *B* можно решить с полиномиальным временем, то и *A* тоже можно решить с полиномиальным временем. Эта разновидность сведения называется *сведением по Куку* — в отличие от *сведения по Карпу*, устанавливающего более жесткие ограничения. Общие выводы остаются справедливыми для обоих определений.



Сведение с полиномиальным временем

Определение. Задача *A* сводится с полиномиальным временем к задаче *B*, если существует алгоритм А, который использует полиномиальное количество вызовов подпрограммы для *B* и проводит полиномиальное время за пределами этих вызовов подпрограммы.

Во всех наших доказательствах используется всего один экземпляр *B* как в разделе 5.4, но теория допускает использование полиномиального количества экземпляров при условии, что вся обработка по преобразованию входных данных *A* во входные данные *B* и решений *B* в решения *A* (и все остальное) также ограничивается полиномиальным временем.

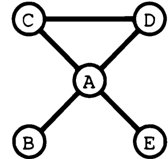
Если задача *A* сводится к задаче *B* с полиномиальным временем, то алгоритм с полиномиальным временем задачи *B* предоставляет алгоритм с полиномиальным временем для *A*. Разумеется, эта концепция знакома вам из области разработки ПО: используя библиотечный метод для решения задачи, вы тем самым сводите свою задачу к другой, уже решенной библиотечным методом (обычно с полиномиальным временем). Некоторые задачи важны тем, что к ним сводятся (с полиномиальным временем) многие другие задачи, в том числе имеющие с ними на первый взгляд мало общего. Рассмотрим пример.

Утверждение С. Задача о вершинном покрытии сводится к задаче 0/1 ЦЛП.

Доказательство. Для заданного экземпляра задачи о вершинном покрытии определите множество неравенств таким образом, что каждой вершине исходного графа соответствует одна переменная, а каждому ребру — одно неравенство, как показано справа. Переменные со значением 1 соответствуют вершинам в покрытии — единственный способ удовлетворения всех неравенств ребер заключается в присваивании значения 1 как минимум одной из двух переменных в каждом неравенстве. Чтобы решить экземпляр задачи о вершинном покрытии, решите экземпляр задачи 0/1 ЦЛП и преобразуйте решение в решение задачи о вершинном покрытии, помещая каждую вершину, у которой соответствующая целочисленная переменная равна 1, в вершинное покрытие.

Задача о вершинном покрытии

найти ≤ 2 вершин, соприкасающихся со всеми ребрами



Формулировка задачи 0/1 ЦЛП

найти 0/1 значений $x_A, \ldots x_E$ удовлетворяющих неравенствам

$x_A + x_B \geq 1$
$x_A + x_C \geq 1$
$x_A + x_D \geq 1$
$x_A + x_E \geq 1$
$x_C + x_D \geq 1$

Решения задачи 0/1 ЦЛП

$x_A = 1$	$x_A = 1$
$x_B = 0$	$x_B = 0$
$x_C = 1$	$x_C = 0$
$x_D = 0$	$x_D = 1$
$x_E = 0$	$x_E = 0$

Решения задачи о вершинном покрытии

{ A, C }   { A, D }

Сведение задачи о вершинном покрытии к задаче 0/1 ЦЛП

Задачи выполнимости часто допускают подобные сведения к ним других задач. Именно поэтому системы их решения так широко применяются на практике, а вопрос об их принадлежности **P** настолько важен. Сведение *не дает* алгоритм с полиномиальным временем выполнения для *задачи о вершинном покрытии*, но оно устанавливает

связь между этими задачами. Такие связи закладывают основу для теории вычислительной сложности.

Чтобы проверить ваше понимание сведения, попробуйте убедиться в том, что следующие три факта истинны.

- Если задача A за полиномиальное время сводится к B, а задача B принадлежит **P**, то A принадлежит **P**.
- Любая задача в **P** за полиномиальное время сводится вообще к любой задаче.
- Сведение с полиномиальным временем *транзитивно*: если задача A за полиномиальное время сводится к B, а задача B сводится к C, то A сводится к C.

Первый факт легко доказывается суммированием затрат на решение A. Второй факт обосновывается тривиально, то есть приходя к нулевому числу экземпляров. Например, сортировка сводится к задаче остановки; таким образом, если мы сможем решить задачу остановки за полиномиальное время, задача сортировки может быть решена за полиномиальное время. Но так как мы знаем, что сортировка может быть выполнена за полиномиальное время, вопрос о возможности решения задачи остановки становится несущественным. Доказательство транзитивности остается читателю для самостоятельной работы (см. упражнение 5.5.30).

NP-полнота

Многие задачи точно принадлежат **NP**, но об их принадлежности **P** ничего не известно. А это означает, что мы можем легко проверить любое заданное решение, но, несмотря на все старания, никому еще не удалось разработать эффективный алгоритм для поиска решения. Интересно, что многие задачи обладают дополнительным свойством, которое убедительно свидетельствует в пользу того, что все они являются вычислительно сложными и **P** ≠ **NP**. Начнем с определения этого свойства.

Определение. Задача поиска B является **NP**-*полной*, если каждая задача поиска A сводится к B.

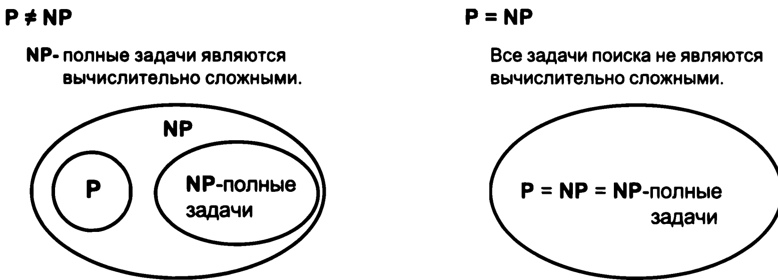
Это исключительно сильное утверждение о задаче. Если задача является **NP**-полной, то из факта существования решающего ее алгоритма с полиномиальным временем (доказательство того, что задача принадлежит P) будет следовать алгоритм с полиномиальным временем для каждой задачи в **NP**, что будет означать, что **P** равно **NP**.

Это определение позволяет нам обновить определение «сложной» задачи и наделить его смыслом «вычислительно сложной, если не выполняется условие **P** = **NP**». В этом смысле **NP**-полные задачи являются самыми сложными задачами поиска. Если *хотя бы одна* **NP**-полная задача может быть решена с полиномиальным временем, то это возможно для *всех задач в* **NP** (другими словами, **P** = **NP**). Таким

образом, коллективная неудача всех исследователей в поиске эффективных алгоритмов для всех таких задач может рассматриваться как коллективная неудача в доказательстве того, что **P** = **NP**.

NP-полнота может показаться чем-то абстрактным (как и недетерминированность), но, как вы увидите, мы можем *доказать*, что практически все задачи поиска, возникающие на практике, являются либо **P**-, либо **NP**-полными. Большинство исследователей, полагая, что **P** ≠ **NP**, воспринимает доказательство **NP**-полноты как доказательство вычислительной сложности задачи и отказывается от поиска алгоритма с полиномиальным временем.

Мы все еще живем в одной из двух возможных Вселенных, но, по крайней мере, уже знаем, в чем заключается суть проблем. *Если задача является* **NP***-полной, разумно предположить, что она является вычислительно сложной.*



Две возможные вселенные (новая версия)

Доказательство NP-сложности задач

Концепция NP-полноты, как и концепция вычислительной сложности, не принесет особой пользы, если мы не будем знать, что некоторые задачи являются NP-полными. Но здесь все меняется! Мы знаем, что многие естественные задачи являются NP-полными, и можем относительно легко идентифицировать новые задачи как NP-полные. Ситуация аналогична той, которая была продемонстрирована для вычислимости (см. раздел 5.4). Трудности возникают с доказательством NP-полноты первой такой задачи.

Теорема (Кук — Левин, 1971). Задача выполнимости логических уравнений является NP-полной.

Предельно сокращенная схема доказательства. Наша цель — показать, что если существует алгоритм с полиномиальным временем для задачи выполнимости логических уравнений, то все задачи в **NP** могут решаться с полиномиальным временем. Для любого экземпляра любой задачи в **NP** доказательство должно начинаться с построения недетерминированной машины Тьюринга для решения задачи, что возможно по определению **NP**. Затем доказательство демонстрирует способ описания каждой

возможности машины Тьюринга в логических формулах (вроде тех, которые встречаются в задаче выполнимости логических уравнений). Это описание устанавливает соответствие между каждым экземпляром каждой задачи в **NP** (который может быть выражен недетерминированной машиной Тьюринга и ее входными данными) и некоторым экземпляром задачи выполнимости (преобразование этой машины и ее входных данных в логические уравнения). Наконец, решение задачи выполнимости, по сути соответствует моделированию работы машины с заданной программой для заданных входных данных, поэтому оно представляет решение заданного экземпляра заданной задачи.

Это доказательство было разработано независимо Стивеном Куком и Леонидом Левиным в начале 1970-х годов. Его принято связывать с обоими именами: статья Кука оказала огромное влияние на распространение концепции **NP**-полноты, но Левин пришел к этому результату раньше. Более подробное доказательство выходит за рамки книги, но вы можете ознакомиться с ним в расширенном курсе информатики. К счастью, достаточно всего одного такого доказательства.

Как и в случае с неразрешимостью проблемы остановки, **NP**-полнота задачи выполнимости логических уравнений чрезвычайно интересна сама по себе, но в действительности ее важность намного шире, потому что она может использоваться для доказательства неразрешимости других важных задач.

Утверждение D. Задача является **NP**-полной, если:

- она принадлежит множеству **NP** (то есть является задачей поиска);
- к ней сводится за полиномиальное время некоторая **NP**-полная задача *A*.

Доказательство непосредственно следует из транзитивности сведения. Любая задача из **NP** за полиномиальное время сводится к задаче *A*, которая за полиномиальное время сводится к заданной задаче.

Для начала примем за A задачу выполнимости логических уравнений. Если мы покажем, что она за полиномиальное время сводится к задаче поиска B, то нам удастся показать, что любая задача из **NP** сводится к B. Иначе говоря, задача B является **NP**-полной.

Сведение по Карпу

В 1972 году Ричард Карп представил сведение задачи выполнимости логических уравнений к 21 известной задаче, которые были известны трудностью решения. В качестве первого примера мы снова рассмотрим задачу 0/1 ЦЛП. Для заданного экземпляра задачи о выполнимости логических уравнений определим множество неравенств, где каждой логической переменной исходных уравнений соответствует одна переменная 0/1. Чтобы преобразовать уравнение в неравенство, замените «= true» на «$\geq$ 1», а каждое отрицание переменной x на $(1 - x)$. Если переменная 0/1 имеет значение 1, то любое неравенство, в котором она встречается без отри-

цания, выполняется; если переменная 0/1 имеет значение 0, то любое неравенство, в котором она встречается с отрицанием, выполняется. Таким образом, решение любого экземпляра задачи 0/1 ЦЛП непосредственно преобразуется в решение соответствующей задачи выполнимости логических уравнений.

Утверждение Е. Задача 0/1 ЦЛП является **NP**-полной.

Доказательство. Ранее уже было показано, что задача 0/1 ЦЛП относится к классу **NP**. Достаточно доказать, что задача выполнимости логических уравнений сводится к задаче 0/1 ЦЛП с полиномиальным временем. Для заданного экземпляра задачи выполнимости логических уровнений определяется множеством неравенств, в котором одна переменная 0/1 соответствует каждой логической переменной. Каждое логическое уравнение преобразуется в неравенство заменой «=true» на «>=1» и каждой инвертированной логической переменной x на (1 – x). Если переменная 0/1 равна 1, то любое неравенство, в котором она присутствует без инвертирования, выполняется. Если переменная 0/1 содержит значение 0, то любое неравенство, в котором она присутствует в инвертированном виде, выполняется. Таким образом, решение любого экземпляра задачи 0/1 ЦЛП непосредственно преобразуется в решение соответствующего экземпляра задачи о выполнимости логических уравнений.

Задача выполнимости логических уравнений

```
   найти значения
true/false x, y, z,    (x' + z)(x + y' + z)(x + y)(x' + y' )
   удовлетворяющие
```

Формулировка задачи выполнимости логических уравнений

```
                          x' + z  = true
   найти значения
true/false x, y, z,     x + y' + z  = true
   удовлетворяющие        x + y  = true
                         x' + y'   = true
```

Формулировка задачи 0/1 ЦЛП

```
                                  (1 - x) + z   1
найти значения 0/1 x, y, z,     x + (1 - y) + z   1
   удовлетворяющие                  x + y   1
                               (1 - x) + (1 - y)    1
```

Решения задачи 0/1 ЦЛП

```
x = 0     x = 1
y = 1     y = 0
z = 1     z = 1
```

Решения задачи выполнимости логических уравнений

```
x = false   x = true
y = true    y = false
z = true    z = true
```

Сведение задачи выполнимости логических уравнений к задаче 0/1 ЦЛП

Пример такого построения показан на с. 816. Обратите внимание: сведение задачи о вершинном покрытии к задаче 0/1 ЦЛП (утверждение C на с. 812) не приводит к аналогичному заключению, потому что **NP**-полнота задачи о вершинном покрытии еще не доказана (пока). Но, к примеру, **NP**-полнота задачи 0/1 ЦЛП может использоваться для доказательства того, что сама задача ЦЛП является **NP**-полной.

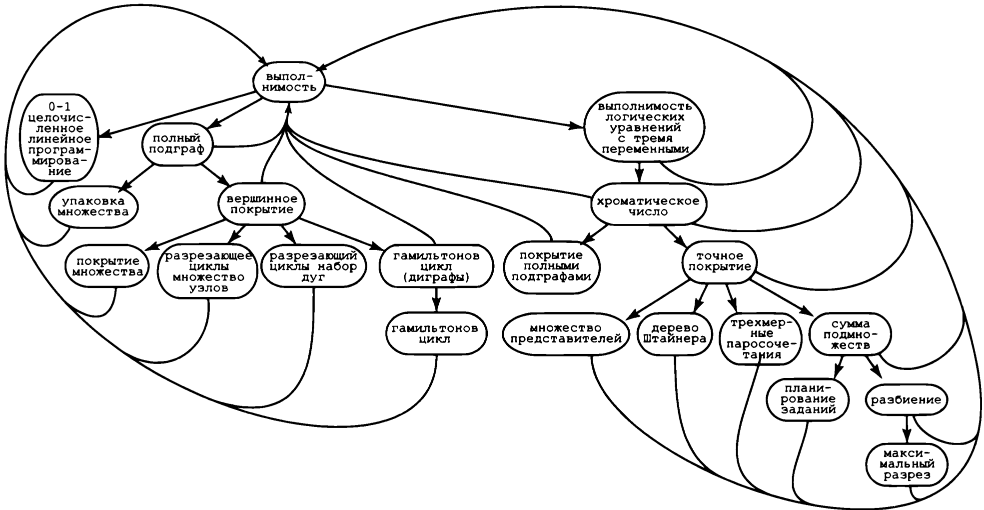
Утверждение F. Задача ЦЛП является **NP**-полной.

Схема доказательства. Ранее уже упоминалось, что задача ЦЛП входит в **NP**. Теперь мы докажем, что задача 0/1 ЦЛП за полиномиальное время сводится к задаче ЦЛП. Чтобы преобразовать произвольный экземпляр задачи 0/1 ЦЛП, преобразуйте его в экземпляр ЦЛП, добавив неравенства вида $x \leq 1$ и $x \geq 0$ для каждой переменной x, ограничивая все переменные значениями 0 и 1. Тогда любое решение экземпляра задачи ЦЛП оказывается непосредственно решением исходной задачи 0/1 ЦЛП.

Итак, алгоритм с полиномиальным временем для задачи ЦЛП предоставит алгоритм с полиномиальным временем для задачи 0/1 ЦЛП, которая предоставит алгоритм с полиномиальным временем для задачи выполнимости логических уравнений, что дает алгоритм с полиномиальным временем для всех задач из **NP** (**P** = **NP**). До выхода статьи Карпа и теоремы Кука — Левина ученые знали, что задача ЦЛП (например) сложна, но не подозревали, что она настолько сложна, что алгоритм с полиномиальным временем предоставит алгоритм с полиномиальным временем для всех задач поиска.

Теперь вы знаете три **NP**-полные задачи и можете доказать **NP**-полноту другой задачи сведением к любой из них. И вы можете видеть, что множество известных **NP**-полных задач способно быстро расширяться.

Задачи в статье Карпа 1972 года «Возможность сведения в комбинаторных задачах» описаны на с. 818, как и доказанные им сведения (стрелка от задачи A к задаче B обозначает сведение за полиномиальное время от A к B, доказанное в статье). Обратите внимание на присутствие в списке задачи о вершинном покрытии, **NP**-полнота которой была доказана сведением за полиномиальное время от задачи выполнимости к поиску полного подграфа, а от полного подграфа — к вершинному покрытию. Статья оказала огромное влияние в научной среде, так как из нее четко следовало, что если хотя бы для одной из этих задач существует алгоритм с полиномиальным временем, то **P** = **NP**. У ученых были разные мнения относительно сложности этих задач, но тот факт, что все они были настолько сложны, что эффективный алгоритм для любой из них означает существование эффективных алгоритмов для всех, был откровением. Ученые немедленно поняли, что искать для любой из этих задач алгоритм с полиномиальным временем (чем годами занимались многие исследователи) не имеет смысла.



Краткие описания NP-полных задач из статьи Карпа, сформулированные в виде задач поиска

Лавина

За десятилетия, прошедшие с момента публикации статьи Карпа, исследователи продемонстрировали взаимосвязь между буквально *десятками тысяч задач* из разнообразных областей, доказав их связи посредством сведения. Некоторые примеры представлены в таблице на с. 820. Интересно, что, в отличие от неразрешимости, количество известных **NP**-полных задач быстро растет. Ежегодно на эту тему публикуются тысячи научных статей. При таком количестве задач из стольких дисциплин вопрос **P = NP?** становится, безусловно, одной из важнейших нерешенных научных проблем нашего времени.

Хотя теория **NP**-полноты и не доказывает, что любая из этих задач является вычислительно сложной, она делает нечто очень важное: разработка алгоритма с полиномиальным временем для любой из этих задач становится эквивалентной доказательству того, что **P = NP**. Другими словами, все они являются *разными воплощениями одной и той же задачи*! И когда вы ищете эффективный (с полиномиальным временем) алгоритм для реконструкции генома, проектирования топологии компьютерной микросхемы или минимизации риска для портфеля ценных бумаг, вы ищете не что иное, как доказательство того, что **P = NP**.

задача	краткое описание
выполнимость логических уравнений	Решить систему логических уравнений
выполнимость логических уравнений с 3 переменными	Решить систему логических уравнений не более чем с 3 переменными на уравнение
0/1 ЦЛП	Решить систему линейных неравенств, ограниченных значениями 0/1
полный подграф	Найти полный подграф, содержащий не менее m вершин
упаковка множества	Найти m или менее попарно изолированных подмножеств из заданного списка
вершинное покрытие	Найти m или менее вершин, соприкасающихся со всеми ребрами графа
покрытие множества	Найти в заданном списке m или менее подмножеств S, объединение которых дает S
разрезающее циклы множество вершин	Найти в графе не более m вершин, при удалении которых граф становится ациклическим
разрезающее циклы множество ребер	Найти в графе не более m ребер, при удалении которых граф становится ациклическим
дерево Штайнера	Соединить множество точек прямыми линиями, суммарная длина которых не превышает m, с возможностью добавления «точек Штайнера», не входящих в исходное множество
направленный гамильтонов цикл	Найти в направленном графе направленный цикл, который посещает каждую вершину ровно один раз
гамильтонов цикл	Найти в графе цикл, который проходит через каждую вершину ровно один раз
раскраска графа	Использовать m и менее цветов для раскраски вершин графа так, чтобы ни одно ребро не соединяло две вершины одного цвета
покрытие подграфами	Разбить граф на m и менее полных подграфов или сообщить о невозможности такого разбиения
точное покрытие	Найти в заданном списке не более m подмножеств S так, чтобы каждый элемент S входил ровно в одно подмножество, или сообщить о том, что это невозможно
множество представителей	Найти подмножество S, содержащее не более m элементов, которое содержит не менее одного элемента из каждого подмножества заданного списка
трехмерные паросочетания	Для заданного множества троек, содержащих один элемент из каждого из трех изолированных множеств, найти такое подмножество, ни один элемент которого не входит в две тройки
укладка ранца	[Эквивалентно нашей формулировке задачи о сумме подмножеств]
планирование заданий	Запланировать выполнение множества заданий определенной длительности на двух процессорах, чтобы все они завершились к заданному времени m
разбиение	Разбить множество целых чисел на два подмножества с равной суммой
максимальный разрез	Найти множество, содержащее не более m ребер, которое разделяет граф на два изолированных подграфа

Краткие описания NP-полных задач из статьи Карпа, сформулированные в виде задач поиска

Область	Типичная NP-полная задача
Аэрокосмическая техника	Оптимальное сеточное разбиение для конечных элементов
Биология	Филогенетическая реконструкция
Химическая технология	Построение сети теплообменников
Химия	Фолдинг белка
Жилищное строительство	Балансировка городских транспортных потоков
Проектирование микросхем	Проектирование топологии СБИС
Экономика	Арбитраж на финансовых рынках
Экология	Оптимальное размещение датчиков загрязнения
Финансовая инженерия	Минимизация портфеля рисков для заданного дохода
Теория игр	Равновесие Нэша, максимизирующее общественное благосостояние
Геномика	Реконструкция генома
Инженерная механика	Структура турбулентности в вязких потоках
Медицина	Реконструкция формы по двумерным ангиокардиограммам
Исследование операций	Задача коммивояжера, целочисленное программирование...
Физика	Статистическая сумма трехмерной модели Изинга
Политика	Индекс влияния Шепли — Шубика
Массовая культура	Разновидности игры судоку, шашки, «сапер», тетрис
Статистика	Оптимальное планирование эксперимента

Примеры NP-полных задач

Проблема NP-полноты

NP-полнота, как и вычислимость, является неизбежным фактом жизни в современном мире. Как упоминалось в нашем обсуждении вычислимости, при поверхностном знакомстве с компьютерами может появиться впечатление, что достаточно мощный компьютер способен на все. Как было показано в многочисленных примерах из раздела 5.4 (и в примерах этого раздела), это предположение, безусловно, неверно. Люди, не знающие о **NP**-полноте, обречены на бесплодные попытки разработать алгоритм для задачи, которая заведомо является **NP**-полной (или **NP**-полнота ее легко доказывается), не подозревая, что это равносильно попыткам побороть известную нерешенную задачу. Всем остальным — в том числе и вам — придется смириться с фактом **NP**-полноты; для этого нужно сначала разобраться с классификацией задачи, а затем принять подходящую стратегию для ее решения на практике.

Практические следствия теории вычислительной сложности

Классификация задач

На практике для подавляющего большинства задач, с которыми вы столкнетесь, есть только два варианта.

- Доказать, что задача принадлежит **P**.
- Доказать, что задача является **NP**-полной.

Есть и другие варианты, но кроме вероятности того, что задача не является задачей поиска или относится к неразрешимым, вы вряд ли встретите их на практике (см. раздел «Вопросы и ответы» на с. 827).

Чтобы доказать, что задача принадлежит **P**, необходимо разработать алгоритм с полиномиальным временем для ее решения — возможно, со сведением за полиномиальное время к задаче, заведомо принадлежащей **P**. Этот процессе принципиально не отличается от написания программы на Java, использующей существующую библиотеку: если известная задача решается с полиномиальным временем, то и клиентское приложение может обладать этим свойством. Зная, что задача принадлежит **P**, можно заняться разработкой улучшенных алгоритмов, как это делалось для многих задач в этой книге.

Чтобы доказать **NP**-полноту задачи, необходимо показать, что она принадлежит **NP**, а другая заведомо **NP**-полная задача сводится к ней за полиномиальное время. Таким образом, алгоритм с полиномиальным временем для новой задачи может использоваться для решения **NP**-полной задачи, которая, в свою очередь, может использоваться для решения всех задач из **NP**. Для многих тысяч задач, о которых мы упоминали выше (кроме задачи выполнимости логических уравнений), **NP**-полнота была продемонстрирована сведением от задач, которые заведомо были **NP**-полными, как в случаях с задачей 0/1 ЦЛП в утверждении E и ЦЛП в утверждении F.

С практической точки зрения этот процесс классификации задач на простые (принадлежащие **P**) или сложные (**NP**-полные) может быть...

- *Тривиальным.* Например, сортировка методом вставки доказывает, что сортировка принадлежит **P**.
- *Нетривиальным, но несложным.* Например, доказательство того, что задача 0/1 ЦЛП является **NP**-полной (утверждение Е), требует некоторого опыта и практики, но оно достаточно понятно.
- *В высшей степени сложным.* Например, задача линейного программирования долгое время считалась открытой, пока в 1980-х годах не была доказана ее принадлежность **P**.
- *Открытым.* Например, задачи *изоморфизма графов* (для двух заданных графов найти способ переименования вершин одного, чтобы он стал идентичен другому) и *разложения на множители* (найти нетривиальные множители для заданного целого числа) до сих пор не классифицированы.

Это весьма активная и «урожайная» область современных исследований, в которой ежегодно публикуются тысячи научных статей. Как видно из списка на с. 820, затрагиваются все направления научного поиска. Вспомните, что наше определение **NP** включает задачи, для которых ученые, инженеры и прикладные программисты стараются найти практически осуществимые решения, — конечно, такие задачи необходимо классифицировать!

Стратегии решения NP-полных задач

Практика требует положительных сдвигов для целого вороха существующих задач, поэтому поиск путей их решения вызывает повышенный интерес. Теория утверждает, что существует много важных задач, для которых мы не можем рассчитывать на создание алгоритма, одновременно достигающего трех важных свойств

- Гарантия оптимального решения задачи (для задач оптимизации).
- Гарантия решения задачи за полиномиальное время.
- Гарантия решения произвольных экземпляров задачи.

Соответственно при обнаружении **NP**-полной задачи необходимо ослабить хотя бы одно из этих трех требований. Невозможно отдать должное всей необъятной области исследований в нескольких абзацах, но мы кратко опишем два подхода, которые оказались успешными.

Приближенные решения

Ослабление требования оптимальности равносильно изменению задачи и разработке приближенного алгоритма, который не обязательно находит наилучшее возможное решение, но решение, которое гарантированно близко к наилучшему. Например, можно легко найти решение оптимизационной версии задачи о вершинном покрытии, которое лежит в пределах множителя 2 от оптимального (см. упражнение 5.5.41). В области проектирования приближенных алгоритмов ведутся активные исследования. К сожалению, этот подход часто непрактичен и часто недо-

статочен для компенсации **NP**-полноты при поиске улучшенных аппроксимаций. Например, *неаппросимируемость* бывает результатом того, что если вы сможете найти приближенный алгоритм для конкретной задачи, которая гарантированно обеспечивает результат в пределах множителя 2 от оптимального, то это означает, что **P** = **NP**. Конечно, разработка приближенных алгоритмов для таких **NP**-полных задач невозможна (конечно, если не выяснится, что **P** = **NP**).

Отход от гарантированного быстродействия для наихудших случаев

Ослабление двух других требований равносильно разработке алгоритма, который эффективно решает типичные экземпляры задачи, возникающие на практике, несмотря на то что существуют варианты входных данных для худшего случая, для которых поиск решения оказывается нереальным. Гарантия полиномиального времени заменяется надеждой на то, что задача будет решаться за полиномиальное время для наборов входных данных, встречающихся на практике, а гарантия решения произвольных экземпляров задачи заменяется пониманием того, что для некоторых экземпляров задачи она будет выполняться за экспоненциальное время. В современных промышленных приложениях системы решения задач выполнимости, ЦЛП и других **NP**-полных задач регулярно используются для решения крупных возникающих задач. Ученые интенсивно изучают конкретные задачи, для которых системы решения работают медленно (или стараются держаться от них подальше!), пока не будет найден какой-либо успешный метод для их решения. Сейчас мы рассмотрим такой метод для задачи выполнимости.

Выполнимость логических уравнений

В завершение этого раздела мы разработаем алгоритм для решения экземпляров печально известной задачи выполнимости. Наша реализация прольет свет на очень важный вопрос, который десятилетиями обсуждался в среде компьютерных теоретиков. *Можно ли отнести задачу выполнимости к легко решаемым задачам или нет*?

задача:

$x' + z = true$

$x + y' + z = true$

$x + y = true$

$x' + y' = true$

формат входных данных:

```
-.+
+-+
++.
--.
```

Представление задачи выполнимости

Прежде всего нам понадобится система обозначений для представления экземпляров задачи и потенциальных решений. Пусть m — количество уравнений, а n — количество переменных. Для представления решения мы используем массив `inSubset[]` длины m с элементами типа `boolean`, который идентифицирует подмножество переменных, которым присваивается значение `true`. Для представления экземпляра задачи используется компактная кодировка, показанная справа. Каждое уравнение представляется строкой из n символов, в которой i-й символ соответствует i-й переменной и содержит `'+'`, если переменная встречается в уравнении (без отрицания), `'-'`, если переменная встречается в уравнении (с отрицанием), или `'.'`, если переменная в уравнении не встречается. Таким образом, каждый экземпляр задачи может быть представлен массивом `clauses[]` из m строк.

С таким представлением можно легко проверить, удовлетворяет ли предполагаемое решение inSubset[] всем уравнениям:

```
public static boolean check(String[] clauses, boolean[] inSubset)
{
   boolean product = true;
   for (int i = 0; i < clauses.length; i++)
   {
      boolean sum = false;
      for (int j = 0; j < inSubset.length; j++)
      {
         if (clauses[i].charAt(j) == '+') sum = sum ||  inSubset[j];
         if (clauses[i].charAt(j) == '-') sum = sum || !inSubset[j];
      }
      product = product && sum;
   }
   return product;
}
```

Функция check(), выполняемая за время, линейно зависящее от размера задачи, доказывает, что задача выполнимости принадлежит **NP**. Она также служит критически важным компонентом системы решения задачи выполнимости, которая будет описана ниже.

Чтобы найти решение экземпляра задачи выполнимости, мы можем перебрать все 2^n возможных потенциальных решений и использовать функцию check() для выявления удовлетворяющего всем уравнениям. Программа SAT (листинг 5.5.1) для этого «считает» все возможные варианты присваивания значений в массиве inSubset[]: false соответствует 0, true — 1, а весь массив — двоичному числу (см. иллюстрацию на с. 796). Чтобы подчеркнуть, что это простое вычисление (даже для машины Тьюринга), метод next() в SAT использует алгоритм, использовавшийся в машине Тьюринга на с. 740, для «увеличения» inSubset[] до следующего набора значений: сканирование от правого края с заменой значений false на true до обнаружения true, которое будет заменено на false. Чтобы завершить выполнение, если не будет обнаружено ни одно значение true, метод next() возвращает в этом случае false; в противном случае возвращается true. Метод-конструктор управляет процессом перебора, вызывая next() для вычисления следующего предполагаемого решения, проверяя это решение вызовом check() и продолжая до тех пор, пока вызов next() не вернет false. Метод main() читает данные из файла, заданного в аргументе командной строки, и создает объект SAT для решения экземпляра задачи выполнимости.

```
public static void main(String[] args)
{
   String filename = args[0];
   In in = new In(filename);
   String[] clauses = in.readAllStrings();
   SAT solver = new SAT(clauses);
   StdOut.println(solver);
}
```

Листинг 5.5.1. Система решения задачи выполнимости

```
public class SAT
{
   private boolean[] inSubset;
   private final String[] clauses;
   private final int n;

   public SAT(String[] clauses)
   {
      this.clauses = clauses;
      n = clauses[0].length();
      inSubset = new boolean[n];
      while (next())
         if (check(clauses, inSubset)) return;
   }

   private boolean next()
   {
      int i = n-1;
      while (inSubset[i])
      {
         inSubset[i--] = false;
         if (i == -1) return false;
      }
      inSubset[i] = true;
      return true;
   }

   public static boolean check(...)
   {  /* См. с. 824 */  }

   public String toString()
   {  /* См. упражнение 5.5.9 */  }

   public static void main(String[] args)
   {  /* См. с. 824 */  }
}
```

```
% more tinySAT.txt
-.+
+-+
++.
--.
```

```
% java SAT tinySAT.txt
011
```

Программа решает произвольный экземпляр задачи выполнимости, перебирая все возможные варианты присваивания значений элементам массива inSubset[]. Этот алгоритм имеет экспоненциальное время выполнения.

Это классическое, фундаментальное упражнение из области программирования, и для него существует много других подходов, связанных с темами, рассмотренными в других местах книги. Эти реализации достаточно интересны, но они мало что меняют в контексте теории вычислений, поэтому мы опишем лишь некоторые из них в упражнениях (см. упражнения 5.5.15, 5.5.16 и 5.5.17).

При всей простоте программа SAT может легко решать большие экземпляры задачи, как видно из примера справа на с. 826. Обратите внимание: SAT прекращает работу сразу же после обнаружения решения. В данном примере из 2^{30} = 1 073 741 824 ва-

риантов ей потребовалось всего 292, чтобы найти решение. Для тестирования использовались «случайные» экземпляры, содержащие в среднем 6 переменных на уравнение, сгенерированные программой из упражнения 5.5.13.

```
% more SAT30-by-30.txt
+...-..-.......--+.........+.-
............+.......-+.-..+...
.....+-...........-..+....-...
+........-.....+..............
..-............-.........-....
+......-+.-.-+...-..........-.
.--..-.......-......+........-
.-......+-..+-.++....+...+.+..
..+.........-....+......--..-.
.-+.-.+..++.....-.+...-+....-.
..-...--........+.....-...-..-
...-.........+........+-.-....
+........-..-+...-.....-....--
....-...........+.............
..-......-+........+.........-
+.++..-..+.....-.+..-.+.+.....
.+..-......+..-.......-.-.....
..................++.-.....+..
...+..+..+..............++....
.................-....-++-....
.........................+..-.
.+.-..+..+...+.+........+.....
..+....-..-.-+....++..+....+..
-.............-..-.--.....+-..
+.-....................+......
..+.......+.....+....+........
..++................+.-....---
..+--+.-.....................+
.+...............+......+-.-..
+..-....++........-.....+.....
% java SAT SAT30-by-30.txt
000000000000000000000100100100
```

Как упоминалось выше, системы решения задач выполнимости широко применяются в промышленности, и в них используются всевозможные приемы для ускорения выявления решений (сокращение перебора, «отсечение ветвей» и т. п.). Например, если экземпляр содержит два уравнения $x + y = true$ и $x' + y = true$, можно сделать вывод, что в любом решении переменная y должна быть истинной; более того, замена y значением *true* во всех остальных уравнениях может привести к дальнейшим упрощениям. Системы решения, используемые в наше время, появились в результате десятилетий подобных разработок. В своей книге *The P = NP Question and Gödel's Lost Letter* Р. Дж. Липтон (R. J. Lipton) удачно описывает ситуацию: «Системы решения задач выполнимости работают — кроме тех случаев, когда они не работают». Можно добавить к этому: «...и тогда их исправляют».

Тем не менее для некоторых вариантов входных данных все системы решения задач выполнимости требуют экспоненциального времени. Например, листинг 5.5.1 анализирует все 2^n возможных назначений для экземпляров задачи, не имеющих решений. К чему окажутся ближе экземпляры, встречающиеся реально, — к худшему случаю или к какой-то иной случайной модели? Никто не знает. В контексте теории вычислений не остается другого выбора, кроме как отнести системы решения к алгоритмам с экспоненциальным временем.

Эта в высшей степени интересная ситуация подчеркивает тот факт, что обсуждаемая в этом разделе теория базируется на анализе худшего случая. Основные идеи теории вычислительной сложности красиво дополняют наше понимание компьютерных вычислений, но программисты из реального мира добились значительных успехов в том, чтобы обойти быстродействие худшего случая.

NP-полнота встречается во всех прикладных областях, рассмотренных в этой книге. **NP**-полные задачи встречаются в элементарном программировании, при сортировке и поиске, при обработке графов, при обработке строк, в научных вычислениях, в программировании сложных систем и вообще в любой области, где вычисления играют сколько-нибудь заметную роль. Мало какие научные теории обладают столь обширным и глубоким влиянием.

Что касается проблемы вычислимости, упоминавшейся в предыдущем разделе, ее основа достаточно надежна: мало кто сомневается в истинности гипотезы Чёрча — Тьюринга, и у нас имеются доказательства того, что некоторые задачи не имеют решения. Если же говорить о проблеме вычислительной сложности, обсуждаемой в этом разделе, два открытия не позволяют выступать с такими однозначными утверждениями. Во-первых, концепция квантовых вычислений заставила некоторых специалистов усомниться в расширенном тезисе Чёрча — Тьюринга (при этом все еще нет никаких доводов в пользу того, что **NP**-полные задачи могут быть решены на квантовом компьютере за полиномиальное время, даже если он станет физически реализуемым вычислительным устройством[1]). Во-вторых, никто еще не доказал, что хотя бы одна задача поиска является вычислительно сложной — это стало бы доказательством того, что **P** и **NP** не равны.

Даже в этом случае существование **NP**-полных задач значительно расширяет наши представления о внутренних ограничениях компьютеров. Такие задачи могут иметь огромную практическую значимость, но нам приходится работать в условиях известных нам ограничений компьютеров: приходится признать, что эффективное решение таких задач при текущем состоянии наших знаний не гарантировано.

Вопросы и ответы

В. Всегда ли практичны алгоритмы с полиномиальным временем?

О. Нет, алгоритмы с n^{100} или $10^{100}n^2$ шагов на практике так же бесполезны, как и алгоритмы с экспоненциальным временем. Но реально встречающиеся константы обычно достаточно малы, чтобы было разумно использовать P как некоторый заменитель понятия множества «практичных» задач.

В. А алгоритмы с экспоненциальным временем всегда бесполезны?

О. Нет. Например, алгоритм с временем выполнения $1{,}00001^n$ с большой вероятностью будет весьма полезен — скажем, для n меньше 1 миллиона. Аналогичным образом программа SAT (листинг 5.5.1) весьма полезна для многих вариантов входных данных, включая случайно сгенерированные варианты, которые мы рассматривали.

В. Чем объясняется асимметрия в определениях алгоритма с полиномиальным временем (определяемого в контексте верхней границы для времени выполнения в худшем случае) и алгоритма с экспоненциальным временем (определяемого в контексте нижней границы времени выполнения в худшем случае)?

О. Обычно мы стараемся отделить «простые» задачи от «сложных». Для простых задач нас интересует верхняя граница с полиномиальным временем, тогда как для сложных — нижняя граница с экспоненциальным временем. Соответственно когда

---

[1] Ожидается, что он сможет выполнять параллельные недетерминированные вычисления. Устройства, реализующие квантовые принципы вычислений, уже созданы, но до полноценного квантового компьютера все еще далеко. — *Примеч. науч. ред.*

речь заходит об алгоритме с экспоненциальным временем, мы имеем в виду, что он выполняется *не менее чем* за экспоненциальное время (для некоторого бесконечного набора вариантов входных данных), а не *не более чем* за экспоненциальное время.

В. Почему определение алгоритма с экспоненциальным временем требует, чтобы алгоритм выполнялся за экспоненциальное время для *бесконечного количества* вариантов входных данных, а не для *всех* вариантов?

О. Это было бы разумным альтернативным определением (причем некоторые ученые предпочитают именно эту, более жесткую версию). Мы предпочитаем менее жесткое определение, потому что любой алгоритм, выполняемый за экспоненциальное время для бесконечного класса вариантов входных данных, можно считать неэффективным.

В. Существуют ли задачи, для которых лучший возможный алгоритм требует экспоненциального времени независимо от гипотезы **P** = **NP**?

О. Да. Впрочем, число реальных задач, требующих экспоненциального времени, невелико. Важным исключением является следующая версия задачи остановки: для заданной машины Тьюринга (или программы Java), не получающей входных данных, и целого числа k остановится ли машина Тьюринга (или программа Java) менее чем за k шагов? Задачу можно решить не более чем за k шагов — достаточно запустить машину Тьюринга (или программу Java), пока она не остановится или не пройдет k шагов. И наоборот, исследователи доказали, что добиться принципиально лучших результатов по сравнению с этим решением «грубой силы» невозможно. Моделирование методом «грубой силы» экспоненциально зависит от размера входных данных, потому что параметр k представим в двоичной форме с использованием $\lg k$ битов (то есть число шагов моделирования k экспоненциально зависит от разрядности параметра).

В. Как классифицировать задачи, которые решаются алгоритмами за время, пропорциональное другим функциям — не полиномиальным и не экспоненциальным, например $n^{\log n}$?

О. Хороший вопрос. Известный пример задачи такого рода — задача изоморфизма графов: для двух заданных графов определить, эквивалентны ли они за пределами имен их вершин. Эта задача принадлежит **NP**, но пока неизвестно, является ли она **NP**-полной. Алгоритм с временем выполнения такого рода был обнаружен в 2015 году. То есть если когда-нибудь для задачи изоморфизма графов будет найден алгоритм с полиномиальным временем выполнения, это не будет означать, что **P** = **NP**.

В. Существуют ли в **NP** задачи, которые не входят в **P** и не являются **NP**-полными?

О. Да. В предположении, что **P** ≠ **NP**, Р. Ладнер (R. Ladner) в 1975 году доказал, что в **NP** существуют задачи, которые не принадлежат **P** и не обладают **NP**-полнотой. Некоторые исследователи полагают, что задачи разложения на множители и изоморфизма графов относятся к этому классу сложности.

В. Я слышал о классе сложности «**NP**-сложные задачи». Что это такое?

О. Задача называется **NP**-сложной (или **NP**-трудной), если все задачи из **NP** сводятся к ней. Определение не отличается от **NP**-полноты, но без требования, чтобы сама задача принадлежала **NP**.

В. Почему современные криптосистемы не базируются на **NP**-полной или **NP**-сложной задаче (вместо разложения на множители)?

О. Ученые пытались идти по такому пути. Но **NP**-полнота связана со сложностью худшего случая, тогда как в криптографических приложениях задача должна быть гарантированно сложной для всех вариантов входных данных, встречающихся на практике.

В. Где я могу больше узнать о **NP**-полноте?

О. Классический учебник по теме — Гэри (Garey) и Джонсон (Johnson), *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Недавно он занял первое место по количеству цитирований в литературе по информатике. Многие важные последующие открытия в этой области описаны в рубрике Джонсона, посвященной **NP**-полноте, в издании *Journal of Algorithms*.

Упражнения

5.5.1. Заполните пропуски в следующей таблице (для больших чисел — с точностью до порядка).

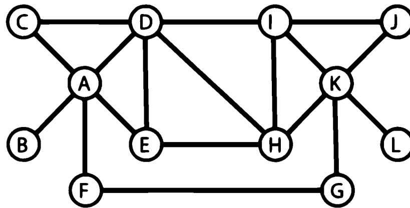
n	$(1,1)^n$	n^4	2^n	n!
10	2,59...		1024	3628800
100	13781	10^8	10^{30}	10^{158}
1000	10^{41}			
10 000				

5.5.2. Напишите Java-программу, реализующую метод исключения Гаусса. На входе программа должна получать n строк (по одной для каждого уравнения), каждая из которых содержит $n + 1$ значений `double` (коэффициенты при переменных и свободный член — значение из правой части уравнения). Сначала предположите, что уникальное решение существует, и отладьте программу на примере, приведенном в тексте. Затем проанализируйте возникающие проблемы и стратегии их решения.

5.5.3. Будет ли правильно характеризовать алгоритм со временем, пропорциональным $1{,}5^n$, для некоторого бесконечного множества вариантов входных данных как «алгоритм с экспоненциальным временем»? Объясните свой ответ. Повторите упражнение для функций $n!$, n^4, $2^{\sqrt{n}}$ и $n^{\log n}$.

5.5.4. Напишите программу, которая читает последовательность из n целых чисел из стандартного потока ввода и находит непустое подмножество чисел, сумма которых равна 0 (или сообщает, что такое подмножество не существует). Переберите все 2^n подмножеств из n целых чисел.

5.5.5. Найдите вершинное покрытие с минимальным количеством элементов для следующего графа:



5.5.6. Приведите приблизительную оценку количества подмножеств с количеством элементов не более m в графе с n вершинами, если (1) m является константой k и (2) значение m равно приблизительно $n/2$.

5.5.7. Напишите метод, который получает граф G и подмножество вершин и определяет, является ли подмножество вершин вершинным покрытием.

5.5.8. Докажите, что любая задача выполнимости логических уравнений может быть преобразована в форму, в которой в левой части не используется ни одна операция «и», а все правые части истинны. *Подсказка*: обратитесь к представлению логических функций в виде суммы произведений (см. раздел 7.1).

5.5.9. Реализуйте метод `toString()` класса `SAT` (листинг 5.5.1).

5.5.10. Разработайте представление для задач выполнимости, использующее двумерный массив целых чисел. Используйте обозначения `+1`, `-1` и `0` по аналогии с тем, как мы использовали `"+"`, `"-"` и `"."` соответственно в представлении массива в формате `String`, которое использовалось в тексте. Докажите, что задача выполнимости входит в NP, разработав метод `check()` с полиномиальным временем выполнения для вашего представления.

5.5.11. Докажите, что задача 0/1 ЦЛП принадлежит **NP**.

5.5.12. Рассмотрите следующую классическую задачу.

Задача коммивояжера. Для заданного множества из n городов и заданного расстояния m найдите маршрут длиной не больше m, проходящий по всем городам, или сообщите, что такого маршрута не существует.

Чтобы избежать технических проблем при сравнении сумм квадратных корней из целых чисел, считайте, что все расстояния задаются целыми числами (допустим, евклидовым расстоянием в метрах или километрах, округленным до ближайшего целого). Докажите, что задача коммивояжера принадлежит **NP**; для этого разработайте метод `check()` с полиномиальным временем, который проверяет, что

заданный маршрут является решением для заданного экземпляра задачи. Предположите, что в аргументах check() передается целое число n (количество городов), целое число m (верхняя граница длины маршрута), два целочисленных массива x[] и y[] (координаты x и y точек) и целочисленный массив tour[], который задает порядок следования городов в маршруте.

5.5.13. Напишите программу GenerateSAT, которая генерирует случайные экземпляры для SAT в формате, описанном в тексте. Программа должна получать следующие аргументы командной строки:

- m — количество уравнений;
- n — количество переменных;
- p — процент неинвертированных переменных;
- q — процент инвертированных переменных.

Решение.

```
public class GenerateSAT
{
    public static void main(String[] args)
    {
        int m = Integer.parseInt(args[1]);
        int n = Integer.parseInt(args[0]);
        double p = Double.parseDouble(args[2]);
        double q = Double.parseDouble(args[3]);
        for (int k = 0; k < m; k++)
        {
           String equation = „";
           for (int i = 0; i < n; i++)
           {
               double x = StdRandom.uniform();
               if (x < p)          equation += „+";
               else if (x < p+q) equation += „-";
               else                equation += „.";
           }
           StdOut.println(equation);
        }
    }
}
```

Файл SAT30-by-30.txt, упоминаемый в тексте, был создан этой программой, запущенной командой java GenerateSAT 30 30 0.1 0.1.

5.5.14. Поэкспериментируйте с различными значениями аргументов программы GenerateSAT из упражнения 5.5.13 и найдите экземпляр задачи выполнимости с 30 переменными, который заставляет SAT протестировать максимально возможное количество экземпляров.

5.5.15. В разделе 6.1 описаны операции Java с двоичными представлениями значений int и представлен код следующего вида:

1 << n для вычисления 2^n;

`(v >> i) & 1` для получения i-го бита с правого края двоичного представления `v`.

Разработайте реализацию `SAT`, в которой используются эти фрагменты кода.

5.5.16. Разработайте реализацию `SAT`, основанную на рекурсивной функции, такой как `TowersOfHanoi` (листинг 2.3.2): присвойте крайнему правому элементу `inSubset[]` значение `false` и сгенерируйте все остальные значения рекурсивным вызовом, затем присвойте крайнему правому элементу значение `true` и выполните рекурсивный вызов для генерирования всех остальных значений (снова). Базовым случаем должен быть вызов `check()`.

5.5.17. Разработайте реализацию `SAT`, основанную на рекурсивной функции, такой как `Beckett` (листинг 2.3.3). В некоторых ситуациях этот метод лучше решения из упражнения 5.5.16, потому что размер подмножества каждый раз изменяется не более чем на 1.

5.5.18. Взяв за основу код `SAT` и метод `check()` на с. 824, разработайте программу `SubsetSum` для поиска решения в задаче о сумме подмножества для любого заданного множества чисел.

5.5.19. Взяв за основу код `SAT` и метод `check()` из упражнения 5.5.7, разработайте программу `VertexCover` для поиска решения задачи о вершинном покрытии для любого заданного графа (и верхней границы `m` количества вершин в покрытии).

5.5.20. Инкапсулируйте код генерирования подмножества из `SAT` в отдельный класс. Затем разработайте реализации `SAT` (листинг 5.1.1), `SubsetSum` (упражнение 5.5.18) и `VertexCover` (упражнение 5.5.19), которые являются клиентами этого класса.

5.5.21. Опишите семейство экземпляров задачи выполнимости, с которыми `SAT` рассматривает все 2^n случаев для n переменных.

Решение: используйте набор из n уравнений, в котором i-е уравнение содержит только i-ю переменную (без отрицания). Эти уравнения выполняются все одновременно только в том случае, когда все переменные истинны, — а это последний случай, рассматриваемый `SAT`.

5.5.22. Допустим, две задачи заведомо являются **NP**-полными. Следует ли из этого, что одна из них сводима к другой за полиномиальное время?

5.5.23. Допустим, задача X является **NP**-полной, X сводится к Y, а Y сводится к X. Обязательно ли задача Y является **NP**-полной?

Решение: нет, так как Y может не принадлежать **NP**.

5.5.24. Может ли существовать алгоритм, решающий **NP**-полную задачу за время, ограниченное $n^{\log n}$, если $\mathbf{P} \neq \mathbf{NP}$? Объясните свой ответ.

5.5.25. Предположим, кто-то открыл алгоритм, который гарантированно решает задачу выполнимости логических уравнений за время, пропорциональное $1{,}1^n$. Следует ли из этого, что другие **NP**-полные задачи могут решаться за время, пропорциональное $1{,}1^n$?

5.5.26. Какую бы важную роль сыграла программа, способная решить задачу о вершинном покрытии за время, пропорциональное $1{,}1^n$?

5.5.27. Какие из следующих выводов можно сделать из факта **NP**-полноты задачи коммивояжера, если предположить $\mathbf{P} \neq \mathbf{NP}$?

a. Не существует алгоритма, решающего произвольные экземпляры задачи коммивояжера.

b. Не существует алгоритма, эффективно решающего произвольные экземпляры задачи коммивояжера.

c. Существует алгоритм, который эффективно решает произвольные экземпляры задачи коммивояжера, но его еще никому не удалось найти.

d. Задача коммивояжера не принадлежит **P**.

e. Все алгоритмы, которые гарантированно решают задачу коммивояжера, выполняются за полиномиальное время для некоторых классов входных данных.

f. Все алгоритмы, которые гарантированно решают задачу коммивояжера, выполняются за экспоненциальное время для всех классов входных данных.

Решение: только b и d.

5.5.28. Какие из следующих выводов можно сделать из того факта, что задача разложения на множители принадлежит **NP**, но при этом о ее принадлежности **P** или **NP**-полноте неизвестно (если предположить, что $\mathbf{P} \neq \mathbf{NP}$)?

a. Существует алгоритм, решающий произвольные экземпляры задачи разложения на множители.

b. Существует алгоритм, эффективно решающий произвольные экземпляры задачи разложения на множители, но его еще никому не удалось найти.

c. Если мы найдем эффективный алгоритм для задачи разложения на множители, его можно будет использовать для решения задачи коммивояжера.

5.5.29. Объясните, почему ни одна из следующих задач не является **NP**-полной.

a. Алгоритм для решения задачи коммивояжера методом «грубой силы».

b. Сортировка слиянием.

c. Проблема остановки.

d. 10-я проблема Гильберта.

Решение: понятие **NP**-полноты описывает задачи, а не конкретные алгоритмы для решения этих задач, поэтому описывать пункты a и b как **NP**-полные неверно (как, скажем, называть **NP**-полными яблоки или апельсины). Проблема остановки и 10-я проблема Гильберта неразрешимы, поэтому они не принадлежат **NP** и, следовательно, не являются **NP**-полными.

5.5.30. Докажите *транзитивность* сведения с полиномиальным временем. Иначе говоря, докажите, что если задача A сводится за полиномиальное время к B, а задача B сводится за полиномиальное время к C, то задача A сводится за полиномиальное время к C.

5.5.31. Пусть A и B — две задачи принятия решения. Допустим, вы знаете, что задача A сводится за полиномиальное время к B. Какие из следующих выводов можно сделать?

a. Если задача B является **NP**-полной, то и задача A является **NP**-полной.

b. Если задача A является **NP**-полной, то и задача B является **NP**-полной.

c. Если задача B является **NP**-полной, а задача A принадлежит **NP**, то и задача A является **NP**-полной.

d. Если задача A является **NP**-полной, а задача B принадлежит **NP**, то и задача B является **NP**-полной.

e. Задачи A и B не могут быть **NP**-полными одновременно.

f. Если задача A принадлежит **P**, то и задача B принадлежит **P**.

g. Если задача B принадлежит **P**, то и задача A принадлежит **P**.

Решение: только d и g.

5.5.32. Докажите, что задача нахождения гамильтонова цикла в направленном графе является **NP**-полной (используя сведение от задачи нахождения гамильтонова цикла для ненаправленных графов).

5.5.33. Допустим, имеется алгоритм для решения задачи выполнимости логических уравнений (в варианте принятия решения) — для любых введенных данных он определяет, существует ли вариант присваивания значений переменным, с которым выполняется логическое выражение. Покажите, как использовать такой алгоритм для нахождения такого варианта.

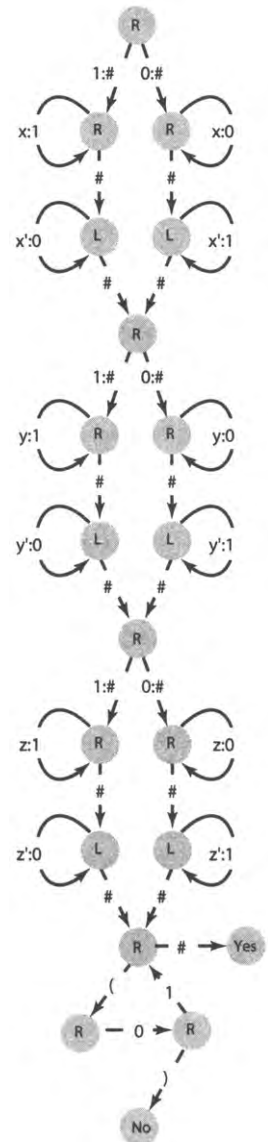
5.5.34. Допустим, имеется алгоритм для решения задачи о вершинном покрытии — для любого графа и любого целого числа m он определяет, существует ли вершинное покрытие с количеством элементов не более m (или сообщает, что оно не существует). Покажите, как использовать такой алгоритм для решения оптимизационной версии задачи — для любого заданного графа найти вершинное покрытие с минимальным количеством элементов.

5.5.35. Объясните, почему мы не знаем, является ли оптимизационная версия задачи о вершинном покрытии задачей поиска.

Решение: не существует способа проверить, что предполагаемое решение является лучшим из возможных. Версия с принятием решения является задачей поиска (если расстояние задается целыми числами), потому что для нахождения лучшего решения можно воспользоваться бинарным поиском.

Упражнения повышенной сложности

5.5.36. *Разложение на множители.* Измените программу `Factors` (листинг 1.3.9) для использования `BigInteger` и воспользуйтесь ею для разложения чисел 1111111111, 11111111111, ... , (10*n* – 1) / 9, ... и т. д. — до тех пор, пока время выполнения не превышает 10 секунд.



5.5.37. *Разложение на множители с принятием решения.* Покажите, как преобразовать версию задачи разложения на множители в версии с принятием решения в задачу поиска. Говоря конкретнее, если имеется статический метод `factor(x,y)`, который возвращает `true`, если `x` имеет нетривиальный множитель, меньший `y`, напишите статический метод `factors()`, который выводит список множителей `x`.

5.5.38. *Машина Тьюринга для проверки задачи выполнимости.* Разработайте машину Тьюринга, которая проверяет, выполняется ли заданный экземпляр задачи выполнимости для заданного множества значений. Предполагается, что используются три переменные, алфавит `#01xyzx'y'z'()#`, а на ленте изначально находятся проверяемые значения `x`, `y` и `z`, за которыми следует сокращенная форма экземпляра задачи выполнимости. Например, для примера, приведенного в тексте, содержимое ленты будет выглядеть так:

```
#011(x'+z)(x+y'+z)(x+y)(x'+y')#
```

Решение: машина Тьюринга изображена справа. Для каждой переменной она читает значение, подставляет его на место всех вхождений переменной в выражении в процессе сканирования вправо, после чего подставляет дополнение этого значения на место всех вхождений инвертированного символа переменной в процессе сканирования влево. После всех замен пять состояний в нижней части машины сканируют выражение в поисках пары круглых скобок, не содержащих значение 1. Если такая пара будет найдена, машина входит в состояние `Yes`, а если нет — в состояние `No`. Решение легко расширяется для *n* переменных, но оно не доказывает, что задача выполнимости принадлежит **NP**. Необходима более сложная конструкция, не зависящая от *n*.

5.5.39. *Моделирование системы проверки задачи выполнимости.* Создайте текстовый файл с табличным представлением универсальной машины Тьюринга для ДКА

из упражнения 5.5.38, загрузите файлы TuringMachine.java и Tape.java с сайта книги, измените их так, как описано в упражнении 5.2.7 и упражнении 5.2.8, и выведите трассировку работы машины для заданных входных данных.

Решение.

```
# 0 1 1 ( x'+ z ) ( x + y'+ z ) ( x + y ) ( x'+ y' ) #     начать с левого края
# # 1 1 ( x'+ z ) ( 0 + y'+ z ) ( 0 + y ) ( x'+ y' ) #     заменить x на 0
# # 1 1 ( 1 + z ) ( 0 + y'+ z ) ( 0 + y ) ( 1 + y' ) #     заменить x' на 1
# # # 1 ( 1 + z ) ( 0 + y'+ z ) ( 0 + 1 ) ( 1 + y' ) #     заменить 1 на y
# # # 1 ( 1 + z ) ( 0 + 0 + z ) ( 0 + 1 ) ( 1 + 0 ) #      заменить 0 на y'
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      заменить 1 на z
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      заменить 0 на z'
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до ( или #
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до 1 перед )
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до ( или #
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до 1 перед )
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до ( или #
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до 1 перед )
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до ( или #
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до 1 перед )
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      сканировать до ( или #
# # # # ( 1 + 1 ) ( 0 + 0 + 1 ) ( 0 + 1 ) ( 1 + 0 ) #      принять
```

5.5.40. *Сумма подмножества* (решение методом динамического программирования). Разработайте программу Java `SubsetSumDP`, которая читает целые числа из стандартного потока ввода и перебирает все возможные варианты выборки для нахождения подмножества, сумма элементов которого равна 0. Используйте следующий метод: создайте двумерный массив `subset[][]` с элементами `boolean` и обеспечьте выполнение инварианта «элемент `subset[i][j]` равен `true`, если существует подмножество первых `j` чисел, сумма которого равна `i`». За какое время выполняется ваша программа? Объясните, почему ваша программа не доказывает, что **P = NP**.

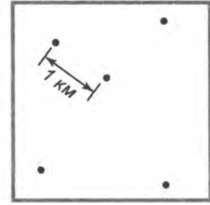
5.5.41. *Приближенный алгоритм для задачи о вершинном покрытии.* Рассмотрите следующий алгоритм для определения вершинного покрытия: удалить ребро, добавить его конечные точки в покрытие, удалить все ребра, инцидентные его конечным точкам; продолжать, пока остаются ребра. Докажите, что в результате будет получено вершинное покрытие, которое содержит не более чем в 2 раза больше вершин, чем лучшее возможное (минимальное) решение.

5.5.42. *Уравнение Пелла.* Создайте программу `Pell` — клиент `BigInteger`, которая читает целое число `c` и находит минимальное решение уравнения Пелла: $x^2 - cy^2 = 1$. Протестируйте с $c = 61$. Наименьшее решение имеет вид (1 766 319 049,

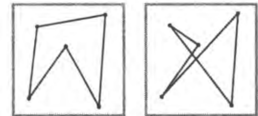
226 153 980), а для c = 313 — (3 218 812 082 913 484, 91 819 380 158 564 160). Доказано, что эта задача не решается за полиномиальное количество шагов (как функции от количества битов во введенном c), потому что количество битов в выходных данных может расти экспоненциально!

задача:

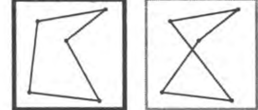
найти маршрут ≤ 5 км через 5 заданных городов

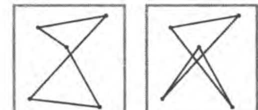


все варианты

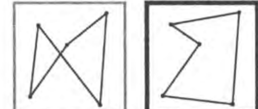


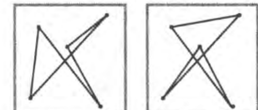
решение

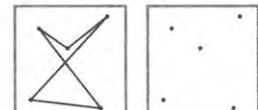




решение







Экземпляр евклидовой задачи коммивояжера

5.5.43. *Евклидова задача коммивояжера.* Разработайте Java-программу `TSP`, которая проверяет все возможные варианты для нахождения маршрута наименьшей длины в задаче коммивояжера (см. упражнение 5.5.12). Используйте рекурсивное решение, которое хранит все города текущего маршрута и проверяет все возможные варианты следующего города. *Примечание*: программа не будет завершаться для больших n, потому что количество возможных вариантов для проверки равно $(n - 1)!/2$, а значение $n!$ намного, намного больше 2^n (см. упражнение 5.5.1). Алгоритм, который бы гарантированно решал эту задачу для больших n, никому не известен.

5.5.44. *Евклидова задача коммивояжера с возвратом.* Измените свою программу из упражнения 5.5.43, чтобы она останавливала поиск на любом пути, длина которого превышает длину текущего известного минимального маршрута. Выведите таблицу с количеством вариантов, проверенных в двух случаях для разных n, используя случайные точки с положительными целочисленными координатами, меньшими 1 миллиона.

Глава 6
Вычислительная машина

В этой главе мы хотим показать, насколько в действительности прост компьютер, которым вы пользуетесь. Мы подробно опишем простую воображаемую машину, обладающую многими чертами реальных процессоров — «сердца» вычислительных устройств, которые нас окружают.

Возможно, вас удивит, но очень многие машины обладают одинаковыми свойствами, включая даже некоторые из самых первых созданных компьютеров. Соответственно, мы сможем рассказать вам историю в историческом контексте. Представьте мир, в котором нет компьютеров; представьте, с каким энтузиазмом было бы встречено такое устройство, — и вы будете не так уж далеки от текущего состояния дел! Мы расскажем свою историю с точки зрения научных вычислений, однако существует не менее увлекательная история с точки зрения коммерческих вычислений, которую мы лишь кратко затронем.

Затем мы постараемся убедить вас в том, что базовые концепции и конструкции, которые были рассмотрены применительно к программированию на языке Java, без особых сложностей реализуются на простой машине с собственным *машинным языком*. Мы подробно рассмотрим условные переходы, циклы, функции, массивы и связные структуры данных. Так как эти же базовые средства были рассмотрены для Java, из этого следует, что некоторые вычислительные задачи, которые были представлены в первой части книги, относительно легко реализуются и на более низком уровне.

Эта простая машина — логическое звено, связывающее ваш компьютер с реальными физическими схемами, изменяющими свое состояние в соответствии с действиями ваших программ. Она поможет вам понять, как работают эти схемы (об этом речь пойдет в следующей главе).

И это еще не все. В завершение этой главы будет рассмотрена фундаментальная идея: одна машина может использоваться для моделирования работы другой машины. А это означает, что мы можем легко изучать воображаемые машины, разрабатывать новые машины, которые будут построены в будущем, и работать с машинами, которые, возможно, никогда не будут построены.

6.1. Представление информации

Если вы хотите понять, как работает компьютер, для начала нужно разобраться в том, как в нем представляется информация. Как известно из программирования на Java, все данные, обрабатываемые цифровыми компьютерами, представляются последовательностью из 0 и 1, будь то числа, текст, исполняемые файлы, графические изображения, аудио- или видеоданные. Для каждого типа данных появились стандартные методы кодирования: стандарт ASCII связывает 7-разрядное двоичное число с каждым из 128 поддерживаемых символов; формат MP3 определяет, как закодировать исходный звуковой файл последовательностью из 0 и 1; графический формат `.png` определяет пикселы цифрового изображения, которые в конечном итоге тоже представляются последовательностью из 0 и 1, и т. д.

Во внутреннем строении компьютера информация чаще всего упорядочивается и структурируется в виде *машинных слов*, которые представляют собой не что иное, как последовательности битов (разрядов) фиксированной длины (известной как *размер слова*, или *разрядность*). Как вы вскоре узнаете, разрядность играет важнейшую роль в архитектуре любого компьютера. Для первых компьютеров типичная разрядность составляла 12 или 16; на протяжении многих лет широко использовались 32-разрядные слова; в наши дни нормой стали 64-разрядные слова[1].

Информация, хранящаяся в каждом компьютере, — всего лишь последовательность слов. Каждое слово состоит из фиксированного количества разрядов, каждый из которых равен 0 или 1. Так как каждое слово может интерпретироваться как число, записанное в двоичной системе счисления, вся информация состоит из чисел, а числа образуют информацию.

Смысл заданной последовательности битов в компьютере зависит от контекста. Это очередная мантра, которая будет неоднократно повторяться в этой главе. Например, как вы увидите, в зависимости от контекста двоичная строка `1111101011001110` может интерпретироваться как положительное целое число 64 206, как отрицательное целое число –1330, как вещественное число –55744,0 или как строка из двух символов `"eN"`.

Какой бы удобной ни была двоичная система для компьютеров, для людей она в высшей степени неудобна. Если вы еще не убеждены в этом, попробуйте запомнить 16-разрядное двоичное число `1111101011001110`, закрыть книгу и записать его без ошибок. Чтобы совместить интересы компьютера, работающего с двоичными данными, с желанием использовать более компактное представление, в этом разделе мы введем *шестнадцатеричную* систему счисления (по основанию 16), достаточно удобную для работы с двоичными данными. Итак, для начала рассмотрим шестнадцатеричную систему.

[1] Первые микрокомпьютеры (персональные компьютеры) были 8-разрядными, при том что даже среди мини-ЭВМ того времени уже преобладали 32-разрядные архитектуры. — *Примеч. науч. ред.*

Двоичная и шестнадцатеричная запись

Пока ограничимся *натуральными* (то есть неотрицательными целыми[1]) числами — важнейшей математической абстракцией для счета. Еще в Древнем Вавилоне люди представляли целые числа в позиционной записи с фиксированным основанием. Конечно, нам лучше всего знакома *десятичная* система счисления (с основанием 10), в которой каждое положительное целое число представлено последовательностью цифр от 0 до 9. А точнее, запись $d_n d_{n-1} ... d_2 d_1 d_0$ представляет целое число

$$d_n 10_n + d_{n-1} 10_n^{-1} + ... + d_2 10^2 + d_1 10^1 + d_0 10^0.$$

Например, запись 10345 представляет целое число

$$10\,345 = 1{\cdot}10\,000 + 0{\cdot}1000 + 3{\cdot}100 + 4{\cdot}10 + 5{\cdot}1.$$

Заменив основание 10 любым целым числом, большим 1, вы получите другую систему счисления, в которой любое целое число представляется последовательностью цифр в диапазоне от 0 до числа, на 1 меньшего основания. Для наших целей особый интерес представляют две системы: двоичная (основание 2) и шестнадцатеричная (основание 16).

Двоичная система

С основанием 2 целые числа представляются последовательностями 0 и 1. В этом случае каждая двоичная (по основанию 2) цифра — 0 или 1 — называется *битом* (разрядом двоичного слова). Биты закладывают основу представления информации в компьютерах. В двоичной системе биты служат коэффициентами степеней 2, а именно последовательность битов $b_n b_{n-1} ... b_2 b_1 b_0$ представляет целое число

$$b_n 2^n + b_{n-1} 2^{n-1} + ... + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

Например, число `1100011` представляет целое число

$$99 = 1{\cdot}64 + 1{\cdot}32 + 0{\cdot}16 + 0{\cdot}8 + 0{\cdot}4 + 1{\cdot}2 + 1{\cdot}1.$$

Самое большое целое число, которое может быть представлено n-разрядным словом, равно 2^{n-1}; в таком числе каждый из n битов равен 1. Например, 8 битов `11111111` представляет число

$$2^8 - 1 = 255 = 1{\cdot}128 + 1{\cdot}64 + 1{\cdot}32 + 1{\cdot}16 + 1{\cdot}8 + 1{\cdot}4 + 1{\cdot}2 + 1{\cdot}1.$$

Это ограничение также можно сформулировать в другом виде: n-разрядное слово может представлять только 2^n неотрицательных целых числа (от 0 до 2^{n-1}). Такие ограничения часто приходится учитывать при обработке целых чисел на компью-

[1] В нашей традиции ноль обычно не считается натуральным числом, поэтому корректнее говорить о *расширенном* ряде натуральных чисел, который включает также и ноль. — *Примеч. науч. ред.*

тере. Серьезный недостаток двоичной записи заключается в том, что количество двоичных цифр, необходимых для представления числа, намного больше, скажем, количества цифр, необходимого для представления того же числа в десятичной форме. Поэтому использовать двоичную запись именно для взаимодействия с компьютером было бы неудобно и непрактично.

Шестнадцатеричная система

В шестнадцатеричной системе последовательность шестнадцатеричных цифр $h_n h_{n-1} ... h_2 h_1 h_0$ представляет целое число

$$h_n 16^n + h_{n-1} 16^{n-1} + ... + h_2 16^2 + h_{11} 6^1 + h_0 16^0$$

Первое затруднение: в системе с основанием 16 нужны цифры для всех значений от 0 до 15. Каждая цифра представляется одним символом, поэтому 10 обозначается буквой A, 11 — буквой B, 12 — буквой C и т. д. (см. таблицу на с. 842). Например, запись `FACE` представляет целое число

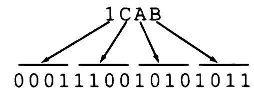
$$64\,206 = 15{\cdot}163 + 10{\cdot}162 + 12{\cdot}161 + 14{\cdot}160$$

Перед вами то же целое число, которое было ранее записано 16 двоичными цифрами. Как видно из примера, количество шестнадцатеричных цифр, необходимых для представления целых чисел в шестнадцатеричном виде, составляет лишь небольшую часть (примерно 1/4) от количества двоичных цифр, необходимых для представления того же числа. Кроме того, разнообразие цифр упрощает запоминание числа. Даже если у вас возникли трудности с запоминанием **`1111101011001110`**, безусловно, вы легко запомните запись `FACE`.

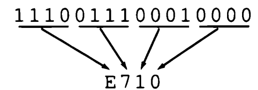
Преобразование между шестнадцатеричной и двоичной записью

Вы можете без труда получить двоичное представление для заданного шестнадцатеричного представления — или наоборот (см. справа). Так как основание шестнадцатеричной системы 16 равно степени основания двоичной системы 2, каждая группа из 4 битов преобразуется в шестнадцатеричную цифру, и наоборот. Чтобы преобразовать шестнадцатеричное число в двоичное, замените каждую шестнадцатеричную цифру четырьмя двоичными, соответствующими ее значению (см. таблицу на с. 842). И наоборот, если задано двоичное представление числа, добавьте начальные 0, чтобы количество двоичных цифр было кратно 4, а затем сгруппируйте их по 4 и замените каждую группу одной шестнадцатеричной цифрой. Вы можете провести математические вычисления и убедиться в том, что такие преобразования всегда верны (см. упражнение 6.1.8), но даже просто несколько примеров позволяют все прояснить.

из шестнадцатеричной в двоичную



из двоичной в шестнадцатеричную



Примеры преобразования между шестнадцатеричной и двоичной системами счисления

десятичная запись	двоичная запись	шестнадцатеричная запись
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Представление целых чисел от 0 до 15

Например, шестнадцатеричное представление целого числа 39 равно 27 , поэтому двоичное представление числа равно `00100111` (начальные нули можно отбросить); двоичное представление числа 228 равно `11100100`, поэтому в шестнадцатеричной записи оно принимает вид `E4`. Возможность быстрого преобразования между двоичной и шестнадцатеричной системами важна тем, что она позволяет эффективно общаться с компьютером. Вы очень быстро освоите этот навык; нужно лишь запомнить, что цифра A эквивалентна 1010, цифра 5 — 0101, цифра F — 1111 и т. д.

Преобразование из десятичной системы в двоичную

Задача получения числового значения строки из 0 и 1, представляющей двоичное число, рассматривалась в одном из наших ранних упражнений по программированию. К следующей рекурсивной программе (решение упражнения 2.3.15), решающей эту задачу, стоит присмотреться повнимательнее[1]:

[1] Машинно-ориентированные вычисления средствами языка высокого уровня обычно реализуются неэффективно (причины упоминаются в тексте). — *Примеч. науч. ред.*

```
public static String toBinaryString(int n)
{
   if (n == 0) return "";
   if (n % 2 == 0)
      return toBinaryString(n/2) + '0';
   else
      return toBinaryString(n/2) + '1';
}
```

Этот рекурсивный метод основан на той идее, что последней цифрой является символ, представляющий значение остатка от деления `n%2` (`'0'`, если `n%2` равно 0, или `'1'`, если `n%2` равно 1), а оставшаяся часть строки является строковым представлением результата деления `n/2`. Пример трассировки программы показан ниже. Метод обобщается для шестнадцатеричной системы (или системы с любым другим основанием). Кроме того, нас интересует задача преобразования строковых представлений в значения типов данных Java. В следующем разделе приведена программа, которая реализует такие преобразования.

```
toBinaryString(109)
   toBinaryString(54)
      toBinaryString(27)
         toBinaryString(13)
            toBinaryString(6)
               toBinaryString(3)
                  toBinaryString(1)
                     toBinaryString(0)
                        return ""
                     return "1"
                  return "11"
               return "110"
            return "1101"
         return "11011"
      return "110110"
   return "1101101"
```

Трассировка вызовов для toBinaryString(109)

Говоря о том, что происходит на компьютере, мы говорим на шестнадцатеричном языке. Содержимое n-разрядного компьютерного слова можно описать $n/4$ шестнадцатеричными цифрами и при необходимости мгновенно преобразовать его снова в двоичную систему. Возможно, вы уже встречали нечто подобное в повседневной жизни. Например, при регистрации нового устройства в Сети необходимо знать его *MAC-адрес*. MAC-адрес вида *1a:ed:b1:b9:96:5e* представляет собой шестнадцатеричную сокращенную запись (с дополнительными двоеточиями и буквами $a-f$ в нижнем регистре вместо букв $A-F$ в верхнем, которые используем мы) 48-разрядного двоичного числа, которое идентифицирует ваше устройство в Сети.

дес.	дв.	шес.	дес.	дв.	шес.	дес.	дв.	шес.	дес.	дв.	шес.
0	00000000	00	32	00100000	20	64	01000000	40	96	01100000	60
1	00000001	01	33	00100001	21	65	01000001	41	97	01100001	61
2	00000010	02	34	00100010	22	66	01000010	42	98	01100010	62
3	00000011	03	35	00100011	23	67	01000011	43	99	01100011	63
4	00000100	04	36	00100100	24	68	01000100	44	100	01100100	64
5	00000101	05	37	00100101	25	69	01000101	45	101	01100101	65
6	00000110	06	38	00100110	26	70	01000110	46	102	01100110	66
7	00000111	07	39	00100111	27	71	01000111	47	103	01100111	67
8	00001000	08	40	00101000	28	72	01001000	48	104	01101000	68
9	00001001	09	41	00101001	29	73	01001001	49	105	01101001	69
10	00001010	0A	42	00101010	2A	74	01001010	4A	106	01101010	6A
11	00001011	0B	43	00101011	2B	75	01001011	4B	107	01101011	6B
12	00001100	0C	44	00101100	2C	76	01001100	4C	108	01101100	6C
13	00001101	0D	45	00101101	2D	77	01001101	4D	109	01101101	6D
14	00001110	0E	46	00101110	2E	78	01001110	4E	110	01101110	6E
15	00001111	0F	47	00101111	2F	79	01001111	4F	111	01101111	6F
16	00010000	10	48	00110000	30	80	01010000	50	112	01110000	70
17	00010001	11	49	00110001	31	81	01010001	51	113	01110001	71
18	00010010	12	50	00110010	32	82	01010010	52	114	01110010	72
19	00010011	13	51	00110011	33	83	01010011	53	115	01110011	73
20	00010100	14	52	00110100	34	84	01010100	54	116	01110100	74
21	00010101	15	53	00110101	35	85	01010101	55	117	01110101	75
22	00010110	16	54	00110110	36	86	01010110	56	118	01110110	76
23	00010111	17	55	00110111	37	87	01010111	57	119	01110111	77
24	00011000	18	56	00111000	38	88	01011000	58	120	01111000	78
25	00011001	19	57	00111001	39	89	01011001	59	121	01111001	79
26	00011010	1A	58	00111010	3A	90	01011010	5A	122	01111010	7A
27	00011011	1B	59	00111011	3B	91	01011011	5B	123	01111011	7B
28	00011100	1C	60	00111100	3C	92	01011100	5C	124	01111100	7C
29	00011101	1D	61	00111101	3D	93	01011101	5D	125	01111101	7D
30	00011110	1E	62	00111110	3E	94	01011110	5E	126	01111110	7E
31	00011111	1F	63	00111111	3F	95	01011111	5F	127	01111111	7F

Представление целых чисел от 0 до 127 в десятичной, двоичной (8 бит) и шестнадцатеричной (2 цифры) записи

дес.	дв.	шес.	дес.	дв.	шес.	дес.	дв.	шес.	дес.	дв.	шес.
128	10000000	80	160	10100000	A0	192	11000000	C0	224	11100000	E0
129	10000001	81	161	10100001	A1	193	11000001	C1	225	11100001	E1
130	10000010	82	162	10100010	A2	194	11000010	C2	226	11100010	E2
131	10000011	83	163	10100011	A3	195	11000011	C3	227	11100011	E3
132	10000100	84	164	10100100	A4	196	11000100	C4	228	11100100	E4
133	10000101	85	165	10100101	A5	197	11000101	C5	229	11100101	E5
134	10000110	86	166	10100110	A6	198	11000110	C6	230	11100110	E6
135	10000111	87	167	10100111	A7	199	11000111	C7	231	11100111	E7
136	10001000	88	168	10101000	A8	200	11001000	C8	232	11101000	E8
137	10001001	89	169	10101001	A9	201	11001001	C9	233	11101001	E9
138	10001010	8A	170	10101010	AA	202	11001010	CA	234	11101010	EA
139	10001011	8B	171	10101011	AB	203	11001011	CB	235	11101011	EB
140	10001100	8C	172	10101100	AC	204	11001100	CC	236	11101100	EC
141	10001101	8D	173	10101101	AD	205	11001101	CD	237	11101101	ED
142	10001110	8E	174	10101110	AE	206	11001110	CE	238	11101110	EE
143	10001111	8F	175	10101111	AF	207	11001111	CF	239	11101111	EF
144	10010000	90	176	10110000	B0	208	11010000	D0	240	11110000	F0
145	10010001	91	177	10110001	B1	209	11010001	D1	241	11110001	F1
146	10010010	92	178	10110010	B2	210	11010010	D2	242	11110010	F2
147	10010011	93	179	10110011	B3	211	11010011	D3	243	11110011	F3
148	10010100	94	180	10110100	B4	212	11010100	D4	244	11110100	F4
149	10010101	95	181	10110101	B5	213	11010101	D5	245	11110101	F5
150	10010110	96	182	10110110	B6	214	11010110	D6	246	11110110	F6
151	10010111	97	183	10110111	B7	215	11010111	D7	247	11110111	F7
152	10011000	98	184	10111000	B8	216	11011000	D8	248	11111000	F8
153	10011001	99	185	10111001	B9	217	11011001	D9	249	11111001	F9
154	10011010	9A	186	10111010	BA	218	11011010	DA	250	11111010	FA
155	10011011	9B	187	10111011	BB	219	11011011	DB	251	11111011	FB
156	10011100	9C	188	10111100	BC	220	11011100	DC	252	11111100	FC
157	10011101	9D	189	10111101	BD	221	11011101	DD	253	11111101	FD
158	10011110	9E	190	10111110	BE	222	11011110	DE	254	11111110	FE
159	10011111	9F	191	10111111	BF	223	11011111	DF	255	11111111	FF

Представление целых чисел от 128 до 255 в десятичной, двоичной (8 бит) и шестнадцатеричной (2 цифры) записи

Далее в этой главе для нас будут представлять особый интерес целые числа, меньшие 256, которые могут быть представлены 8 битами (двоичными цифрами) или двумя шестнадцатеричными цифрами. Для справки мы привели на двух предыдущих страницах полную таблицу записи этих чисел в десятичной, двоичной и шестнадцатеричной системах. Не жалейте времени на изучение таблицы, постарайтесь понять связь между этими формами записи и научитесь уверенно работать с такими числами. Если после этого вам покажется, что все это и так понятно, а таблица только напрасно занимает место, — значит, мы достигли своей цели!

Разбор и формирование строковых представлений

Преобразования между разными представлениями целых чисел — интересная вычислительная задача, с которой мы впервые столкнулись в листинге 1.3.7, а затем вернулись к ней в упражнении 2.3.15. Также для решения этой задачи постоянно использовались методы Java. Чтобы закрепить идею о представлении чисел в позиционных системах счисления с различными основаниями, рассмотрим программу для преобразования произвольного числа из одной системы счисления в другую.

Разбор

Преобразование строки символов во внутреннее представление называется *разбором* (или *парсингом*). С раздела 1.1 мы использовали методы Java (такие, как `Integer.parseInt()`) и наши собственные методы (такие, как `StdIn.readInt()`) для преобразования строк, вводимых с клавиатуры, в значения типов данных Java. Ранее мы работали с десятичными числами (представленными строками символов от 0 до 9), а теперь рассмотрим метод для разбора чисел, записанных в произвольной системе счисления. Для простоты мы ограничимся основаниями, не превышающими 36, и расширим систему обозначений по аналогии с шестнадцатеричной записью путем использования букв A–Z для представления цифр от 10 до 35. *Примечание*: класс Java `Integer` содержит метод `parseInt()` с двумя аргументами, который обладает схожей функциональностью (если не считать того, что он работает также и с отрицательными числами).

i	n	обработанные символы
0	1	1
1	3	11
2	6	110
3	13	1101
4	27	11011
5	54	110110
6	109	1101101

Парсинг parseInt(1101101, 2)

Листинг 6.1.1. Преобразование натурального числа в другую систему счисления

```
public class Convert
{
   public static int toInt(char c)
   {  /* См. упражнение 6.1.12. */  }

   public static char toChar(int i)
   {  /* См. упражнение 6.1.13. */  }

   public static int parseInt(String s, int d)
   {
      int n = 0;
      for (int i = 0; i < s.length(); i++)
         n = d*n + toInt(s.charAt(i));
      return n;
   }

   public static String toString(int n, int d)
   {
      if (n == 0) return "";
      return toString(n/d, d) + toChar(n % d);
   }

   public static void main(String[] args)
   {
      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         int baseFrom = StdIn.readInt();
         int baseTo = StdIn.readInt();
         int n = parseInt(s, baseFrom);
         StdOut.println(toString(n, baseTo));
      }
   }
}
```

```
% java Convert
1101101 2 10
109

FACE 16 10
64206

FACE 16 2
1111101011001110

109 10 2
1101101

64206 10 16
FACE

64206 10 32
1UME

1UME 32 10
64206
```

Программа преобразования читает строки и пары оснований из стандартного потока ввода и при помощи методов parseInt() и toString() преобразует строку с представлением целого числа из системы счисления с первым основанием в систему со вторым основанием.

Одна из характерных особенностей современных типов данных заключается в том, что внутреннее представление скрыто от пользователя, поэтому для достижения желаемого результата со значениями типа данных можно использовать только определенные для этого типа операции. В частности, следует исключить прямые обращения к битам, представляющим значение типа данных, и вместо них использовать операции типа данных.

Первая примитивная операция, которая понадобится для разбора чисел, — метод для преобразования символа в целое число. В упражнении 6.1.12 приведен метод `toInt()`, который получает в аргументе символ из диапазона 0–9 или A–Z и возвра-

щает значение `int` от 0 до 35 (0–9 для цифр, 10–35 для букв). С этим примитивом простой метод `parseInt()` из листинга 6.1.1 разбирает строковое представление целого числа в системе с произвольным основанием `b` (от 2 до 36) и возвращает полученное число в виде значения Java `int`. Как обычно, вы можете убедиться в этом, проанализировав код внутри цикла. На каждой итерации значение `n` типа `int` содержит целое число, представляющее все цифры, встреченные до настоящего момента. Для этого достаточно умножить результат на основание и прибавить следующую цифру. Приведенная трассировка поясняет смысл происходящего: для получения нового значения `n` предыдущее значение `n` умножается на основание исходной системы счисления, а к результату прибавляется следующая цифра (выделенная жирным шрифтом). Скажем, чтобы разобрать строку **1101101**, мы вычисляем $0 \cdot 2 + 1 = 1, 1 \cdot 2 + 1 = 3, 3 \cdot 2 + 0 = 6, 6 \cdot 2 + 1 = 13, 13 \cdot 2 + 1 = 27, 27 \cdot 2 + 0 = 54$ и $54 \cdot 2 + 1 = 109$. А при разборе `FACE` как шестнадцатеричного числа мы вычисляем $0 \cdot 16 + 15 = 15, 15 \cdot 16 + 10 = 250, 250 \cdot 16 + 12 = 4012$, и $4012 \cdot 16 + 14 = 64\,206$. Это особый случай вычисления многочлена *с использованием схемы Горнера* (см. упражнение 2.1.31).

i	n	обнаруженные символы
0	15	"**F**"
1	250	"F**A**"
2	4012	"FA**C**"
3	64206	"FAC**E**"

Трассировка parseInt(FACE, 16)

Для простоты мы не стали включать в этот код проверку ошибок. Например, метод `parseInt()` должен выдавать исключение, если значение, возвращенное `toInt()`, не меньше основания. Кроме того, он должен выдавать исключение при переполнении, так как программе может быть передана строка с числом большим, чем позволяет представить тип Java `int`.

Строковое представление

Использование метода `toString()` для вычисления строкового представления значения типа данных — еще один прием, которым мы пользовались с самого начала книги. Мы используем рекурсивный метод, который обобщает метод преобразования из десятичной системы в двоичную (решение упражнения 2.3.15), рассмотренный ранее в этом разделе. Кроме того, будет полезно рассмотреть метод для получения строкового представления целого числа в любой заданной системе счисления, хотя класс Java `Integer` содержит метод `toString()` с двумя аргументами, обладающий схожей функциональностью.

И снова первой примитивной операцией, которая нам понадобится, станет метод для преобразования целого числа в символ (цифру). В упражнении 6.1.13 приводится метод `toChar()`, который получает значение `int` в диапазоне от 0 до 35 и возвращает символ в диапазоне 0–9 (для значений меньше 10) или A–Z (для значений от 10 до 35). При наличии такого примитива метод `toString()` в листинге 6.1.1 реализуется еще проще, чем `parseInt()`. Этот рекурсивный метод основан на том, что последняя цифра является символьным представлением остатка от деления `n%d`, а оставшаяся часть строки содержит строковое представление частного `n/d`. Вычисления, по сути, воспроизводят процесс разбора в обратном направлении, как видно из трассировки.

```
toString(64206, 16)
  toString(4012, 16)
    toString(250, 16)
      toString(15, 16)
        toString(0, 16)
          return ""
        return "F"
      return "FA"
    return "FAC"
  return "FACE"
```

Трассировка вызовов для toString(64206, 16)

Говоря о содержимом машинных слов, необходимо включить и незначащие лидирующие нули, поэтому мы заполняем все двоичные цифры. По этой причине в `Convert` включается версия `toString()` с тремя аргументами; в третьем аргументе передается желаемое количество цифр в возвращенной строке. Например, вызов `toString(15, 16, 4)` возвращает `000F`, а вызов `toString(14, 2, 16)` возвращает `0000000000001110`. Реализация этой версии остается читателю для самостоятельной работы (см. упражнение 6.1.15).

Программа в листинге 6.1.1 — универсальный инструмент для перевода чисел между разными системами счисления, который сводит воедино все описанные концепции. Пока стандартный поток ввода не пуст, главный цикл в тестовом приложении-клиенте читает из него строку, за которой следуют два целых числа (основания двух систем счисления: для исходной строки и для результата). Программа выполняет заданное преобразование и выводит результат. Для этого она при помощи `parseInt()` преобразует входную строку в значение Java `int`, а затем использует `toString()` для преобразования этого числа в его строковое представление в заданной системе счисления. Загрузите программу и поэкспериментируйте с ней, чтобы освоить принципы представления и преобразования чисел.

Целочисленная арифметика

Знакомство с операциями над целыми числами мы начнем с основных арифметических операций, таких как сложение и умножение. Более того, ранние вычислительные устройства в основном и предназначались для многократного выполнения таких операций. В следующей главе будет рассмотрена концепция построения вычислительных устройств, которые умеют выполнять арифметические операции, так как в каждом компьютере имеется встроенное оборудование для их выполнения. А пока мы покажем, что основные операции с числами, которые вы изучали в школе для десятичной записи, прекрасно работают в двоичной и в шестнадцатеричной системах.

Сложение

В школе вы научились складывать десятичные числа. Сначала нужно сложить две младшие (крайние правые) цифры; если сумма больше 10, вы записываете остаток от деления суммы на 10 и переносите 1 в следующей разряд. Затем то же самое делается со следующими цифрами, но на этот раз с учетом переноса из предыдущего разряда. Эта процедура обобщается для любого основания. Например, если числа записаны в шестнадцатеричной системе, то при суммировании цифр `7` и `E` следует записать `5` и перенести `1` в следующий разряд, потому что `7+E` в шестнадцатеричной записи равно `15`. Если же числа записаны в двоичной системе, то суммирование двух разрядов, содержащих `1`, с переносом `1` дает `1` с переносом `1` в следующий разряд, потому что в двоичной записи `1+1=11`. Примеры справа демонстрируют вычисление суммы $4567_{10} + 366_{10} = 4933_{10}$ в десятичной, шестнадцатеричной и двоичной системах счисления. Как и в школе, незначащие лидирующие нули не записываются.

десятичная система

```
0011   ← перенос
4567
 366
----
4933
```

шестнадцатеричная система

```
0011
11D7
 16E
----
1345
```

двоичная система

```
0000111111110
 1000111010111
     101101110
 -------------
 1001101000101
```

Сложение

Целые числа без знака

Если вы хотите работать с целыми числами в компьютерном мире, вам приходится мириться с ограничениями на количество и размер представляемых целых чисел. Как упоминалось ранее, n-разрядное слово может представить всего 2^n целых числа. Если вас интересуют только неотрицательные целые числа (числа без знака), будет естественно использовать двоичное представление для целых чисел от 0 до 2^{n-1} с добавлением лидирующих нулей, чтобы каждое слово соответствовало целому числу, а каждое целое число в определенном диапазоне соответствовало слову. В таблице на с. 851 показаны 16 беззнаковых целых чисел, которые могут быть представлены 4-разрядным словом, а в таблице внизу — диапазоны беззнаковых целых чисел, представимых 16-, 32- и 64-разрядными словами, которые используются в типичных компьютерах.

длина в битах	наименьшее число	наибольшее число
4	0	15
16	0	65535
32	0	4 294 967 295
64	0	18 446 744 073 709 551 615

Диапазоны беззнаковых целых чисел, представимых словами разной длины

десятичная запись	двоичная запись
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

4-разрядные беззнаковые целые числа

Переполнение

перенос в этом разряде — признак переполнения

```
↓
1000000111111000
 1111111111111000
 0000000000001000
 ----------------
 0000000000000000
```

Перенос (16-разрядное беззнаковое целое число)

Как упоминалось в разделе 1.2, при программирования на языке Java вы должны следить за тем, чтобы результат арифметических операций не превышал максимального представимого значения. Такое превышение называется *переполнением*. При сложении целых чисел без знака переполнение выявляется легко: если сложение в старшем (крайнем левом) разряде приводит к переносу, значит, результат слишком велик для такого представления. Значение одного бита легко проверяется даже на аппаратном уровне (как вы вскоре увидите), поэтому и в компьютерах, и в языках программирования обычно предусмотрены низкоуровневые операции для проверки возможного переполнения. Как ни странно, в Java такой проверки нет (см. раздел 1.2)[1].

[1] Факт переполнения обычно отмечается в регистре *статуса* или *флагов* процессора. В языках высокого уровня доступ к ним действительно часто не предусмотрен и переполнение можно обнаружить (или предсказать) лишь по косвенным признакам. — *Примеч. науч. ред.*

Умножение

Как и в случае со сложением, школьный алгоритм умножения прекрасно работает в других системах счисления (в двоичном примере трудно разобраться из-за каскадных переносов: если вы захотите проследить за происходящим, складывайте числа попарно). Специалисты разработали алгоритмы умножения, которые намного лучше подходят для машинной реализации и значительно превосходят этот метод по эффективности. В ранних компьютерах программистам приходилось выполнять умножение на программном уровне (такая реализация будет продемонстрирована позднее, в упражнении 6.3.35). Обратите внимание: при разработке алгоритмов умножения переполнение создает гораздо больше проблем, чем при сложении, потому что разрядность результата может быть вдвое больше разрядности операндов. Следовательно, при умножении двух n-разрядных чисел следует приготовиться к получению $2n$-разрядного результата.

десятичная система

```
     4567
   *  366
    27402
   27402
  13701
  1671522
```

шестнадцатеричная система

```
     11D7
   *  16E
     F9C2
    6B0A
   11D7
   198162
```

двоичная система

```
          1000111010111
        * 0000101101110
         1000111010111
        1000111010111
       1000111010111
     1000111010111
    1000111010111
  1000111010111
  11001100000001011100010
```

Примеры умножения

Конечно, в этой книге нам не удастся подробно описать все методы, которые были разработаны для выполнения арифметических операций в компьютере. Конечно, компьютер должен эффективно выполнять деление, возведение в степень и другие операции. Мы во всех подробностях опишем сложение/вычитание, а также приведем краткую информацию о других операциях.

Также необходимо иметь возможность выполнять вычисления с отрицательными и вещественными числами. Далее кратко описаны стандартные представления для выполнения таких операций.

Отрицательные числа

Разработчики компьютеров быстро поняли, что целочисленный тип данных можно легко приспособить для работы с отрицательными числами. Для этого используется представление, известное как *дополнение до 2*, или *дополнительный код*.

Вероятно, первое, что приходит в голову, — представление в *прямом коде*, в котором первый (старший) разряд числа определяет знак, а остальные — модуль (абсолютное значение). Например, если представление состоит из 4 битов, последовательность **0101** соответствует числу +5, а последовательность **1101** — числу −5.

А в n-разрядном дополнительном коде положительные числа представляются так же, как прежде, но каждое отрицательное число $-x$ представляется двоичным числом $2^n - x$ (положительным без знака). Например, в таблице на с. 854 показаны два целых числа в дополнительном коде, которые могут быть представлены 4-разрядным словом. Вы видите, что последовательность `0101` по-прежнему представляет число +5, но `1011` представляет −5, потому что $2^4 - 5 = 11_{10} = 1011_2$.

Если один разряд резервируется для знака, то самое большое число, которое может быть представлено в дополнительном коде, равно примерно половине самого большого числа без знака, представимого словом той же разрядности. Как видно из примера с 4-разрядным словом, в дополнительном коде существует небольшая асимметрия: в нем представляются положительные числа от 1 до 7, отрицательные от −8 до −1 и существует единственное представление для 0. В общем случае в n-разрядном дополнительном коде наименьшее возможное отрицательное число равно -2^{n-1}, а самое большое положительное число равно $2^{n-1} - 1$. В таблице на с. 854 представлены наименьшее и наибольшее (по модулю) 16-разрядные целые числа в дополнительном коде.

десятичная запись	двоичная запись
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
−8	1000
−7	1001
−6	1010
−5	1011
−4	1100
−3	1101
−2	1110
−1	1111

4-разрядные целые числа (дополнительный код)

Тот факт, что дополнительный код фактически стал стандартной заменой прямого кода, объясняется двумя причинами. Во-первых, так как у 0 существует всего одно представление (двоичная строка, состоящая только из 0), проверка значения на равенство 0 реализуется предельно просто. Во-вторых, арифметические операции

с этим представлением тоже легко реализуются — пример реализации для сложения приводится позднее в этом разделе. Кроме того, как и в случае с прямым кодом, первый разряд определяет знак, а значит, и проверка значения на отрицательность тоже реализуется просто. Проектирование аппаратного обеспечения компьютера — задача достаточно сложная, поэтому возможность реализовать это упрощение за счет правил представления чисел выглядит заманчиво.

десятичная запись	двоичная запись
0	0000000000000000
1	0000000000000001
2	0000000000000010
3	0000000000000011
...	...
32765	0111111111111101
32766	0111111111111110
32767	0111111111111111
–32768	1000000000000000
–32767	1000000000000001
–32766	1000000000000010
–32765	1000000000000011
...	...
–3	1111111111111101
–2	1111111111111110
–1	1111111111111111

16-разрядные целые числа (дополнительный код)

Сложение

Сложение двух n-разрядных целых чисел в дополнительном коде реализуется просто: числа просто суммируются так, как если бы они были беззнаковыми. Например, 2 + (–7) = `0010` + `1001` = `1011` = –5. Доказать, что эта реализация всегда работает, если результат находится в диапазоне допустимых значений (от -2^{n-1} и $2^{n-1} - 1$), несложно.

- Если оба числа неотрицательные, то действуют правила стандартного двоичного сложения (при условии, что результат меньше 2^{n-1}).
- Если оба числа отрицательные, то сумма равна

$$(2^n - x) + (2^n - y) = 2^n + 2^n - (x + y)$$

- Если число x отрицательное, число y положительное, а результат отрицательный, имеем

$$(2^n - x) + y = 2^n - (x - y).$$

- Если число x отрицательное, число y положительное, а результат положительный, имеем

$$(2^n - x) + y = 2^n + (y - x).$$

Во втором и в четвертом случае дополнительное слагаемое 2^n не влияет на n-разрядный результат (перенос за пределы разрядной сетки), поэтому стандартное двоичное сложение (с игнорированием переноса) дает желаемый результат. Обнаружить переполнение становится немного сложнее, чем с двумя беззнаковыми числами, — мы вернемся к этому в разделе «Вопросы и ответы».

```
0000000000000000
 0000000001000000      64
 0000000000101010     +42
 ----------------     ---
 0000000001101010     106

1111111111000000
 0000000001000000      64
 1111111111010110     -42
 ----------------     ---
 0000000000010110      22

0000000000000000
 1111111111000000     -64
 0000000000101010     +42
 ----------------     ---
 1111111111101010     -22

1111111111000000
 1111111111000000     -64
 1111111111010110     -42
 ----------------     ---
 1111111110010110    -106
```

Сложение (16-разрядный дополнительный код)

Вычитание

Чтобы вычислить $x - y$, мы вычисляем $x + (-y)$. Иначе говоря, для вычитания можно воспользоваться стандартным двоичным сложением, если мы будем знать, как получить $-y$. Как выясняется, изменение знака числа в дополнительном коде реализуется очень просто: следует изменить значения всех битов, а затем прибавить 1. На с. 856 (вверху) изображены три примера этого процесса — доказательство того, что этот способ работает, предоставляется читателю для самостоятельной работы.

Дополнительный код актуален для программистов Java, потому что значения типов `short`, `int` и `long` представляются соответственно 16-, 32- и 64-разрядными

целыми числами в дополнительном коде. Это объясняет границы допустимых значений, которые должны знать программисты Java (в нижней таблице).

```
 0000000000001010      10
 1111111111110101    изменяем значения всех разрядов
+0000000000000001    прибавляем 1
 1111111111110110     -10

 1111111111110110     -10
 0000000000001001    изменяем значения всех разрядов
+0000000000000001    прибавляем 1
 0000000000001010      10

 0001001101001110    4942
 1110110010110001    изменяем значения всех разрядов
+0000000000000001    прибавляем 1
 1110110010110010   -4942
```

Смена знака у чисел в дополнительном коде

16-разрядное	*наименьшее*	– 32,768
	наибольшее	32,767
32-разрядное	*наименьшее*	– 2,147,483,648
	наибольшее	2,147,483,647
64-разрядное	*наименьшее*	– 9,223,372,036,854,775,808
	наибольшее	9,223,372,036,854,775,807

Диапазоны представления целых чисел в дополнительном коде

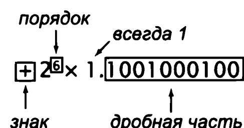
Кроме того, использование дополнительного кода в языке Java объясняет поведение переполнения, которое мы впервые наблюдали в разделе 1.2 (см. подраздел «Вопросы и ответы» этого раздела и упражнение 1.2.10). Например, вы видели, что, если прибавить 1 к самому большому положительному целому числу, вы получите самое большое (по модулю) отрицательное число. В 4-разрядном дополнительном коде увеличение `0111` на 1 дает `1000`; в 16-разрядном дополнительном коде увеличение `0111111111111111` на 1 дает `1000000000000000`. (Обратите внимание: это единственный случай, в котором увеличение целого числа в дополнительном коде на 1 не приводит к ожидаемому результату.) Поведение других случаев в упражнении 1.2.10 также легко объяснимо. Многие годы это поведение приводило в замешательство программистов, которые не желали утруждать себя изучением дополнительного кода. Вот вам убедительный пример: в Java вызов `Math.abs(-2147483648)` возвращает `-2147483648`, отрицательное число!

Вещественные числа

Как на компьютере представляются вещественные числа? Эта задача немного сложнее представления целых чисел, так как потребует выбора из многих вариантов. На первых порах разработчики аппаратного обеспечения компьютеров опробовали великое множество конкурирующих форматов, разработанных за первые десятилетия компьютерной эпохи. Арифметические операции с вещественными числами довольно долго реализовывались на программном уровне и выполнялись относительно медленно по сравнению с целочисленной арифметикой.

К середине 1980-х годов необходимость в стандартизации стала очевидной (разные компьютеры могли получать несколько отличающиеся результаты для одного вычисления), поэтому Институт инженеров по электротехнике и электронике (*IEEE*, Institute for Electrical and Electronic Engineers) разработал стандарт *IEEE 754* — этот стандарт продолжает дорабатываться до сих пор. Стандарт весьма обширный, и знать его во всех подробностях нет необходимости, но основные идеи будут кратко описаны ниже. Для пояснений будет использоваться 16-разрядная версия, называемая *двоичным форматом с плавающей точкой половинной точности IEEE 754*, сокращенно *half precision* или `binary16`. Основные принципы относятся также и к 32- и 64-разрядным версиям, используемым в Java.

Формат с плавающей точкой



Строение числа с плавающей точкой

В компьютерных системах используется представление вещественных чисел, называемое *форматом с плавающей точкой*. Оно очень похоже на так называемую научную (экспоненциальную) форму записи, если не считать того, что все данные представлены в двоичной системе. Экспоненциальная форма записи — это числа вида $+6{,}0221413 \times 10^{23}$, состоящие из *знака*, *мантиссы* и *порядка* (*показателя степени*). Обычно число выражается таким образом, чтобы целая часть мантиссы состояла из одной цифры (не равной 0); это называется *условием нормализации*. В формате с плавающей точкой число состоит из тех же трех элементов.

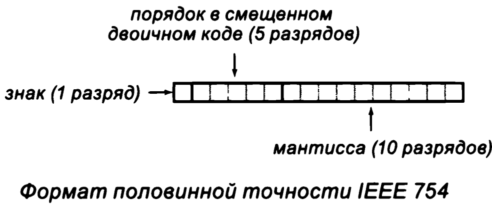
Знак

Первый бит числа с плавающей точкой содержит *знак*. Ничего особенного в нем нет: знаковый разряд равен 0 для положительных чисел (и нуля) или 1 для отрицательных чисел. Таким образом, проверка числа на положительность или отрицательность реализуется просто.

Порядок

В следующих t разрядах числа с плавающей точкой хранится его *порядок*. Количество разрядов, используемых форматами `binary16`, `binary32` и `binary64`, равно 5, 8 и 11 соответственно. Порядок числа с плавающей точкой выражается не в до-

полнительном коде, а в *смещенном двоичном коде*, в котором вычисляется значение $R = 2^{t-1} - 1$ (15, 127 и 1023 для t = 5, 8 и 11 соответственно), и любое десятичное число x в диапазоне от $-R$ до $R + 1$ представляется двоичным представлением $x + R$ (иногда такое «смещенное» значение называют не порядком, а *характеристикой*). В таблице справа приведены 5-разрядные представления в смещенном двоичном коде чисел из диапазона от −15 до +16. Значения `0000` и `1111` используются, согласно стандарту, для других целей.



Формат половинной точности IEEE 754

десятичная запись	двоичная запись
−15	00000
−14	00001
−13	00010
−12	00011
−11	00100
−10	00101
−9	00110
−8	00111
−7	01000
−6	01001
−5	01010
−4	01011
−3	01100
−2	01101
−1	01110
0	01111
1	10000
2	10001
3	10010
4	10011
5	10100
6	10101
7	10110
8	10111
9	11000
10	11001
11	11010
12	11011
13	11100
14	11101
15	11110
16	11111

5-разрядные целые числа (в смещенном двоичном коде)

Дробная часть

Остальные разряды числа с плавающей точкой предназначены для хранения мантиссы: 10, 23 и 53 разрядов для форматов `binary16`, `binary32` и `binary64` соответственно. Условие нормализации требует, чтобы цифра в целой части мантиссы всегда была равна 1, поэтому эта цифра в представление не включается!

С этими правилами процесс *декодирования* числа, записанного в формате IEEE 754, проходит тривиально, как показывает иллюстрация на следующей странице. Согласно стандарту, первый (старший) разряд заданной 16-разрядной величины содержит знак, в следующих 5 разрядах хранится порядок в смещенном двоичном коде (-6_{10}), а следующие 10 разрядов содержат дробную часть, определяющую мантиссу 1,1012. Процесс кодирования числа проходит сложнее из-за необходимости нормализации и расширения двоичного преобразования для дробей. И снова первым будет знаковый разряд, в следующих 5 разрядах хранится порядок, а в следующих 10 разрядах — дробная часть мантиссы. Выполнение этих операций станет непростым упражнением по программированию даже в таком высокоуровневом языке, как Java (см. упражнение 6.1.25, но сначала

прочитайте о работе с отдельными битами в следующем подразделе). Вы поймете, почему числа с плавающей точкой не поддерживались ранними компьютерами и почему на выработку стандарта потребовалось столько времени.

Преобразование IEEE 754 в десятичную запись

1 01001 1010000000

$$-\ 2^{9-15} \times 1.101_2 = -\ 2^{-6}\ (1 + 2^{-1} + 2^{-3}) = -0.0253906250_{10}$$

Преобразование десятичной записи в IEEE 754

$$100.25_{10} = 2^{6}\ (1 + 2^{-1} + 2^{-4} + 2^{-8}) = +\ 2^{21-15} \times 1.100100012$$

0 10101 1001000100

Примеры преобразования между десятичной записью и IEEE 754

Типы данных Java `Float` и `Double` содержат метод `floatToIntBits()`, который может использоваться для проверки кодировки с плавающей точкой. Например, вызов

```
Convert.toString(Float.floatToIntBits(100.25), 2, 16)
```

выводит результат `0101011001000100`, что соответствует второму из приведенных примеров.

Арифметические операции

Выполнение арифметических операций с числами с плавающей точкой также может стать интересным упражнением для программиста. Например, умножение двух чисел с плавающей точкой состоит из следующих шагов.

- Выполнение операции XOR со знаками.
- Суммирование порядков.
- Умножение дробных частей.
- Нормализация результата.

Если вас заинтересует эта тема, вы сможете изучить подробности этого процесса и соответствующих процессов сложения и умножения в упражнении 6.1.25. Сложение в действительности немного сложнее умножения, потому что оно должно начинаться с «денормализации» для приведения в соответствие порядков обоих слагаемых.

Вычисления с плавающей точкой часто усложняются тем, что в них почти всегда участвуют приближенные значения вещественных чисел, а ошибки приближения в длинных последовательностях вычислений могут накапливаться. Так как

у 64-разрядного формата (используемого в типе данных Java `double`) разрядность дробной части более чем вдвое больше, чем у 32-разрядного формата, многие программисты предпочитают использовать тип `double` для уменьшения ошибок приближенного представления; именно так мы и поступаем в этой книге[1].

Операции с битами в Java

Как видно из описания формата с плавающей точкой, кодирование информации в двоичном виде иногда оказывается непростым делом. Рассмотрим средства языка Java, которые упрощают разработку программ, реализующих кодирование и декодирование данных. Это становится возможным благодаря тому, что в Java целые значения представляются в дополнительном коде и значения типов `short`, `int` и `long` явно определены как 16-, 32- и 64-разрядные двоичные числа в дополнительном коде. Не все языки идут по этому пути; некоторые из них требуют, чтобы подобный код был написан на языке более низкого уровня, определяют специальные типы данных для последовательностей битов и/или требуют сложных высокозатратных преобразований. Мы ограничимся 32-разрядными значениями `int`, но все операции также будут работать для значений типов `short` и `long`.

Двоичные и шестнадцатеричные литералы

В языке Java целочисленные литеральные значения могут записываться в двоичной системе (с префиксом `0b`) или в шестнадцатеричной системе (с префиксом `0x`). Эта запись существенно упрощает код при работе с двоичными значениями. Такие литералы могут использоваться там же, где и десятичные литералы; это всего лишь альтернативный способ записи целочисленного значения. Если переменной типа `int` присваивается шестнадцатеричный литерал, содержащий менее 8 цифр, Java подставляет лидирующие нули. Некоторые примеры приведены в таблице ниже.

двоичный литерал	шестнадцатеричный литерал	более короткая форма
`0b01000000010101000100111101011001`	`0x40544F59`	
`0b11111111111111111111111111111111`	`0x0000000F`	`0xF`
`0b00000000000000000001001000110100`	`0x00001234`	`0x1234`
`0b00000000000000001000101000101011`	`0x00008A2B`	`0x8A2B`

[1] Неточность представления значений и связанные с этим ошибки — принципиальное и неустранимое свойство формата с плавающей точкой. Для чувствительных к этому применений используют арифметику с контролируемой точностью на основе форматов с *фиксированной* точкой (или целочисленных). — *Примеч. науч. ред.*

Сдвиги и битовые операции в Java

Чтобы программы могли работать с отдельными битами в значениях `int`, в языке Java поддерживаются операции, перечисленные в таблице «Битовые операции...». Вы можете вычислить дополнение значений разрядов (0 заменяются 1, а 1 заменяются 0), выполнять поразрядные логические операции (AND, OR и XOR), описываемые в нижней части следующей страницы, над соответствующими парами разрядов и сдвигать значения влево и вправо на заданное количество разрядов. Сдвиг вправо существует в двух разновидностях: *логический сдвиг*, при котором освободившиеся разряды слева заполняются 0, и *арифметический сдвиг*, при котором они заполняются содержимым знакового разряда (так называемое размножение знакового разряда, см. «Вопросы и ответы» в конце раздела). Примеры выполнения операций приведены на с. 862.

значения	32-разрядные целые числа						
типичные литералы	`0b00000000000000000000000000001111 0b1111 0xF 0x1234`						
операции	поразрядное дополнение (инверсия)	поразрядное AND	поразрядное OR	поразрядное XOR	сдвиг влево	сдвиг вправо (логический)	сдвиг вправо (арифметический)
операторы Java	~	&	|	^	<<	>>>	>>

Битовые операции встроенного типа данных int

Сдвиги и маски

Одним из основных применений битовых операций является *маскирование* — выделение одного или нескольких разрядов внутри слова. Обычно маски удобнее определять в виде шестнадцатеричных констант. Например, маска `0x80000000` может использоваться для выделения крайнего левого разряда в 32-разрядном слове, маска `0x000000FF` — для выделения крайних правых 8 разрядов, а маска `0x007FFFFF` — для выделения крайних правых 23 разрядов. Если же пойти дальше, сдвиги и маски часто применяются для выделения целочисленных значений, представляемых смежными группами битов.

- Инструкция сдвига вправо используется для смещения нужных разрядов к правому краю слова.
- Чтобы выделить k разрядов, создайте литерал — маску, все разряды которой равны 0, за исключением k правых, которые равны 1.
- Используйте поразрядную конъюнкцию (операцию AND) для отделения битов. В нулевых позициях маски результат будет содержать также 0, а в единичных позициях маски результат заполняется интересующими вас значениями.

дополнение (инверсия)

```
~ 01010001110101110000000000001111
  --------------------------------
  10101110001010001111111111110000
```

операция поразрядного AND

```
  01010001110101110000000000001111
& 00110001011011100011000101101110
  --------------------------------
  00010001010001100000000000001110
```

операция поразрядного OR

```
  01010001110101110000000000001111
| 00110001011011100011000101101110
  --------------------------------
  01110001111111110011000101101111
```

операция поразрядного XOR

```
  01010001110101110000000000001111
^ 00110001011011100011000101101110
  01100000101110010011000101100001
```

сдвиг влево на 6 разрядов

```
  01010001110101110000000000001111
<<00000000000000000000000000000110
  --------------------------------
  01110101110000000000001111000000
```

сдвиг вправо на 3 разряда

```
  01010001110101110000000000001111
>>00000000000000000000000000000011
  --------------------------------
  00001010001110101110000000000001
```

Сдвиги и битовые операции (32 разряда)

Эта последовательность операций позволяет использовать результат так же, как любое другое значение `int`, — часто это именно то, что вам нужно[1]. Далее в этой главе мы покажем, как использовать сдвиги и маски для выделения шестнадцатеричных цифр, как в примере внизу справа.

x	y	AND(x,y)	OR(x,y)	XOR(x,y)
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	1		0

Таблица истинности для поразрядных функций

выражение	значение
`0x00008A2B >> 8`	`0x0000008A`
`(0x00008A2B >> 8) & 0xF`	`0x0000000A`

Пример использования сдвига и маски

[1] Для более удобной и компактной записи подобных операций в некоторых языках (например, в C) предусмотрены *битовые поля*. — *Примеч. науч. ред.*

Листинг 6.1.2. Извлечение компонентов числа в формате с плавающей точкой

```
public class ExtractFloat
{
   public static void main(String[] args)
   {
      float x = Float.parseFloat(args[0]);
      int t = Float.floatToIntBits(x);

      if ((t >> 31) & 1 == 1) StdOut.println("Sign:     -");
      else                    StdOut.println("Sign:     +");

      int exponent = ((t >> 23) & 0xFF) - 127;
      StdOut.println("Exponent: " + exponent);

      double fraction =  1.0 * (t & 0x007FFFFF) / (1 << 23);
      double mantissa =  1.0 + fraction;
      StdOut.println("Mantissa: " + mantissa);

      StdOut.println(mantissa * (1 << exponent));
   }
}
```

Эта программа демонстрирует битовые операции в языке Java: она извлекает поля знака, порядка и дробной части из значения с плавающей точкой, введенного как аргумент командной строки, после чего использует порядок и дробную часть для вычисления абсолютного значения числа.

```
% java ExtractFloat -100.25
    Sign: -
Exponent: 6
Mantissa: 1.56640625
100.25
```

Практический пример использования битовых операций приведен в листинге 6.1.2: программа демонстрирует применение сдвига и маски для извлечения знака, порядка и дробной части из числа с плавающей точкой. Многие пользователи компьютеров работают, не задумываясь о представлении чисел на таком низком уровне (в самом деле, до сих пор в этой книге мы обходились без него), но, как вы увидите, операции над отдельными битами играют важную роль в самых разнообразных приложениях.

Символы

Чтобы работать с текстом, необходимо определить двоичную кодировку для символов. Основной механизм прост: таблица определяет соответствие между символами и n-разрядными двоичными целыми числами без знака. В 6 разрядах можно закодировать 64 различнных символа, в 7 разрядах — 128, в 8 разрядах — 256 и т. д. Как и в случае с числами с плавающей точкой, по мере широкого распространения

компьютеров появилось много разных схем представления, и в разных ситуациях продолжают использоваться разные кодировки.

ASCII

Американский стандарт *ASCII* (American Standard Code for Information Interchange) был разработан как стандарт в 1960 году и с тех пор получил широкое распространение. Это 7-битовый код, хотя в современных компьютерах он чаще всего используется в 8-битовых байтах (начальный бит игнорируется)[1].

Стандарт ASCII разрабатывался прежде всего для обмена данными с телетайпами, предназначенными для отправки и приема текста. По этой причине многие кодируемые символы служили управляющими для таких устройств. Некоторые управляющие символы использовались коммуникационными протоколами (например, `ACK` означает «подтверждение» (acknowledge); другие управляли печатью (например, символ `BS` означает «забой» (backspace), а `CR` — «возврат каретки» (carriage return)).

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1_	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2_	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3_	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4_	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5_	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6_	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7_	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Таблица преобразования шестнадцатеричных кодов в ASCII

Таблица вверху описывает преобразование 8-разрядных двоичных чисел (или эквивалентных им двузначных шестнадцатеричных чисел) в символы и обратно в соответствии с кодировкой ASCII. Первая шестнадцатеричная цифра определяет строку, а вторая — столбец; на пересечении этой строки и столбца находится кодируемый символ. Например, числом `31` кодируется цифра `1`, числом `4A` — буква `J` и т. д. Таблица относится к 7-битовой кодировке ASCII, поэтому первая шестнадцатеричная цифра должна быть не более `7`. Шестнадцатеричные числа, начинающиеся с `0` и `1` (а также числа `20` и `7F`), соответствуют непечатным управляющим символам, таким как «возврат каретки» CR, который теперь называется

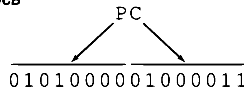
[1] В «расширенной» таблице ASCII кодам с `80` до `FE` соответствуют символы национальных алфавитов (например, русского или греческого), псевдографики и некоторые другие (конкретный набор символов зависит от *кодовой страницы*). Не все приложения корректно обрабатывают такие символы. — *Примеч. науч. ред.*

«символом новой строки» (другие подобные символы в современных программах используются редко)[1].

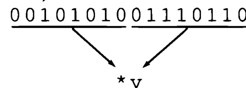
Юникод

В современном мире XXI века пользователь никак не обойдется сотней (или около того) ASCII-символов из XX века, поэтому появился новый стандарт, называемый *Юникодом.* Используя 16 бит для большинства символов (и 24 или 32 бита для отдельных символов), Юникод позволяет поддерживать десятки тысяч символов из широкого спектра мировых языков. Кодировка UTF-8 (преобразование символов в последовательности 8-битовых байтов и наоборот, в которой большинство символов представляется двумя байтами) быстро завоевывает популярность как новый стандарт. Сложные, но всеобъемлющие правила кодирования реализуются в большинстве современных систем (таких, как Java), так что программистам обычно не приходится беспокоиться о технических подробностях. Стандарт ASCII продолжает существовать и в Юникоде: каждый ASCII-символ представляется одним байтом, так что ASCII-файлы являются частными случаями файлов UTF-8 (и обладают обратной совместимостью с ними).

Преобразование ASCII (два символа) в двоичную запись



Преобразование двоичной записи в ASCII (два символа)



Примеры преобразования ASCII-символов в двоичную запись

Обычно мы стараемся упаковать в компьютерное слово как можно больше информации, поэтому два ASCII-символа можно закодировать 16 битами (см. пример выше), четыре символа — 32 битами, восемь символов — 64 битами и т. д. В высокоуровневых языках (таких, как Java) эти подробности, а также кодирование и декодирование UTF-8 реализуются (и инкапсулируются) типом данных `String`, который часто использовался в книге. Тем не менее программистам Java важно понимать основные особенности используемого представления, так как они, безусловно, могут повлиять на требования программ к ресурсам. Например, многие программисты обнаружили, что затраты памяти в их программах неожиданно удвоились, когда в 2000-е годы язык Java перешел с ASCII на Юникод и для ко-

[1] В различных операционных системах для обозначения конца строки приняты разные обозначения. Так, в ОС семейства Unix это обычно `CR` (как и указано авторами), но в Microsoft DOS и Windows — пара символов `LF` и `CR`. — *Примеч. науч. ред.*

дирования каждого символа ASCII стал использоваться 16-разрядный тип `char`. Опытные программисты знают, как при необходимости упаковать два ASCII-символа в значение `char`, чтобы сэкономить память.

Выводы

В общем случае ваши программы должны правильно работать независимо от используемого представления данных. Многие языки программирования полностью поддерживают эту точку зрения. К сожалению, этот подход может вступить в прямое противоречие со стремлением в полной мере использовать возможности компьютера — в основном за счет использования оборудования так, как планировали его разработчики. Примитивные типы Java создавались для поддержки этой точки зрения. Например, если на компьютере аппаратно поддерживается сложение или умножение 64-разрядных целых чисел, то нам хотелось бы, чтобы каждая операция сложения или умножения сводилась к одной машинной инструкции — в этом случае программа будет выполняться с максимальной быстротой. По этой причине программист должен попытаться найти соответствие между типами данных с операциями, критичными по времени, и примитивными типами, реализованными аппаратно. Возможно, для этого потребуется более глубокое понимание системы и ее программного обеспечения, но, по крайней мере, стремление оптимизировать производительность — достойная цель.

Вы писали программы, которые выполняли вычисления с различными типами данных. В этом разделе мы хотели показать, что, поскольку любая последовательность битов может интерпретироваться многими разными способами, *смысл заданной последовательности битов в компьютере зависит от контекста*. Ваши программы могут интерпретировать биты так, как вы хотите. По самим по себе битам невозможно определить, какой тип данных они представляют, — и как вы вскоре увидите, даже понять, представляют ли они какие-нибудь данные.

двоичная запись	шестнадцатеричная запись	без знака	дополнительный код	число с плавающей точкой (binary16)	ASCII-символы
0001001000110100	1234	4660	4660	0,0007572174072265625	DC2 4
1111111111111111	FFFF	65535	−1	−131008,0	DEL DEL
1111101011001110	FACE	64206	−1330	−55744,0	e N
0101011001000100	5644	22052	22 052	100,25	V D
1000000000000001	8001	32769	−32 767	−0,0000305473804473876953125	NUL SOH
0101000001000011	5043	20547	20 547	34,09375	P C
0001110010101011	1CAB	7339	7339	0,004558563232421875	FS +

Шесть способов интерпретации различных 16-разрядных значений

Чтобы подчеркнуть эту мысль, в таблице на с. 867приводятся несколько различных 16-разрядных значений вместе с интерпретацией этих значений как двоичные целые числа, шестнадцатеричные целые числа, беззнаковые целые числа, целые числа в дополнительном коде, числа с плавающей точкой в формате `binary16` и пары ASCII-символов. Это всего лишь несколько примеров бесчисленных способов представления информации в компьютере.

Вопросы и ответы

В. Как узнать размер слова на моем компьютере?

О. Узнайте название процессора, а затем обратитесь к спецификации этого процессора. Скорее всего, на вашем компьютере используется 64-разрядный процессор. А если нет, возможно, стоит подумать о покупке нового компьютера.

В. Почему Java использует для представления значений `int` 32 разряда, если у большинства компьютеров 64-разрядные слова?

О. Это решение было давно принято создателями языка. У языка Java есть одна отличительная особенность: спецификация полностью определяет представление типа `int`. Преимущество такого подхода заключается в том, что старые программы Java будут работать на новых компьютерах с большей вероятностью, чем программы, написанные на языках с различающимися для разных машин (*машинно-зависимыми*) представлениями. Впрочем, есть и недостаток: 32 разряда часто оказывается недостаточно. Например, в 2014 году компании Google пришлось изменить 32-разрядное представление своего счетчика просмотров, когда стало ясно, что сверхпопулярное видео *Gangham Style* будет просмотрено более 2 147 483 647 раз. В языке Java можно переключиться на тип `long`.

В. Но ведь, по идее, такие проблемы должны решаться системой?

О. Некоторые языки (например, Python) не ограничивают размер целых чисел, чтобы система могла при необходимости использовать для целочисленных значений несколько машинных слов. В Java в таких случаях можно воспользоваться классом `BigInteger`.

В. Что делает класс `BigInteger`?

О. Он позволяет выполнять вычисления с целыми числами, не беспокоясь о переполнении. Например, если импортировать `java.math.BigInteger`, то код

```
BigInteger x = new BigInteger("2");
StdOut.println(x.pow(100));
```

выводит `1267650600228229401496703205376` — значение 2^{100}. Значение `BigInteger` можно рассматривать как строку (на самом деле внутреннее представление более эффективно), а класс предоставляет методы для стандартных арифметических

и многих других операций. Например, этот класс применяется в криптографии: арифметические действия с числами, состоящими из сотен цифр, играют важную роль в некоторых криптографических системах. Реализация использует столько цифр, сколько потребуется, так что с переполнением проблем не будет. Конечно, такие операции обходятся намного дороже, чем встроенные операции типов `long` или `int`, поэтому программисты Java не используют `BigInteger` для целых чисел в диапазоне, поддерживаемом `long` или `int`.

В. Почему именно шестнадцатеричная система? Разве нет других оснований, которые справятся с этой задачей?

О. Восьмеричная система счисления (с основанием 8) широко применялась в ранних компьютерных системах с 12-, 24- и 36-разрядными словами, потому что содержимое слова могло выражаться 4, 8 или 12 восьмеричными цифрами соответственно. Преимущество восьмеричной записи перед шестнадцатеричной в таких системах заключалось в том, что в ней использовались только знакомые десятичные цифры от 0 до 7, так что примитивные устройства ввода/вывода (такие, как цифровые клавиатуры) могли использоваться для ввода как десятичных, так и восьмеричных чисел. Однако восьмеричная система неудобна для 32- и 64-разрядных слов, потому что эти размеры не делятся на 3 нацело. (На 5 и 6 они тоже не делятся, так что переключение на систему счисления с большим основанием тоже маловероятно.)

В. Как защититься от переполнения?

О. Это не так просто, потому что для каждой операции нужна отдельная проверка. Например, если `x` и `y` — положительные числа и нужно вычислить `x+y`, можно проверить, что `x < Integer.MAX_VALUE - y`.

Другой способ основан на «повышающем преобразовании» к типу с более широким диапазоном представимых значений. Например, если вы выполняете вычисления со значениями `int`, их можно преобразовать в значения `long`, а затем преобразовать результат обратно в `int` (если он не слишком велик для этого).

В Java 8 можно воспользоваться методом `Math.addExact()`, который выдает исключение при переполнении.

В. Как компьютер может обнаружить переполнение при сложении значений в дополнительном коде?

О. Правило простое, хотя доказать его несколько сложнее: нужно проверить значения переноса *в* старший (самый

выходной перенос отличен от входного

```
 ^
1000000000000000
 1111111111111000        -8
 1000000000000100    -32764
 ----------------
 1111111111111100        -4 x
```

выходной перенос отличен от входного

```
 ^
0111111111111000
 0111111111111000     32760
 0000000000001000     +   8
 ----------------
 1000000000000000    -32768 x
```

Переполнение (16-разрядный дополнительный код)

левый) разряд и *из* него. Если эти значения различны, значит, произошло переполнение (см. примеры).

В. Для чего нужен арифметический сдвиг?

О. Для целых чисел со знаком в дополнительном коде арифметический сдвиг вправо на 1 эквивалентен целочисленному делению на 2. Например, значение `(-16) >> 3` равно `-2` (см. примеры). Чтобы проверить, правильно ли вы понимаете этот оператор, самостоятельно вычислите значения `(-3) >> 1` и `(-1) >> 1`[1]. Это называется «расширением (распространением, размножением) знака» в отличие от «дополнения нулями» при логическом сдвиге.

положительное число

```
     x:  0000000000010000   16
x >> 3:  0000000000000010    2
```

заполнение 0

отрицательное число

```
     x:  1111111111110000  -16
x >> 3:  1111111111111110   -2
```

заполнение 1

Арифметический сдвиг (16-разрядный дополнительный код)

В. Чему равен результат `x >> y`, если значение `y` отрицательное или превышает 31?

О. Java использует только пять младших разрядов второго операнда. Это соответствует поведению физического оборудования на типичном компьютере.

В. Я так и не понял примеры из подраздела «Вопросы и ответы» раздела 1.2, где утверждалось, что условие `(0.1+0.1==0.2)` истинно, а условие `(0.1+0.1+0.1==0.3)` ложно. Вы можете объяснить, как это происходит?

О. Литерал вида `0.1` или `0.3` в исходном коде Java преобразуется к ближайшему 64-разрядному числу IEEE 754 (если есть два одинаково подходящих варианта — тому, у которого младший бит равен 0), то есть значению Java `double`. Вот как выглядят эти значения для литералов `0.1`, `0.2` и `0.3`:

литерал	ближайшее 64-разрядное число IEEE 754
0.1	`0.1000000000000000055511151231257827021181583404541015625`
0.2	`0.2000000000000000111022302462515654042363166809082031250`
0.3	`0.2999999999999999888977697537484345957636833190917968750`

Как видите, результат `0.1 + 0.1` равен `0.2`, но результат `0.1 + 0.1 + 0.1` больше `0.3`. Собственно, ситуация принципиально не отличается от того, что для целочислен-

[1] Соответственно логический сдвиг вправо эквивалентен делению на 2 беззнакового целого числа. В ряде языков программирования способ сдвига (арифметический или логический) выбирается автоматически в зависимости от типа операнда. Сдвиг влево всегда выполняется одинаково; он может исказить знак операнда. — *Примеч. науч. ред.*

ного сложения результат `2/5 + 2/5` равен `4/5` (оба значения равны 0), но результат `2/5 + 2/5 + 2/5` не равен `6/5`.

В. Вызов `System.out.println(0.1)` выводит `0.1`, а не значение из таблицы на предыдущей странице. Почему?

О. Программистам такая точность обычно не нужна, поэтому `println()` усекает вывод для удобства чтения. Дробная часть должна содержать хотя бы одну цифру и кроме нее столько цифр (и не более!), сколько необходимо для того, чтобы значение аргумента однозначно отличалось от соседних значений типа `double`. Для более точного управления форматом вывода можно использовать `printf()`, а для повышения точности — класс `BigDecimal`. Например, если импортировать `java.math.BigDecimal`, фрагмент

```
double x = 0.1;
StdOut.println(new BigDecimal(x));
```

выводит результат `0.1000000000000000055511151231257827021181583404541015625`.

Упражнения

6.1.1. Переведите десятичное число `92` в двоичную систему счисления.

Решение. `1011100`.

6.1.2. Переведите восьмеричное число `31415` в двоичную систему счисления.

Решение. `011001100001101`.

6.1.3. Переведите восьмеричное число `314159` в десятичную систему счисления.

Решение. Это не восьмеричное число! Вы можете провести вычисления — и даже воспользоваться `Convert`. Вы получите результат `104561`, но `9` просто не является допустимой восьмеричной цифрой. Проверка числа на действительность включена в версию `Convert` на сайте книги. Преподаватели нередко включают подобные каверзы в контрольные работы, так что будьте внимательны!

6.1.4. Переведите шестнадцатеричное число `BB23A` в восьмеричную систему счисления.

Решение. Сначала переведите число в двоичную запись `1011 1011 0010 0011 1010`, затем разбейте биты на тройки `10 111 011 001 000 111 010` и преобразуйте в восьмеричную запись по группам `2731072`.

6.1.5. Сложите два шестнадцатеричных числа `23AC` и `4B80` и приведите результат в шестнадцатеричной форме. *Подсказка*: выполните сложение прямо в шестнадцатеричной форме вместо того, чтобы переводить числа в десятичную запись, складывать, а потом переводить их обратно.

6.1.6. Предположим, m и n — положительные целые числа. Сколько единичных битов (то есь битов, равных 1) содержит двоичное представление 2^{m+n}?

6.1.7. Как выглядит единственное десятичное число, которое в шестнадцатеричной записи состоит из тех же цифр, переставленных в обратном порядке?

Решение. Число `53` — в шестнадцатеричной записи `35`.

6.1.8. Докажите, что преобразование шестнадцатеричного числа по одной цифре в двоичную форму (и наоборот) всегда дает правильный результат.

6.1.9. Протокол IPv4, разработанный в 1970-е годы, определяет правила взаимодействия между компьютерами в Интернете. Каждому компьютеру в Сети должен быть назначен уникальный интернет-адрес. В протоколе IPv4 используются 32-разрядные адреса. Сколько компьютеров может быть подключено к Интернету? Хватит ли этого, чтобы каждый мобильный телефон и каждый тостер обладал собственным адресом?[1]

6.1.10. IPv6 — интернет-протокол, в котором каждому компьютеру присваивается 128-разрядный адрес. Сколько компьютеров может быть подключено к Интернету в случае принятия этого стандарта? Достаточно ли этого?

Решение. 2^{128}. Пожалуй, в краткосрочной перспективе этого хватит — 5000 адресов на квадратный микрометр поверхности Земли!

6.1.11. Заполните значения выражений в следующей таблице:

выражение	**~0xFF**	**0x3 & 0x5**	**0x3 | 0x5**	**0x3 ^ 0x5**	**0x1234 << 8**
значение					

6.1.12. Разработайте реализацию метода `toInt()`, описанного в тексте, для преобразования символа в диапазоне 0–9 или A–Z в значение типа `int` в диапазоне от 0 до 35.

Решение.

```
public static int toInt(char c)
{
    if ((c >= '0') && (c <= '9')) return c - '0';
    return c - 'A' + 10;
}
```

6.1.13. Разработайте реализацию метода `toChar()`, описанного в тексте, для преобразования значения типа `int` в диапазоне от 0 до 35 в символ в диапазоне 0–9 или A–Z.

[1] Адрес IP представляет собой структуру, поля которой интерпретируются по достаточно сложным правилам, поэтому не все значения могут быть реально использованы. С другой стороны, в стандарте IP предусмотрены в том числе и средства преодоления нехватки адресов. — *Примеч. науч. ред.*

Решение.

```
public static char toChar(int i)
{
    if (i < 10) return (char) ('0' + i);
    return (char) ('A' + i - 10);
}
```

6.1.14. Измените программу `Convert` (и ответы из двух предыдущих упражнений) так, чтобы они использовали тип `long`, проверяли возможное переполнение, а также проверяли, что цифры входной строки лежат в диапазоне, определяемом основанием системы счисления.

Решение. см. файл Convert.java на сайте книги.

6.1.15. Добавьте в программу `Convert` версию метода `toString()`, которая получает третий аргумент — длину строки. Если заданная длина меньше необходимой, метод возвращает только правые цифры; если больше — дополняет результат лидирующими нулями. Например, вызов `toString(64206, 16, 3)` должен возвращать `"ACE"`, а вызов `toString(15, 16, 4)` должен возвращать `"000F"`. *Подсказка*: начните с вызова версии с двумя аргументами.

6.1.16. Напишите программу Java `TwosComplement`, которая получает как аргумент командной строки значение `i` типа `int` и разрядность слова `w` и выводит `w`-разрядное двоичное представление `i` в дополнительном коде и его же шестнадцатеричное представление. Предполагается, что значение `w` кратно 4. Программа должна работать примерно так:

```
% java TwosComplement -1 16
1111111111111111 FFFF
% java TwosComplement 45 8
00101101 2D
% java TwosComplement -1024 32
11111111111111111111110000000000 FFFFFC00
```

6.1.17. Модифицируя программу `ExtractFloat`, создайте на ее основе программу `ExtractDouble`, которая делает то же самое со значениями `double`.

6.1.18. Напишите программу Java `EncodeDouble`, которая получает значение `double` из командной строки и кодирует его в формате числа с плавающей точкой по стандарту IEEE 754 *binary32*.

6.1.19. Заполните пропуски в таблице.

двоичная запись	число с плавающей точкой
0010001000110100	
1000000000000000	
	7,09375
	1024

6.1.20. Заполните пропуски в таблице.

двоичная запись	шестнадцатеричная запись	беззнаковое целое	дополнительный код	ASCII-символы
1001000110100111				
	9201			
		1000		
			−131	
				??

Упражнения повышенной сложности

6.1.21. *Строковое и числовое представление IP-адреса.* IP-адрес (IPV4) состоит из числовых полей w, x, y и z и обычно записывается в виде строки $w.x.y.z$ — так называемая десятичная (или «точечная») запись. Соответствующее *числовое значение* IP-адреса вычисляется по формуле $16777216w + 65536x + 256y + z$. Для заданного числового значения n соответствующие поля IP-адреса вычисляются по формулам `w = (n / 16777216) mod 256`, `x = (n / 65536) mod 256`, `y = (n / 256) mod 256`, `z = n mod 256`. Напишите функцию, которая получает числовое значение IP-адреса и возвращает десятичную запись IP-адреса в формате `String`, и другую функцию, которая получает десятичную запись IP-адреса и возвращает соответствующее числовое значение IP-адреса в формате `int`. Например, если задано значение `3401190660`, первая функция должна вернуть строку `202.186.13.4`.

6.1.22. *IP-адрес.* Напишите программу, которая получает как аргумент командной строки строку из 32 символов, представляющую IP адрес в двоичном формате, и выводит соответствующий IP-адрес в десятичной записи, с разделением групп точками. Например, двоичный IP-адрес `01010000000100000000000000000000` должен преобразовываться в `80.16.0.1`.

6.1.23. *MAC-адрес.* Напишите функции для преобразования MAC-адресов в 48-разрядные значения `long` и обратно.

6.1.24. *Кодирование Base64.* Base64 — популярный метод кодирования данных для передачи двоичных данных в сети Интернет. Он преобразует произвольные данные в ASCII-текст, который может беспроблемно передаваться между системами. Напишите программу для чтения произвольных двоичных файлов и кодирования их в Base64.

6.1.25. *Вычисления в формате с плавающей точкой.* Напишите класс `FloatingPoint` с тремя переменными экземпляров `sign`, `exponent` и `fraction`. Реализуйте операции сложения и умножения, добавьте методы `toString()` и `parseFloat()`. Обеспечьте поддержку 16-, 32- и 64-разрядных форматов.

6.1.26. *Кодирование ДНК.* Разработайте класс `DNA` для эффективного представления строк, состоящих исключительно из символов `A`, `T`, `C` и `G`. Добавьте конструктор для преобразования строки символов во внутреннее представление, метод `toString()` для преобразования внутреннего представления в строку, метод `charAt()` для получения символа с заданным индексом (из внутреннего представления) и метод `indexOf()`, который получает в аргументе подстроку в формате `String` и возвращает первое вхождение этой подстроки в строке, хранящейся во внутреннем представлении. Во внутреннем представлении используйте массив значений `int`, в каждом элементе которого упакованы 16 символов (по два бита на символ).

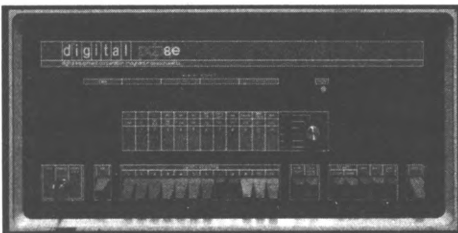
6.2. Машина TOY

Чтобы вы лучше поняли природу вычислений на вашем компьютере, мы рассмотрим в этом разделе TOY — воображаемую машину, спроектированную специально для этой книги, которая очень похожа на компьютеры, получившие широкое распространение в 1970-е годы. Мы изучаем сейчас эту машину, потому что ей свойственно все то же, что и современным микропроцессорам, которыми оснащены ваш смартфон, ваш комьютер, множество других компьютеров и бесчисленное множество разнообразных вычислительных устройств, разработанных за прошедшие годы. На иллюстрации далее изображены PDP-8 — реальный компьютер 1970-х годов — и наш воображаемый компьютер TOY.

TOY демонстрирует, что простые вычислительные модели могут выполнять полезные и нетривиальные вычисления, а также служит ориентиром, который поможет вам понять основные особенности вашего собственного компьютера. Один из удивительных фактов эволюции компьютерных технологий за несколько последних десятилетий состоит в том, что все компьютеры используют одинаковую базовую архитектуру — подход, который был повсеместно принят почти сразу же после того, как его впервые сформулировал Джон фон Нейман в 1945 году.

Начнем с описания основных компонентов машины TOY. Таких компонентов немного, и понять предназначение каждого из них несложно. Из подобных компонентов строятся все компьютеры.

PDP-8, 1970-е годы



TOY, вне времени

Два компьютера: настоящий и воображаемый

Затем вы узнаете, как использовать и программировать машину TOY. Мы начнем с основных способов представления информации, описанных в предыдущем разделе, и перейдем к операциям с этой информацией. Иначе говоря, мы работаем с типами данных, которые реализует аппаратное обеспечение машины TOY: множествами значений и операций над этими значениями. Работа на таком уровне называется *программированием на машинном языке*. Изучение программирования на уровне машинного языка поможет вам лучше понять не только связь между программами Java и вашим компьютером, но и саму природу вычислений. На самом деле программирование на машинном языке продолжает использоваться в приложениях, критичных по времени (работе с видео, работе со звуком и обработке научных данных). Как вы увидите, научиться программировать на машинном языке не так уж сложно.

В главе 7 мы расскажем, как построить такой компьютер на физическом уровне. Этот последний шаг, поднимающий завесу тайны над вашим компьютером, поможет вам лучше понять связь между программами Java и реальным миром.

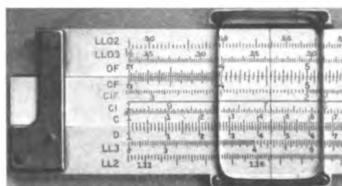
Эта важная абстрактная прослойка присутствует во всех компьютерах. Полное и точное описание того, что может делать процессор, определяет целевой язык для перевода программ, написанных на языках высокого уровня (таких, как Java), и в то же время формирует основу для построения электронных компонентов, реализующих машину.

Краткая историческая справка

Пожалуй, современный мир уже невозможно представить без компьютеров. Чтобы начать с чего-то определенного, представьте 1950-е годы, когда после Второй мировой войны началась техническая революция. В это время уже были автомобили, самолеты, спутники и всевозможные технологические достижения, которые известны нам и сегодня, но у среднего человека — и даже среднего ученого или инженера — не было доступа к компьютеру.

Первые компьютеры создавались для разного рода научных, инженерных и коммерческих приложений. Вторая мировая война продемонстрировала ценность этих вычислений — от баллистических таблиц, построенных Джоном фон Нейманом, до дешифровальной машины, разработанной Аланом Тьюрингом (не говоря уже о вычислениях, проведенных в Лос-Аламосе в ходе разработки атомной бомбы). Только представьте мир, в котором в ходе деятельности банка или проектирования автомобиля не использовались компьютеры.

До 1970-х годов студенты и инженеры в основном применяли для вычислений *логарифмическую линейку* — устройство определенно неэлектронное и нецифровое, которое тем не менее было очень полезным, особенно при вычислении логарифмов и умножении. Еще одним популярным инструментом был справочник *таблиц функций*: например, чтобы вычислить `sin(x)`, вам приходилось искать значение в книге!



Логарифмическая линейка

Компьютеры постепенно становились более доступными. На первых порах они обычно использовались группами специалистов совместно, а работа с ними была довольно неудобной. Тем не менее с первого взгляда становилось ясно, что технологии вычислений далеко превзошли возможности логарифмических линеек и таблиц функций. За очень короткий промежуток времени доступ к большим компьютерам сделал логарифмические линейки и таблицы функций пережитком прошлого. Многие годы люди пользовались калькуляторами, которые базировались на той же технологии, что и компьютеры, но специализировались на инженерных и бухгалтерских расчетах и были более портативными. Для простых вычислений хватало калькуляторов, а для сложных научных и инженерных вычислений разработчики писали компьютерные программы, как и сейчас. Характерно, что основные принципы архитектуры первых устройств, созданных для таких целей, дожили до сегодняшнего дня и продолжают поддерживать работу бесчисленных приложений, преобразивших мир.

Компоненты TOY

Начнем с обзора основных компонентов, которые встречаются практически во всех компьютерах вот уже более полувека, а в машине TOY они сведены к своим первоосновам.

Память

Память — важнейший компонент любого компьютера. В памяти размещаются не только обрабатываемые данные и результаты вычислений, но и любые программы, которые выполняются на этой машине. Память TOY состоит из 256 слов, каждое из которых состоит из 16 бит. По современным стандартам, не много, конечно, но вас удивит, какие разнообразные вычисления может обеспечивать эта машина. И еще одно соображение: эти $256 \times 16 = 4096$ битов могут принимать 2^{4096} возможных значений, так что абсолютное большинство возможных состояний TOY никогда не будет наблюдаться в этой Вселенной.

В шестнадцатеричной системе содержимое слова памяти записывается 4 шестнадцатеричными цифрами. Кроме того, слова считаются пронумерованными от 0 до 255, поэтому каждое слово идентифицируется шестнадцатеричным номером из двух цифр, которое называется *адресом*. Например, можно сказать «значение слова по адресу `1E` равно `0FA2`» или просто «ячейка памяти `1E` содержит значение `0FA2`».

	0_	1_	2_	3_	4_	5_	6_	7_	8_	9_	A_	B_	C_	D_	E_	F_
_0	7A10	8A15	8A2B	7101	7101	7800	8AFF	7101	7101	BB0E	0000	0000	0000	0000	0000	0000
_1	7BEF	8B16	8B2C	75FF	7A00	8CFF	8BFF	A90A	A90A	1EE1	0000	0000	0000	0000	0000	0000
_2	9AFF	1CAB	2CAB	7901	7B01	CC55	7101	140A	180A	900E	0000	0000	0000	0000	0000	0000
_3	9BFF	9C17	CC29	2C59	894C	188C	7900	7B00	C98F	1EE1	0000	0000	0000	0000	0000	0000
_4	7101	0000	DC27	CC3B	C94B	C051	22B9	C97C	AC09	900E	0000	0000	0000	0000	0000	0000
_5	7900	0008	2BBA	1991	9AFF	98FF	C26B	2991	2CBC	1EE1	0000	0000	0000	0000	0000	0000
_6	22B9	0005	C022	1A09	1CAB	0000	1CA9	2441	CC96	EF00	0000	0000	0000	0000	0000	0000
_7	C200	0000	EF00	8B3D	1AB0	0000	8DFF	AC04	1991	0000	0000	0000	0000	0000	0000	0000
_8	1CA9	0000	0000	FF22	1BC0	0000	BD0C	2EBC	1809	0000	0000	0000	0000	0000	0000	0000
_9	AD0C	0000	00C3	2AA1	2991	0000	1991	DE7B	A909	0000	0000	0000	0000	0000	0000	0000
_A	9DFF	0000	0111	CA36	C044	0000	C064	1B0C	DCBE	0000	0000	0000	0000	0000	0000	0000
_B	1441	0000	0000	9B3E	0000	0000	1AA9	C074	1981	0000	0000	0000	0000	0000	0000	0000
_C	C006	0000	0000	0000	000C	FF60	BB0A	EF00	1809	0000	0000	0000	0000	0000	0000	0000
_D	0000	0000	0000	005B	0000	FF70	EF00	0000	A909	0000	0000	0000	0000	0000	0000	0000
_E	0000	0000	0000	0000	0000	9BFF	0000	0000	C083	0000	0000	0000	0000	0000	0000	0000
_F	0000	0000	0000	0000	0000	0000	0000	0000	BE08	0000	0000	0000	0000	0000	0000	0000

Дамп памяти машины TOY (256 16-разрядных слов)

Для экономии места мы часто используем синтаксис массивов и пишем «M[1E] содержит 0FA2». Содержимое памяти TOY может быть задано таблицей из 16 строк вроде приведенной вверху. Первый столбец задает значения адресов от 00 до 0F; второй столбец — значения адресов от 10 до 1F и т. д. Такая таблица называется *дампом памяти*. В данном примере память содержит все примеры, рассматриваемые в этой главе (!).

Инструкции

Еще одна важнейшая особенность машины TOY (и практически любого другого компьютера) заключается в том, что содержимое слова памяти может интерпретироваться и как данные, и как инструкция (команда) в зависимости от контекста. Например, вы уже знаете из предыдущего раздела, что значение 1234 может интерпретироваться как целое число 4660_{10} или вещественное число 0,00302886962890625; в этом разделе вы узнаете, что оно также может представлять машинную команду (инструкцию) для сложения двух чисел. Программист должен сам позаботиться о том, чтобы программы работали с данными как с данными, а с инструкциями — как с инструкциями. Далее будет рассмотрен способ кодирования всех инструкций TOY, чтобы вы могли декодировать произвольное 16-разрядное значение как инструкцию (и закодировать любую инструкцию в виде 16-разрядного значения).

R[0]	0000
R[1]	0001
R[2]	000A
R[3]	0000
R[4]	0000
R[5]	0000
R[6]	0030
R[7]	0000
R[8]	003A
R[9]	0000
R[A]	0A23
R[B]	0B44
R[C]	0C78
R[D]	0000
R[E]	0000
R[F]	0000

Регистры TOY

Регистры

Регистр — компонент машины (точнее, обычно ее *процессора*), который может использоваться для хранения последовательности битов (как и слово основной памяти). Регистры используются для хранения промежуточных результатов в ходе вы-

числений. Считайте, что они играют роль переменных при программировании TOY[1]. Машины TOY содержат 16 регистров с номерами от 0 до F. Как и в случае с памятью, мы используем синтаксис массивов, а для обращений к регистрам используются обозначения от R[0] до R[F]. Так как все регистры, как и ячейки памяти, являются 16-разрядными, содержимое каждого регистра представляется шестнадцатеричным значением из 4 цифр, а содержимое всех регистров может быть представлено таблицей из 16 шестнадцатеричных чисел, каждое из которых состоит из 4 цифр. В таблице на с. 877 (внизу) показано содержимое регистров в ходе типичных вычислений (см. далее). По действующим (для TOY!) правилам R[0] всегда содержит 0000.

Арифметико-логическое устройство

Арифметико-логическое устройство (АЛУ), вычислительное ядро TOY — исполнительный компонент машины, который выполняет все ее вычисления. Как правило, инструкция TOY приказывает («инструктирует») АЛУ вычислить некоторую функцию: взять содержимое двух регистров в качестве аргументов и поместить результат в третий регистр. Например, инструкция TOY 1234 направляет содержимое R[2] и R[3] в АЛУ, суммирует их, а затем направляет результат в R[4]. Позднее в этом разделе вы узнаете, как составлять программы TOY из таких элементарных действий.

Счетчик адреса программы и регистр инструкций

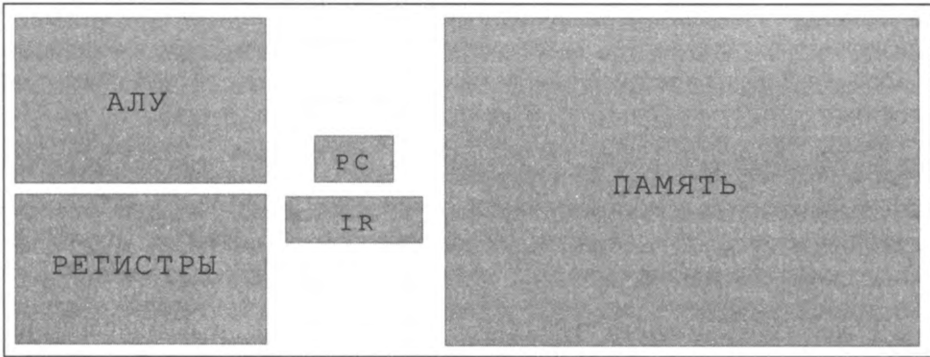
8-разрядный *счетчик адреса программы* (PC, Program Counter) представляет собой внутренний регистр для хранения адреса следующей исполняемой инструкции. 16-разрядный *регистр инструкций* (IR, Instruction Register) содержит текущую исполняемую инструкцию. Эти регистры занимают центральное место в работе машины. Программы не обращаются к IR напрямую, но программисты постоянно знают его содержимое.

На диаграмме на с. 879 изображены все эти основные компоненты. В главе 7 будет рассмотрен процесс создания *электронных схем*, реализующих все эти функции. А оставшаяся часть этой главы будет посвящена взаимодействию этих компонентов и тому, как программист может управлять ими для выполнения необходимых вычислений.

Цикл «выборка-инкремент-исполнение»

Машина TOY выполняет инструкции, многократно повторяя определенную последовательность действий. Сначала она проверяет значение PC и производит выборку (копирует) содержимое адресуемой им ячейки памяти в IR. Затем счетчик

[1] Регистровая память обычно меньше по объему, но быстрее основной. Регистры могут быть (в зависимости от архитектуры машины) включены в общее адресное пространство основной памяти, но при этом, как правило, все равно выделяются регистр-*аккумулятор* АЛУ и регистры *специальных функций*. — *Примеч. науч. ред.*



Компонент машины TOY

адреса программы PC увеличивается на 1. (Например, если счетчик равен `10`, он принимает значение `11`.) Наконец, она интерпретирует 16-разрядное значение в IR как инструкцию и выполняет ее по правилам машины TOY (см. ниже). Каждая инструкция может изменять содержимое регистров, основной памяти и даже самого счетчика. После выполнения инструкции машина повторяет весь цикл «выборка-инкремент-исполнение», используя новое значение счетчика для поиска следующей инструкции. Процесс продолжается бесконечно — или до тех пор, пока не будет выполнена инструкция останова. Как и программа на языке Java, программа на машинном языке может входить в бесконечный цикл. Чтобы вывести машину TOY из бесконечного цикла, программисту придется выполнить сброс или даже отключить питание.

Цикл «выборка-инкремент-исполнение»

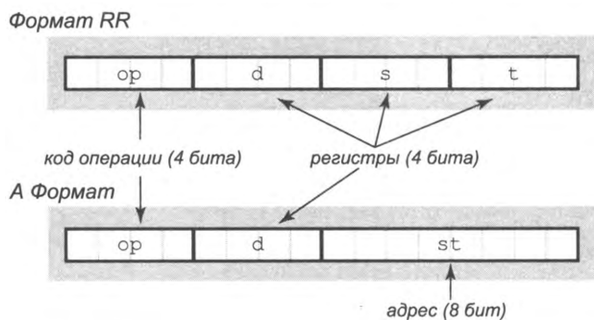
Инструкции

Любое 16-разрядное значение (содержимое произвольного слова памяти) может интерпретироваться как инструкция TOY. Каждая инструкция некоторым образом изменяет состояние машины (значение слова памяти, регистра, счетчика PC). Для описания действия инструкций используется псевдокод, который очень похож на код Java, но работает непосредственно со словами памяти, регистрами и PC.

Структура инструкции

Для кодирования инструкций используется шестнадцатеричная запись: каждая 16-разрядная инструкция записывается в виде 4 шестнадцатеричных цифр. Первая цифра в этой записи определяет *код операции* — признак выполняемой операции. Машина поддерживает 16 инструкций, поэтому одной шестнадцатеричной цифры хватит для идентификации их всех. Вторая цифра инструкции задает *регистр* —

каждая инструкция использует или изменяет значение некоторого регистра. Регистров всего 16, и одной шестнадцатеричной цифры достаточно для идентификации их всех. Большинство инструкций относится к одному из двух форматов, которые определяют интерпретацию третьей и четвертой шестнадцатеричных цифр. В инструкциях формата *RR* («регистр-регистр») каждая из двух оставшихся цифр определяет регистр. В инструкциях формата *A* («адрес») третья и четвертая цифры (совместно) определяют адрес памяти.



Строение команд TOY

Набор инструкций

На следующей странице приведена таблица с описанием всех инструкций TOY. Это полный справочник по программированию TOY, к которому следует обращаться при написании программ TOY. Далее все инструкции будут описаны более подробно.

Останов

Операция с кодом 0 (самая простая инструкция) просто приказывает машине остановиться, то есть прервать цикл «выборка-инкремент-исполнение». В этом состоянии программист может проверить содержимое памяти, чтобы увидеть результаты вычислений. TOY игнорирует три последние шестнадцатеричные цифры инструкции останова, так что любая из комбинаций `0000`, `0123` или `0FFF` является инструкцией останова.

Инструкции арифметических операций

Инструкции с кодами операции `1` и `2` относятся к *арифметическим*, они приказывают АЛУ выполнить арифметическую операцию с двумя регистрами (`R[s]` и `R[t]`) и записать результат в третий регистр (`R[d]`). Например, инструкция `1234` означает «прибавить `R[3]` к `R[4]` и поместить результат в `R[2]`», а инструкция `2AAC` — «вычесть `R[C]` из `R[A]` и поместить результат в `R[A]`». Можно сказать, что эти инструкции реализуют *целочисленный* тип данных TOY: множество значений образуют 16-разрядные целые числа, и для этого множества определяются операции сложения и вычитания.

код операции	описание	формат	псевдокод
0	останов	–	
1	сложение	RR	R[d] <- R[s] + R[t]
2	вычитание	RR	R[d] <- R[s] — R[t]
3	поразрядное AND	RR	R[d] <- R[s] & R[t]
4	поразрядное XOR	RR	R[d] <- R[s] ^ R[t]
5	сдвиг влево	RR	R[d] <- R[s] << R[t]
6	сдвиг вправо	RR	R[d] <- R[s] >> R[t]
7	загрузка адреса	A	R[d] <- addr
8	загрузка из памяти	A	R[d] <- M[addr]
9	запись в память	A	M[addr] <- R[d]
A	косвенная загрузка из памяти	RR	R[d] <- M[R[t]]
B	косвенное сохранение в память	RR	M[R[t]] <- R[d]
C	переход по нулю	A	if (R[d] == 0) PC <- addr
D	переход по положительному значению	A	if (R[d] > 0) PC <- addr
E	передача управления на адрес из регистра	–	PC <- R[d]
F	передача управления с сохранением точки возврата	A	R[d] <- PC; PC <- addr

Набор инструкций TOY

Инструкции работы с адресами памяти

Коды операций `7`, `A` и `B` относятся к *инструкциям работы с адресами памяти* TOY. Например, инструкция `7423` означает «присвоить `R[4]` значение `0023`» или просто «`R[4]` = `0023`» (обратите внимание на начальные нули). Инструкции с кодами операций `A` и `B` используются для *косвенных* обращений к памяти по адресу, записанному в регистре (так называемая *косвенная* или, точнее, *косвенная регистровая* адресация). Понимание этих инструкций поможет лучше понять работу ссылок в Java. Мы рассмотрим эти инструкции более подробно, когда будем рассматривать реализацию массивов и связных структур.

У кода операции `7` есть еще одно важное применение для работы с целыми числами: после того как биты будут загружены в регистр, их можно использовать в арифметических инструкциях (интерпретировать их как представление целого числа). Например, инструкция `7C01` записывает в регистр `R[C]` значение `0001` (`R[C]` = `0001`).

Инструкции логических операций

Коды операций с `3` по `6` зарезервированы за *логическими* инструкциями, которые приказывают АЛУ выполнить операции с *битами* в регистрах — по аналогии с операциями, которые были описаны для Java в разделе 5.1. Код операции `3` выполняет *битовую* операцию — поразрядное AND, при которой каждый бит `R[d]` равен 1, если оба соответствующих бита `R[s]` и `R[t]` равны 1; в противном случае бит равен 0. Код операции `4` соответствует битовой операции поразрядного XOR, при которой каждый бит `R[d]` равен 1, если оба соответствующих бита `R[s]` и `R[t]` различны; в противном случае бит равен 0. Код операции 5 заносит в `R[d]` результат сдвига содержимого `R[s]` влево на количество разрядов, хранящееся в регистре `R[t]`, с потерей выдвигаемых разрядов и заполнением освободившихся разрядов нулями по мере надобности. Операция с кодом 6 работает аналогично, но разряды сдвигаются вправо, а освобождающиеся разряды заполняются содержимым знакового разряда (см. подраздел «Вопросы и ответы» раздела 6.1). Логические инструкции завершают реализацию типа данных `int`, соответствующую правилам Java. В машине TOY, как и в языке Java, мы иногда изменяем правила абстракции данных, интерпретируя значения как последовательности из 16 битов, а не как целые числа. Логические инструкции — операции сдвига и битовые операции — применяются при реализации и декодировании любых других типов данных, какие, например, встречаются в коде Java.

операция поразрядного AND

```
   0101000111010111
&  0011000101101110
   0001000101000110
```

операция поразрядного XOR

```
   0101000111010111
^  0011000101101110
   0110000010111001
```

сдвиг влево на 6 разрядов

```
   0001000111010111
<< 0000000000000110
   0111010111000000
```

сдвиг вправо на 3 разряда (арифметический)

```
   0000000111010111
>> 0000000000000011
   0001000000111010

   1000000111010111
>> 0000000000000011
   1111000000111010
```

Логические инструкции: битовые операции и операции сдвига

Инструкции для работы с памятью

Коды операций 8 и 9 предназначены для *работы с памятью*, то есть передачи данных между ячейками памяти и регистрами. Например, инструкция 8234 означает «загрузить в R[2] содержимое слова памяти M[34]», или сокращенно R[2] = M[34]; а инструкция 9234 означает «сохранить в слове памяти M[34] содержимое R[2]», или M[34] = R[2].

Инструкции передачи управления

Коды операций с C до F предназначены для управления последовательностью выполнения программы: они изменяют содержимое PC и играют важную роль в реализации управляющих конструкций, таких как условные переходы, циклы и функции. Например, инструкция C212 означает «записать в PC значение 12, если значение R[2] равно 0», а D212 означает «записать в PC значение 12, если значение R[2] положительное». Обратите внимание: C0xx означает «записать в PC значение xx», так как регистр R[0] всегда содержит 0. Эта операция называется *безусловным переходом*. Изменение значения PC фактически означает передачу управления, потому что следующая инструкция всегда берется из адреса памяти, определяемого значением PC. Коды операций E и F будут более подробно описаны ниже, когда мы будем рассматривать реализацию функций.

Возможно, при виде этого набора инструкций вы подумали, что он в лучшем случае минимально достаточный. И конечно, это так[1], но одна из целей этой главы — убедить вас в том, что даже такое небольшое множество инструкций позволяет писать программы, эквивалентные программам Java, которые были представлены в первой половине книги (или любым другим программам). А пока важнее всего запомнить, что у вас имеется вся информация, необходимая для декодирования *любых* 16-разрядных значений как инструкций машины TOY.

Ваша первая программа для машины TOY

Ваша первая программа для машины TOY складывает два целых числа (листинг 6.2.1). Как и в случае с программой HelloWorld.java из раздела 1.1, мы начинаем с такой простой программы, чтобы сосредоточиться на подробностях ее выполнения. Код в листинге 6.2.1, называемый *машинным кодом*, демонстрирует некоторые соглашения, которые мы будем использовать для программ TOY.

- В листинг включаются все данные и код, относящиеся к программе.
- Каждая строка содержит адрес памяти (2 шестнадцатеричные цифры) и значение слова по этому адресу (4 шестнадцатеричные цифры).

---

[1] Тем не менее многие реальные микропроцессоры и микроконтроллеры имеют набор инструкций, лишь вдвое-втрое превосходящий TOY: например, 16-разрядный MSP430 — 51 инструкцию, 8-разрядный PIC12 — всего 35. — *Примеч. науч. ред.*

Листинг 6.2.1. Первая программа для машины TOY

```
10:  8A15  R[A] <- M[15]          загрузить первое слагаемое в a
11:  8B16  R[B] <- M[16]          загрузить второе слагаемое в b
12:  1CAB  R[C] <- R[A] + R[B]    c = a + b
13:  9C17  M[17] <- R[C]          сохранить результат
14:  0000  halt

15:  0008  integer value 810
16:  0005  integer value 510
17:  0000  result
```

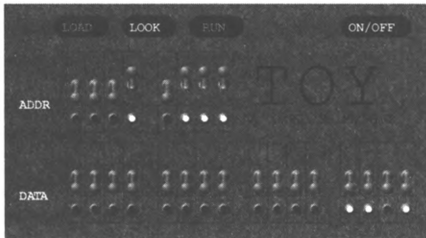
В начале работы счетчик PC содержит значение `10`. *Программа суммирует два числа в ячейках памяти* `15` *и* `16` *и помещает результат* `000D` *в ячейку памяти* `17`.

PC	IR	R[A]	R[B]	R[C]	M[17]
10	8A15	0008			
11	8B16	0008	0005		
12	1CAB	0008	0005	000D	
13	9C17	0008	0005	000D	000D
14	0000	0008	0005	000D	000D

Трассировка выполнения инструкций

дамп памяти

10	8A15
11	8B16
12	1CAB
13	9C17
14	0000
15	0008
16	0005
17	**000D**



0000 0000 0000 1101 ← *результат (двоичная запись)*
0 0 0 D ← *результат (шестнадцатеричная запись)*

- Начальное значение PC всегда равно адресу первой инструкции (выделяется жирным шрифтом).
- В третьем столбце приводится псевдокод каждой инструкции.

Сама программа состоит всего из 5 шестнадцатеричных чисел, каждое из которых состоит из 4 цифр. Эти числа хранятся в памяти по адресам `10`–`14`. С псевдокодом эта программа становится предельно понятной — внешне она очень похожа на программу Java.

Чтобы выполнить трассировку программы для машины TOY, достаточно записать значения PC и IR для каждой выполняемой инструкции, а также значение каж-

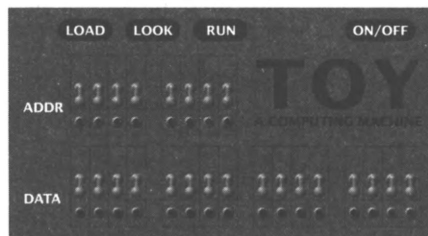
дого регистра или слова памяти, измененного после выполнения. В таблице под листингом приведена такая трассировка для программы 6.2.1. Чтобы привести результат вычислений, мы приводим содержимое памяти при достижении останова и выделяем саму инструкцию останова и измененные ячейки памяти жирным шрифтом. В данном случае изменяется только одно значение в памяти: в ячейку `17` записывается вычисленный результат `000D`.

На первый взгляд все совсем просто, но мы еще не описали процесс непосредственного выполнения программы. Для языка Java можно было привести четкое описание: вы создаете файл с программой, затем используете компилятор и виртуальную машину Java для его исполнения и просматриваете результат в терминальном окне. С машиной TOY все сложнее: у нее нет операционной системы, нет приложений, нет редактора, эмулятора терминала, компилятора или исполнительной среды — нет даже клавиатуры или экрана.

Вообще говоря, «результат» выполнения программы показан на диаграмме под листингом 6.2.1: на индикаторах в нижней части передней панели компьютера отображается вычисленный результат `000D`, в двоичной системе — `0000000000001101`. Далее мы шаг за шагом рассмотрим процесс запуска этой программы на машине TOY для достижения этого результата.

Управление машиной TOY

Вы работаете со своим компьютером при помощи устройств ввода/вывода, таких как клавиатура, экран или сенсорная панель. У машины TOY также есть свои устройства ввода/вывода — в определенном смысле. Пора разобраться в том, как программист взаимодействует с такими машинами, чтобы запустить на них программу. Внизу изображена передняя панель воображаемой машины TOY. Она оснащена тремя простыми устройствами для управления, ввода и вывода: кнопками, переключателями и световыми индикаторами. Все они представляют собой простые механизмы с двумя состояниями «вкл./выкл.», к тому же их совсем немного: всего 24 переключателя и индикатора и всего 4 кнопки. Ничего более. Ни клавиатуры, ни принтера, ни экрана. Ни подключения к Интернету, ни беспроводного адаптера, ни динамика, ни сенсорной панели; только кнопки для управления, переключатели для ввода, индикаторы для вывода. Но и этого достаточно для выполнения содержательных вычислений на компьютере общего назначения с такими же основными особенностями, как и у вашего компьютера.



Кнопки

Вероятно, простейшая управляющая операция с компьютером — его включение или выключение. У компьютера PDP-8 для этой цели служил ключ; у машины TOY — кнопка. Первое, что должен сделать программист, — включить машину (а последнее — выключить ее). Также кнопки выполняют еще три базовые функции.

- *Записать* слово в память компьютера.
- *Прочитать* значение слова из памяти компьютера.
- *Запустить* программу.

Вскоре мы рассмотрим все эти функции в контексте.

Переключатели

Программист, работающий с такой машиной, вводит двоичные значения при помощи переключателей. Переключатель в верхнем положении обозначает 1, а в нижнем — 0. Машина TOY оснащена двумя блоками переключателей: блок из 8 переключателей задает адрес ячейки памяти, а блок из 16 переключателей задает значение слова памяти. Эти переключатели являются *устройствами ввода* TOY.

Индикаторы

Устройствами вывода TOY служат блоки световых индикаторов под переключателями. И снова блок из 8 индикаторов используется для вывода адреса ячейки памяти, а блок из 16 индикаторов — для вывода значения слова памяти.

Запуск программы

Чтобы запустить программу на такой машине, как TOY, программист обычно приходит в заранее назначенное время с программой, записанной на листке бумаги. Чтобы вы поняли суть происходящего, мы во всех подробностях рассмотрим действия, необходимые для запуска первой программы. Для компактности мы будем «мыслить в шестнадцатеричной системе», указывая шестнадцатеричные значения для переключателей и индикаторов, тогда как в действительности каждый переключатель или индикатор соответствует одному биту. Например, когда мы говорим «установить на переключателях DATA данных `1CAB`», мы имеем в виду «установить переключатели DATA, соответствующие `1` в битовой строке `0001110010101011`». Вот как с учетом сказанного происходит реализация и запуск программы из листинга 6.2.1.

- Включите машину (нажмите кнопку ВКЛ/ВЫКЛ).
- Установите на переключателях ADDR значение `11`, а на переключателях DATA — `8B16`, нажмите кнопку LOAD.
- Установите на переключателях ADDR значение `12`, а на переключателях DATA — `1CAB`, нажмите кнопку LOAD.

- Установите на переключателях ADDR значение `13`, а на переключателях DATA — `9C01`, нажмите кнопку LOAD.
- Установите на переключателях ADDR значение `14`, а на переключателях DATA — `0000`, нажмите кнопку LOAD.
- Установите на переключателях ADDR значение `15`, а на переключателях DATA — `0008`, нажмите кнопку LOAD.
- Установить на переключателях ADDR значение `16`, а на переключателях DATA — `0005`, нажмите кнопку LOAD.
- Установите на переключателях ADDR значение `10`, нажмите кнопку RUN.
- Установите на переключателях ADDR значение `17`, нажмите кнопку LOOK.
- Запишите вычисленный ответ, отображаемый на индикаторах DATA (`000D`).
- (Обычно) повторите пять предыдущих шагов с другими значениями данных.
- Выключите машину (нажмите кнопку ВКЛ/ВЫКЛ).

Включая машину, мы ничего не можем предполагать относительно содержимого памяти, регистров и PC (кроме того, что `R[0]` содержит `0000`), поэтому перед запуском программы необходимо загрузить программу и данные в машину (семь шагов после включения машины). После запуска программы следует проверить содержимое ячейки памяти с результатом, чтобы узнать ответ.

Поразмыслите над фундаментальной природой этого интерфейса. По сути, программист обменивается данными с машиной *по одному биту*. Каким бы примитивным ни был этот процесс, раньше он устраивал разработчиков, потому что разработка программ была значительно эффективнее вычислений с бумагой и карандашом, логарифмической линейкой или арифмометром — единственными широко доступными альтернативами того времени. Вскоре вы увидите много примеров того, как короткие программы вычисляют значения, которые было бы очень трудно получить другим способом.

Конечно, вскоре появились более совершенные устройства ввода/вывода, такие как клавиатуры, принтеры, перфоленты и магнитные ленты, но, несомненно, именно этот метод программирования стал отправной точкой для многих ученых, инженеров и учащихся (да, именно так студенты университетов учились программировать в начале 1970-х годов).

Условные переходы и циклы

Чтобы рассматриваемые нами программы TOY становились все более интересными, мы воспользуемся тем же подходом, что и в самом начале изучения программирования в главе 1. В ходе изучения машинных инструкций мы рассмотрели типы данных и базовые операции; теперь можно перейти к управляющим конструкциям. Первыми управляющими конструкциями, рассмотренными в главе 1, были услов-

ные переходы и циклы. Итак, рассмотрим реализации этих конструкций на базе инструкций переходов машины TOY.

В качестве примера будет использован *алгоритм Евклида* для вычисления наибольшего общего делителя двух положительных целых чисел a и b. В разделе 2.3 была рассмотрена разновидность этого алгоритма для целочисленного деления (с остатком). Так как машина TOY не имеет инструкции деления, мы начнем со следующей версии, реализованной на Java:

```
public static int gcd(int a, int b)
{
    while (a != b)
        if (b > a) b = b - a;
        else       a = a - b;
    return a;
}
```

Этот код основан на простой идее: если b больше a, то любое число, являющееся делителем как a, так и b, также является делителем a и b-a (это относится и к наибольшему общему делителю); а если a больше b, то любое число также является делителем b и a-b. Если a и b положительные, каждая итерация цикла уменьшает большее из этих двух значений (так, что оно остается положительным), так что в в конечном итоге a и b оказываются равными наибольшему общему делителю всех участвовавших в этом процессе чисел, включая исходные значения a и b. Трассировка значений a и b для примеров ввода приведена справа.

a	b
7215	6123
1092	6123
1092	5031
1092	3939
1092	2847
1092	1755
1092	663
429	663
429	234
195	234
195	39
156	39
117	39
78	39
39	39

Трассировка евклидова алгоритма

Алгоритм Евклида был впервые сформулирован более 2000 лет назад, и с тех пор были изучены многие его версии. Эта конкретная версия иногда бывает медленной: например, можно заметить, что значение 1092 вычитается шесть раз, прежде чем a становится меньше b. (Операция получения остатка достигает той же цели за одну операцию деления.) Если одно из чисел очень мало, то алгоритм будет выполняться за время, пропорциональное абсолютному значению другого. Например, чтобы узнать, что наибольший общий делитель 7215 и 7214 равен 1, этому алгоритму потребуется 7214 итераций. Впрочем, довольно подробно были изучены и модификации алгоритма Евклида, обладающие намного большей эффективностью, — даже для таких машин, как TOY (см. упражнение 6.2.19).

На какое-то время мы сосредоточимся на реализации кода в теле функции, предполагая, что программист введет

входные данные в нужные ячейки памяти, как и в случае программы 6.2.1. В следующем разделе вы узнаете, как оформить этот код в функцию.

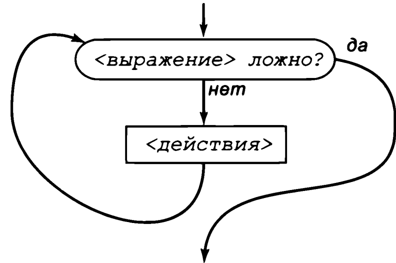
Ключевую роль в реализации играет эффективное использование инструкций передачи управления TOY для реализации циклов и условных переходов.

Чтобы реализовать цикл `while` (см. справа), мы размещаем код, вычисляющий значение выражения, с некоторой ячейки памяти *yy* и реализуем цикл инструкцией `C0`*yy* (безусловным переходом по адресу *yy*). Внутри цикла размещается код вычисления выражения, который оставляет `0` в некотором регистре (допустим, `R[1]`) в том и только в том случае, если значение ложно. Затем инструкция условного перехода `C1`*xx* передает управление команде после цикла (по адресу *xx*), если регистр содержит нуль. Реализация ветвления `if`, изображенная на с. 890, выполнена аналогично.

Код Java

```
while (<expression>)
    <statements>
```

блок-схема



код TOY

yy:	**машинный код для вычисления** <выражение> **в случае** false **в** R[1] **остается** 0	
C1xx	← **переход к** xx, **если значение** R[1] **равно** 0	
	машинный код для выполнения <действия>	
C0yy	← **переход к** yy	
xx:		

Реализация цикла

Диаграммы наглядно демонстрируют, что любой цикл или условный переход из программы Java можно напрямую реализовать в программе TOY. Каждая строка кода Java соответствует всего нескольким инструкциям переходов TOY. Чтобы это утверждение выглядело более убедительно, в нескольких упражнениях в конце этой главы рассматриваются реализации программ Java, описанных ранее в книге. Как и в программах на языке Java, с ветвлениями и циклами разработчик переходит из мира программ, состоящих из десятков строк кода, в мир программ с тысячами и миллионами выполняемых инструкций.

В программах TOY разработчик, впрочем, не ограничен только таким способом реализации ветвлений и циклов. Например, реализация из листинга 6.2.2 в действительности вычисляет логическое выражение лишь однократно: как для цикла `while`, так и для ветвления `if`. Программисты могут использовать инструкции передачи управления для создания на TOY таких управляющих конструкций, которые не имеют прямых аналогов среди условных переходов и циклов. Людям понадобилось много лет, чтобы сжиться с концепцией ограничения набора структурных элементов программ (условные переходы, циклы, вложение), которые применяются в современном программировании. Для простоты в книге будет использоваться промежуточный подход: мы будем руководствоваться конструкциями, рассмотренными ранее (и приводить Java-подобный код для документирования), но

будем применять сокращенную запись для упрощения кода там, где это уместно.

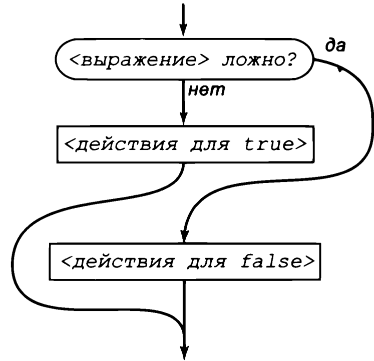
Таким образом, программист может при помощи переключателей ввести программу из листинга 6.2.2 и ее данные в ячейки памяти с 20 по 2D, установить на адресных переключателях значение 20, нажать кнопку RUN и понаблюдать за результатом на индикаторах, проверяя ячейку 2D. Программа вычисляет наибольший общий делитель 195 и 273, который равен 39. Программист может запускать программу сколько угодно раз: для этого достаточно ввести новые пары чисел в ячейки 2B и 2C, сбросить адресные переключатели в состояние 20, нажать кнопку RUN и проверить результат в 2D.

Отслеживание трассировки — хороший способ освоить программирование TOY. Сначала программа загружает число 195 (00C3) в R[a], а 273 (0111) в R[B]. Затем она выполняет вычитание и помещает результат −78 (FFC3) в R[C]. Так как значение отлично от нуля, программа входит в цикл и проверяет, является разность положительной или отрицательной. Так как значение отрицательное, программа вычитает R[A] из R[B], оставляя в R[B] значение 78 (004E), и возвращается к началу цикла. На следующей итерации программа вычитает R[B] из R[A], оставляя 117 (0075) в R[A]. Затем программа снова вычитает R[B] из R[A], оставляя в R[A] значение 39 (0027). Последняя итерация цикла вычитает R[A] из R[B], и в обоих регистрах остается результат 39 (0027).

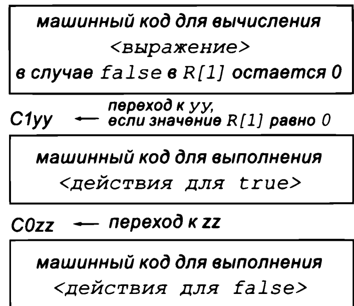
Код Java

```
if (<expression>)
   <statements for true>
else
   <statements for false>
```

блок-схема



код TOY



Реализация условного перехода

Алгоритм Евклида принадлежит к числу классических алгоритмов. Можно представить, сколько волнения испытывает начинающий программист, когда он видит, что подобные вычисления так легко поручаются компьютеру. Захочется ли кому-нибудь выполнять такие вычисления вручную? Возможность быстрой реализации подобных алгоритмов привлекала математиков и ученых, которые стали первыми программистами, стимулировала поиск более быстрых алгоритмов и инициировала исследования в разработке эффективных алгоритмов, продолжающиеся до сегодняшнего дня.

Листинг 6.2.2. Условные переходы и циклы: алгоритм Евклида

```
20:  8A2B  R[A] <- M[2B]               a = p
21:  8B2C  R[B] <- M[2C]               b = q
22:  2CAB  R[C] <- R[A] - R[B]         while (a != b)
23:  CC29  if (R[C] == 0) PC <- 29     {
24:  DC27  if (R[C]  > 0) PC <- 27        if (b > a)
25:  2BBA  R[B] <- R[B] - R[A]               b = b - a
26:  C022  PC <- 22                       else
27:  2AAB  R[A] <- R[A] - R[B]               a = a - b
28:  C022  PC <- 22                    }
29:  9A2D  вернуть a (= b = GCD)       return a
2A:  0000  останов
```

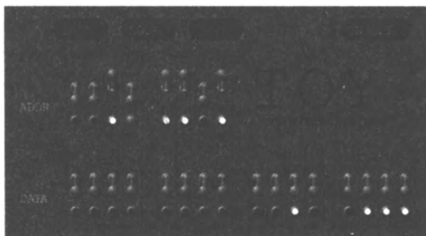
2B: 00C3 *целое значение* 195_{10}

2C: 0111 *целое значение* 273_{10}

2D: 0000 *результат*

В начале работы счетчик PC содержит значение 20. *Программа находит наибольший общий делитель двух чисел в ячейках* 2B *и* 2C *и помещает вычисленный результат в ячейку памяти* 2D.

PC	IR	R[A]	R[B]	R[C]
20	8A2B	00C3		
21	8B2C	00C3	0111	
22	2CAB	00C3	0111	FFC3
23	CC29	00C3	0111	FFC3
24	DC27	00C3	0111	FFC3
25	2BBA	00C3	004E	FFC3
26	C022	00C3	004E	FFC3
22	2CAB	00C3	004E	0075
23	CC29	00C3	004E	0075
24	DC27	00C3	004E	0075
27	2AAB	0075	004E	0075
28	C022	0075	004E	0075
22	2CAB	0075	004E	0027
23	CC29	0075	004E	0027
24	DC27	0075	004E	0027
27	2AAB	0027	004E	0027
28	C022	0027	004E	0027
22	2CAB	0027	004E	FFEA
23	CC29	0027	004E	FFEA
24	DC27	0027	004E	FFEA
25	2BBA	0027	0027	FFEA
26	C022	0027	0027	FFEA
22	2CAB	0027	0027	0000
23	CC29	0027	0027	0000
29	9A2D	0027	0027	0000
2A	0000	0027	0027	0000



0000 0000 0010 0111 ← *результат (двоичная запись)*

0 0 2 7 ← *результат (шестнадцатеричная запись)*

Выполнение программ, хранящихся в памяти

Одна из важнейших особенностей машины TOY заключается в том, что ее программы хранятся в виде чисел, причем как данные, так и программы размещаются в одной и той же общей основной памяти. Это глубокая концепция с интересной историей, а ее понимание совершенно необходимо для понимания природы вычислений.

Инструкции как данные и данные как инструкции

Для демонстрации этой фундаментальной идеи рассмотрим программу из листинга 6.2.3, предназначенную для суммирования чисел. Последовательность имеет любую длину и завершается значением `0000`. На с. 893 показана трассировка работы этой программы; она заслуживает того, чтобы тщательно разобраться в ней (`R[1]` и `M[1B]` исключены из трассировки, потому что их значения изменяются только один раз). Работа программы начинается примерно так же, как в листинге 6.2.1, но затем происходит нечто совсем иное. После суммирования первых двух чисел и сохранения результата в `R[A]` программа загружает инструкцию `8B1D` из ячейки `12` в `R[D]`, увеличивает ее на 1 и сохраняет полученный результат `8B1E` в той же ячейке `12`. Затем *происходит возврат к ячейке* `12`, и прочитанная из нее инструкция загружает в `R[B]` уже следующее число, которое прибавляется к `R[A]`; цикл продолжается до тех пор, пока в данных не будет обнаружено значение `0000`.

Такой код называется *самомодифицирующимся*. Мы привели эту программу, потому что она в компактной форме демонстрирует фундаментальную концепцию программ, хранящихся в памяти. Что содержит ячейка памяти `12` — инструкцию программы или данные? И то и другое! Когда мы увеличиваем ее содержимое на 1, это данные, но при обращении к ней путем считывания по адресу PC и загрузки в IR это инструкция. Так как программа и данные хранятся в общей памяти, машина может изменять во время выполнения как свои данные, так и собственный код. Таким образом, код и данные — одно и то же (или, по крайней мере, могут быть одним и тем же). Содержимое памяти интерпретируется как инструкция, когда к нему обращается счетчик адреса программы, и как данные — когда к нему обращается инструкция.

Самомодифицирующийся код редко встречается в современных вычислениях, потому что он создает слишком много трудностей с пониманием, отладкой и сопровождением[1]. В следующем разделе будет рассмотрен альтернативный метод суммирования последовательности чисел. Тем не менее возможность интерпретации инструкций как данных чрезвычайно важна для программирования, и мы еще вернемся к этой теме в оставшейся части этой главы.

---

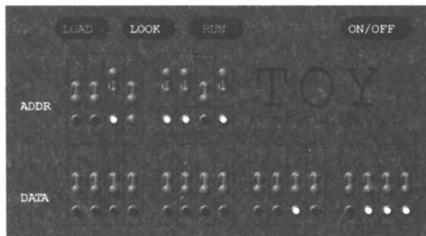
[1] Также подобный код не может работать непосредственно в постоянной памяти (ПЗУ, Read Only Memory — ROM). Размещение программы в ПЗУ для ряда применений предпочтительно или даже необходимо. — *Примеч. науч. ред.*

Листинг 6.2.3. Самомодифицирующийся код: вычисление суммы

```
10: 7101  R[1] <- 0001
11: 8A1C  R[A] <- M[1C]          загрузить первое число в a
12: 8B1D  R[B] <- M[1D]          while (b != 0)
13: CB19  if (R[B]==0) PC <- 19  {
14: 1AAB  R[A] <- R[A] + R[B]       a = a + b
15: 8D12  R[D] <- M[12]             изменить инструкцию в M[12]
16: 1D1D  R[D] <- R[D] + 1            чтобы загрузить следующее число
17: 9D12  M[12] <- R[D]               в b при следующей итерации
18: C012  PC <- 12               }
19: 9A1B  сохранить результат
1A: 0000  останов

1B: 0000  результат
1C: 0001  данные
1D: 0008  целое значение 8₁₀
1E: 001B  целое значение 27₁₀
1F: 0040  целое значение 64₁₀
20: 0000
```

В начале работы счетчик PC содержит значение 10. *Программа суммирует последовательность чисел в ячейках с* 1C *до* 1F *(завершаемую* 0000*) и сохраняет результат в ячейке памяти* 1B.

PC	IR	R[A]	R[B]	R[D]	M[12]
10	7101				8B1D
11	8A1C	**0001**			8B1D
12	8B1D	0001	**0008**		8B1D
13	CB19	0001	0008		8B1D
14	1AAB	**0009**	0008		8B1D
15	8D12	0009	0008	**8B1D**	8B1D
16	1D1D	0009	0008	**8B1E**	8B1D
17	9D12	0009	0008	8B1E	**8B1E**
18	C012	0009	0008	8B1E	8B1E
12	8B1E	0009	**001B**	8B1E	8B1E
13	CB19	0009	001B	8B1E	8B1E
14	1AAB	**0024**	001B	8B1E	8B1E
15	8D12	0024	001B	**8B1E**	8B1E
16	1D1D	0024	001B	**8B1F**	8B1E
17	9D12	0024	001B	8B1F	**8B1F**
18	C012	0024	001B	8B1F	8B1F
12	**8B1F**	0024	**0040**	8B1F	8B1F
13	CB19	**0064**	0040	8B1F	8B1F
14	1AAB	0064	0040	8B1F	8B1F
15	8D12	0064	0040	**8B1F**	8B1F
16	1D1D	0064	0040	**8B20**	8B1F
17	9D12	0064	0040	8B20	**8B20**
18	C012	0064	0040	8B20	8B20
12	8B20	0064	**0000**	8B20	8B20
13	CB19	0064	0000	8B20	8B20
19	9A1B	**0064**	0000	8B20	8B20
1A	0000				



0000 0000 0010 0111 ← *результат (двоичная запись)*
0 0 2 7 ← *результат (шестнадцатеричная запись)*

Некоторые последствия

После некоторых размышлений становится ясно, что примеры интерпретации программы как данных неоднократно встречаются в современной вычислительной инфраструктуре.

- Любое приложение интерпретируется как данные во время загрузки или установки и лишь после запуска — как программа.
- Компиляторы — программы, которые читают текст других программ (входные данные) и генерируют программы на машинном языке (выходные данные), так что все *языки программирования* напрямую зависят от этой возможности.
- Современные *облачные технологии* зависят от концепции *виртуальной машины*, то есть запуска программы, написанной для одного компьютера, на другом компьютере. В самом деле, сама машина TOY является виртуальной машиной, как будет показано в разделе 6.4.

Конечно, это далеко не полный список примеров, и мы еще вернемся к этой концепции в этой главе.

Интерпретация программ как данных сопряжена с определенным риском. Например, компьютерные вирусы представляют собой (вредоносные) программы, механизм распространения которых основан на написании новых или изменении существующих программ. При этом, как было показано в главе 5, из теории Тьюринга следует, что не существует эффективного способа отличить вредоносный вирус от полезного приложения или данных. Этот практический недостаток является неизбежным следствием модели программ, хранимых в памяти[1]. Конкретный пример в контексте машины TOY рассматривается в следующем разделе.

Машины фон Неймана

Как упоминалось ранее, в 1940-е и 1950-е годы ученые и инженеры интенсивно занимались машинными вычислениями, и не только в таких военных областях, как баллистика, атомное оружие и криптография, но и в вполне мирных сферах, например для космических полетов и метеорологии. Электронные компоненты могли работать намного быстрее механических, и это обещало большие перспективы.

Впрочем, многие ранние компьютеры моделировали работу механических калькуляторов. Оператору приходилось «программировать» компьютер, подключая кабели и устанавливая переключатели; это занятие было монотонным, долгим и ненадежным. Вся память предназначалась для хранения данных. Одной из таких

[1] Это актуально не только для машинного кода, но и для программ, хранящихся в виде текста и выполняемых интерпретатором (или виртуальной машиной, например Java). Проблема частично решается разграничением *прав доступа* к областям памяти. Подробнее см. раздел 6.4. — *Примеч. науч. ред.*

машин был ENIAC, разрабатывавшийся в середине 1940-х годов в Университете Пенсильвании Эккертом (Eckert) и Мокли (Mauchly).

Джон фон Нейман (1903–1957)

В то же время (а точнее, чуть раньше — в 1930-х годах) царил ажиотаж и в области математики, потому что Алан Тьюринг представил свои оригинальные теоретические конструкции, рассмотренные в главе 5. Работа Тьюринга позволила лучше понять истинную природу вычислений.

Ученый из Принстонского университета Джон фон Нейман работал консультантом у Эккерта и Мокли над проектом ENIAC и его запланированным наследником EDVAC. Он интересовался как баллистическими вычислениями, которые были главной целью машины, так и масштабными вычислениями, необходимыми для разработки атомной бомбы.

В 1945 году во время поездки на поезде из Принстона в Лос-Аламос фон Нейман писал свой отчет о запланированных усовершенствованиях в ENIAC. Этот документ, «Первый проект отчета по EDVAC», содержал полное описание модели вычислительного устройства с программой, хранимой в памяти. Так как фон Нейман был профессором в Принстоне в то время, когда Тьюринг был аспирантом в этом же университете, на него, безусловно, повлияли идеи Тьюринга, и ему представилась уникальная возможность для заполнения пробела между теориями Тьюринга и практическими трудностями, с которыми столкнулись Эккерт и Мокли. Вскоре после прибытия фон Неймана в Лос-Аламос молодой лейтенант по имени Герман Голдстайн (Herman Goldstine), поняв, какой интерес представляет эта идея, позаботился о широком распространении документа. Ученые по всему миру немедленно осознали потенциал модели с программой, хранимой в памяти, и с тех пор компьютеры, построенные на ее основе (практически все компьютеры), называются ***машинами фон Неймана***. Многие историки полагают, что заслуга также в равной степени принадлежит Эккерту и Мокли (как и Тьюрингу!), но именно документ фон Неймана, безусловно, стал той искрой, которая распространила эту идею по миру.

Модель программы, хранимой в памяти, позволяет компьютерам выполнять любые вычисления, не требуя от пользователя физической модификации или изменения конфигурации оборудования. Эта простая, но фундаментальная модель использовалась практически во всех компьютерах с того времени, когда фон Нейман впервые сформулировал ее.

По прошествии времени архитектура фон Неймана кажется очевидной. Тем не менее многие исследовательские группы в то время работали в других направлениях, а вопрос о том, будет ли компьютер с хранимой в памяти программой таким же мощным и универсальным, как и компьютер с изменяемой физической конфи-

гурацией, оставался открытым. Собственно, теория Тьюринга как раз показывает, что способность физического изменения конфигурации компьютера не позволит ему решать больше задач (при условии, что он уже обладает достаточно полным базовым набором инструкций как в случае с машиной TOY)[1]. Фон Нейман был одним из немногих людей в мире (не считая самого Тьюринга), которые смогли осознать этот факт. Его умение полностью сформулировать идею в практическом контексте во время длинной поездки в поезде стало счастливым озарением, которое глубоко изменило мир.

Вопросы и ответы

В. Программисты действительно вводили программы на переключателях?

О. Да. Многим, многим людям приходилось этим заниматься. Даже когда появились более совершенные устройства ввода/вывода, программы вводились на переключателях, которые управляли работой таких устройств.

В. Чем регистр отличается от слова памяти?

О. И в регистре, и в слове памяти хранятся 16-разрядные целые числа, но они играют разную роль в работе компьютера. Память предназначена для хранения программ и данных — и ее объем должен быть как можно больше. Регистры предназначены для промежуточного хранения данных, передаваемых и получаемых от АЛУ, — и для работы машины необходимо небольшое количество регистров. Как правило, для регистров в компьютерах используется более дорогостоящая технология, потому что регистры задействованы практически в каждой инструкции и доступ к ним должен быть как можно более быстрым. Количество регистров в компьютере является проектировочным решением.

В. В наши дни еще выпускаются специализированные компьютеры или микропроцессоры?

О. Да, потому что на аппаратном уровне простые операции можно выполнять быстрее, чем на программном.

В. Трудно представить себе программирование без специальных конструкций для циклов и условных переходов. Люди действительно так работали?

О. Конечно. Программисты планировали логику программ на блок-схемах, и в ранних языках высокого уровня существовала инструкция перехода `goto`, которая преобразовывалась в соответствующую инструкцию на машинном языке. Идея «струк-

[1] Помимо фон-неймановской (или *принстонской*), используется (и имеет свои преимущества) *гарвардская* архитектура с отдельными памятью программ и памятью данных, а также их модификации и комбинации. Тем не менее они следуют единым фундаментальным принципам: программное управление и хранение программ в памяти. Физически перестраиваемые архитектуры также находят применение — как правило, для достижения экстремального быстродействия. — *Примеч. науч. ред.*

турного программирования», использующего только циклы, ветвления и функции, родилась в академической среде, но не воспринималась программистами серьезно до 1970-х годов. Знаменитой поворотной точкой стало письмо Эдсгера Дейкстры редактору журнала *Communications of the ACM*, написанное в 1968 году; оно было озаглавлено «О вреде GOTO» (*Goto Statement Considered Harmful*). В своем письме Дейкстра аргументировал необходимость исключения безусловного перехода `goto` из всех языков высокого уровня как создающего слишком много проблем с пониманием, отладкой и сопровождением программ[1]. За десятилетие эта точка зрения стала общепринятой, и с тех пор полезность структурного программирования уже не вызывает сомнений.

Упражнения

6.2.1. Сколько битов памяти содержит машина TOY (с учетом всех регистров вместе с PC и основной памятью)?

6.2.2. Машина TOY использует 8-разрядные адреса памяти, а это означает, что машина может обращаться к 256 словам памяти. Сколько слов памяти можно адресовать с 32-разрядными адресами? 64-разрядными адресами?

6.2.3. Допустим, вы хотите использовать такой же формат инструкций, как в машине TOY, но с 32-разрядными адресами. Какой размер слова должен использоваться? Опишите проблему, присущую такой архитектуре.

6.2.4. Приведите одну инструкцию, которая заносит в счетчик адреса программы адрес памяти `15` независимо от содержимого других регистров или ячеек памяти.

Решение. `C015` или `F015`. В обеих инструкциях используется то, что регистр `R[0]` всегда содержит 0000.

6.2.5. Приведите 7 инструкций (с разными кодами операций), помещающих значение `0000` в регистр `A`.

Решение. `1A00`, `2A`xx, `3A0`x, `4A`xx, `5A0`x, `6A0`x, `7A00`, где x — любая шестнадцатеричная цифра.

6.2.6. Приведите три инструкции (с разными кодами операций), заполняющие счетчик адреса программы значением `00` без изменения содержимого других регистров или ячеек памяти.

Решение. `C000`, `E0`xy, `F000`.

6.2.7. Приведите пять инструкций (с разными кодами операций), которые фактически являются «пустыми». Исключите инструкции, у которых вторая цифра равна 0.

Решение. `1`xx`0`, `1`x`0`x, `2`xx`0`, `3`xxx, `5`xx`0`, `6`xx`0` или `D0`xx, где x — любая шестнадцатеричная цифра, кроме `0`.

[1] Стоит заметить, что редактором журнала был Никлаус Вирт. — *Примеч. науч. ред.*

6.2.8. Приведите шесть инструкций, записывающих содержимое R[B] в R[A].

Решение. 1AB0, 1A0B, 2AB0, 3ABB, 4A0B, 4AB0, 5AB0 и 6AB0.

6.2.9. В наборе инструкций TOY нет инструкции перехода по неотрицательному значению. Объясните, как перейти на адрес памяти 15, если содержимое R[A] больше или равно 0.

Решение. Используйте инструкции перехода по положительному значению и по нулю одну после другой: CA15 DA15.

6.2.10. Заполните пропуски в таблице:

двоичная запись	шестнадцатеричная запись	инструкция TOY
0001001000110100	1234	R[2] <- R[3] + R[4]
1111111111111111	FFFF	
1111101011001110	FACE	
0101011001000100	5644	
1000000000000001	8001	
0101000001000011	5043	
0001110010101011	1CAB	
		R[F] <- R[F] & R[F]
		R[8] <- M[88]
	7777	
		if (R[C] == 0) PC <- CC

6.2.11. Среди инструкций TOY нет получения абсолютного значения (модуля). Приведите последовательность инструкций TOY, которая присвоит R[d] абсолютное значение R[s].

6.2.12. Среди инструкций TOY нет операции поразрядного NOT-AND (AND с отрицанием). Приведите последовательность из трех инструкций TOY, которая присвоит каждому разряду R[d] значение 1 в том и только в том случае, если хотя бы один из соответствующих разрядов R[s] и R[t] равен 0.

6.2.13. Среди инструкций TOY нет операции поразрядного OR. Приведите последовательность из трех инструкций TOY, которая присвоит каждому разряду R[d] значение 1 в том и только в том случае, если хотя бы один из соответствующих разрядов R[s] и R[t] равен 1.

Решение. 3DAB 4EAB 1CDE.

6.2.14. Среди инструкций TOY нет операции поразрядного NOT-OR (OR с отрицанием). Приведите последовательность из трех инструкций TOY, которая присвоит каждому разряду `R[d]` значение 0 в том и только в том случае, если хотя бы один из соответствующих разрядов `R[s]` и `R[t]` равен 1.

6.2.15. Среди инструкций TOY нет битовой операции отрицания NOT. Приведите последовательность из трех инструкций TOY, которая присвоит каждому разряду `R[d]` значение, обратное значению соответствующего разряда `R[s]`.

Решение. `7101 2B01 4BAB` или `7101 2B01 2BBA`.

6.2.16. Покажите, что инструкция вычитания избыточна. Иначе говоря, объясните, как вычислить `Rd = Rs - Rt` последовательностью инструкций TOY, не включающих код операции `2`.

6.2.17. Какие из 16 инструкций TOY не используют все 16 разрядов?

Решение. Инструкция останова (используются только первые 4 разряда), косвенной загрузки из памяти (не использует третью шестнадцатеричную цифру), косвенного сохранения в памяти (не использует третью шестнадцатеричную цифру), перехода по содержимому регистра (не использует две шестнадцатеричные цифры).

6.2.18. Некоторые целые значения TOY интерпретируются как отрицательные в дополнительном коде. Для каких инструкций это существенно?

Решение. Инструкция перехода по положительному значению интерпретирует целые числа от `0001` до `7FFF` как положительные. Инструкция сдвига вправо является арифметическим сдвигом, то есть если крайний левый бит равен 1, то освободившиеся биты заполняются 1. Все остальные инструкции (даже вычитание!) не зависят от того, поддерживает ли машина TOY отрицательные целые числа.

6.2.19. Доработайте функцию TOY для вычисления наибольшего общего делителя (`gcd`), приведенную в тексте; для этого добавьте переменную `c`, инициализированную 0, и измените цикл `while` следующим образом.

- Если `a` и `b` четные, то `gcd(a, b) = 2 gcd(a/2, b/2)`, поэтому разделите `a` и `b` на 2 и увеличьте `c` на 1.
- Если `a` четное, а `b` нечетное, то `gcd(a, b) = gcd(a/2, b)`, поэтому разделите `a` на 2.
- Если `a` нечетное, а `b` четное, разделите `b` на 2 (по аналогии).
- Остается одна возможность: `a` и `b` нечетные, поэтому действуйте не как прежде (замените большее число их разностью, которая, кстати, четная).

В конце сдвиньте результат на `c` позиций влево, чтобы учесть исключенные степени 2. Напишите программу, которая читает из стандартного ввода два целых числа и записывает в стандартный вывод их наибольший общий делитель. (Анализ, выходящий за рамки книги, показывает, что время выполнения этого алгоритма для худшего случая квадратично зависит от разрядности чисел — это намного быстрее, чем в версии, описанной в тексте.)

6.3. Программирование на машинном языке

В этом разделе мы продолжим разговор о том, как механизмы языка Java, упоминавшиеся ранее, реализуются на машине TOY — в контексте программ TOY, решающих актуальные задачи. При этом мы прежде всего постараемся убедить вас в том, что программирование TOY может быть ничуть не менее интересным и творческим, чем программирование на Java, а машина TOY намного мощнее, чем может показаться на первый взгляд.

Говоря более конкретно, мы рассмотрим программы TOY, которые реализуют *функции*, *массивы*, *стандартный ввод и вывод*, а также *связные структуры*, то есть все основные структурные элементы, применяемые в программировании. В принципе, это показывает, что для каждой из написанных нами программ на Java можно написать соответствующую программу на машинном языке. Возможность такого преобразования сомнений не вызывает, потому что компилятор Java именно это и делает.

Конечно, машина может накладывать ограничения по ресурсам. Возможно ли выполнить полезные вычисления с 4096 битами памяти? В этом разделе мы постараемся показать, что это возможно. Тем не менее технологические достижения тоже нельзя игнорировать. В разделе 6.4 будут рассмотрены расширения TOY, с которыми эта машина становится чуть более похожей на современный компьютер без изменения модели программирования. На такой машине, безусловно, можно написать программу TOY, способную выполнить любые вычисления, выполняемые любой программой Java на вашем компьютере.

Через практические примеры вы увидите, что на машинном языке можно разрабатывать программы для выполнения самых разных задач. Более того, многие из первых прикладных программ были реализованы именно таким способом, потому что в течение многих лет снижение быстродействия от использования языков высокого уровня оказывалось слишком значительным. Большая часть такого кода была написана на *языке ассемблера* (или просто *ассемблере*) — языке, который имеет много общего с машинным, но позволяет использовать символические имена для кодов операций, регистров и ячеек памяти (см. упражнение 6.4.13). В 1970-е годы нередко встречались написанные на языке ассемблера программы из десятков тысяч строк. Впрочем, программисты даже сегодня пишут такой код для приложений, критичных по быстродействию.

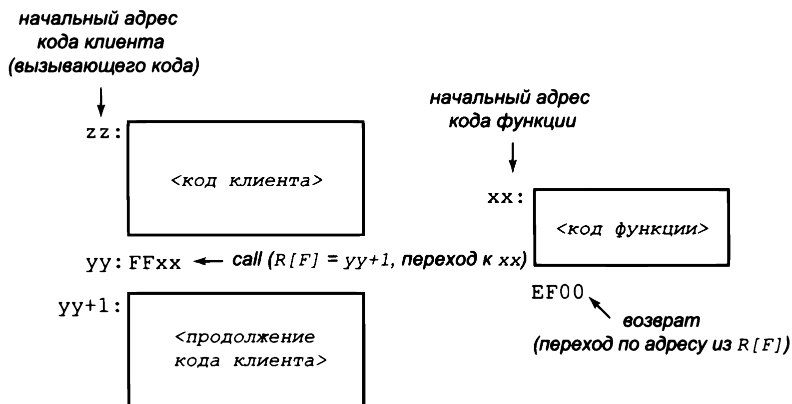
А самое главное — мы постараемся дать вам представление о том, что в действительности происходит в компьютере при выполнении вашей программы.

Функции

После условных переходов и циклов в главе 2 рассматривались *функции*. Инструкции передачи управления TOY приспособлены для этого. Поскольку мы преследу-

ем цель продемонстрировать концепции, мы выбрали один из простейших способов реализации функций.

- Передать управление функции.
- Передать аргументы от клиента (из вызывающего кода) в функцию.
- Вернуть значение из функции клиенту (в вызывающий код).
- Вернуть управление клиенту.



Реализация вызова функции

В выполнении всех этих задач нам помогут регистры. Для алгоритма Евклида последовательность операций выглядит так.

- Использовать инструкцию передачи управления с сохранением точки возврата. Инструкция сохраняет в заданном регистре *адрес возврата* (то есть адрес памяти следующей инструкции в вызывающем коде); мы используем регистр `R[F]`.
- Использовать `R[A]` и `R[B]` для аргументов и возвращаемых значений.
- Использовать инструкцию передачи управления по адресу в регистре для возврата управления в вызывающий код. Говоря конкретнее, `EF00` записывает в PC сохраненный адрес возврата из `R[F]`.

Типичная схема передачи управления при вызове функции изображена вверху.

В соответствии с этими правилами код листинга 6.2.2 легко преобразуется в *функцию*, которая вычисляет наибольший общий делитель `R[A]` и `R[B]` и оставляет результат как в `R[A]`, так и в `R[B]`: в `M[29]` записывается инструкция передачи управления по регистру `EF00`, чтобы можно было использовать инструкцию передачи управления с сохранением точки возврата `FF22` для вызова функции. Обновленная реализация приведена на с. 902.

функция для вычисления наибольшего общего делителя

```
22:  2CAB  R[C] <- R[A] - R[B]      c = a - b
23:  CC29  if (R[C] == 0) PC <- 29  while (a != b)
24:  DC27  if (R[C]  > 0) PC <- 27  {  if (b > a)
25:  2BBA  R[B] <- R[B] - R[A]            b = b - a
26:  C022  PC <- 22                    else
27:  2AAB  R[A] <- R[A] - R[B]            a = a - b
28:  C022  PC <- 22                 }
29:  EF00  PC <- R[F]               return
                                    [R[A] = R[B] = GCD]
```

В листинге 6.3.1 приведен код клиента, который с помощью этой функции проверяет, является ли целое число простым. (И снова: эта задача была бы проще, если бы в наборе инструкций присутствовала инструкция деления!) Способ проверки прост: число является простым в том и только в том случае, если наибольший общий делитель его и каждого меньшего положительного числа равен 1 (если у числа имеется делитель, больший 1, то наибольшим общим делителем исходного числа и делителя будет сам делитель). Как и в случае с листингом 2.3.1, вычисления прекращаются при достижении верхней границы наименьшего множителя. Так как функция квадратного корня недоступна, мы просто используем 255, потому что 255^2 больше любого положительного 16-разрядного числа в дополнительном коде. (Можно также вычислить другую верхнюю границу для квадратного корня из заданного числа.)

В трассировке под кодом программы приводятся значения `R[9]`, `R[A]` и `R[B]` до и после вызова функции, а также при достижении инструкции останова.

Обратите внимание: функция использует регистр `R[C]`, так что вызывающая программа не может рассчитывать на то, что `R[C]` после вызова функции будет иметь то же значение, как перед вызовом. И конечно, функция не может использовать `R[9]`, потому что вызывающая программа хранит в этом регистре индекс, или `R[F]`, потому что здесь хранится адрес возврата. При такой нехватке ресурсов подобные соглашения («контракты») между программами играют важную роль в программировании на столь низком уровне; это распространенное явление.

Как и в случае с условными переходами и циклами, в TOY можно реализовать механизмы вызова функций, не имеющие подходящих аналогов в Java. Например, для хранения возвращаемых значений можно использовать сразу несколько регистров. Некоторые программисты сохраняют все регистры перед вызовом функции — это одна крайность; другие программисты требуют, чтобы каждая функция сохраняла значения всех регистров перед выполнением своей работы, а затем восстанавливала их исходные значения перед возвратом управления. В современных механизмах вызова функций чаще применяется второй вариант.

Механизм вызова функции, использованный в листинге 6.3.1, не подходит для рекурсивных функций, но для них можно разработать более общий механизм с использованием стека (см. упражнение 6.3.27). Как обычно, наша цель не описать все подробности, а только убедить вас в том, что базовые конструкции, существующие в Java, достаточно легко реализуются на машинах типа TOY.

Листинг 6.3.1. Вызов функции: проверка числа на простоту

```
30:  7101  R[1] <- 0001
31:  75FF  R[5] <- 255
32:  7901  R[9] <- 0001            i = 1
33:  2C59  R[C] <- R[5] - R[9]     while (i < 255)
34:  CC3B  if (R[C] == 0) PC <- 3B {
35:  1991  R[9] <- R[9] + R[1]       i = i + 1
36:  1A09  R[A] <- R[9]              a = i
37:  8B3D  R[B] <- M[3D]             b = p
38:  FF22  R[F] <- PC; PC <- 22       a = b = gcd(a, b)
39:  2AA1  R[A] <- R[A] - R[1]       if (gcd(a, b) != 1) break
3A:  CA36  if (R[A] == 0) PC <- 36 }
3B:  9B3E  M[3E] <- R[B]             x = 1 если p простое число
3C:  0000  останов

3D:  005B  целое значение 91₁₀     p
3E:  0000  результат               x
```

В начале работы счетчик PC содержит значение 30. Программа проверяет, является ли число в M[3D] простым. Результат равен 1, если число простое, и наименьший делитель, больший 1, в противном случае. Программа использует функцию `gcd()`*, приведенную ранее, вызывая ее инструкцией FF22.*

PC	IR	R[A]	R[B]	R[9]	M[3E]
37	8B3D	0002	005B	0002	0000
38	FF22	0001	0001	0002	0000
37	8B3D	0003	005B	0003	0000
38	FF22	0001	0001	0003	0000
37	8B3D	0004	005B	0004	0000
38	FF22	0001	0001	0004	0000
37	8B3D	0005	005B	0005	0000
38	FF22	0001	0001	0005	0000
37	8B3D	0006	005B	0006	0000
38	FF22	0001	0001	0006	0000
37	8B3D	0007	005B	0007	0000
38	FF22	0007	0007	0007	0007
3B	0000	0006	0007	0007	0007

трассировка для PC = 37, 38 и 3B

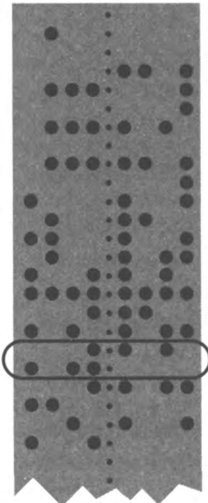
i	gcd(i, 91)
2	1
3	1
4	1
5	1
6	1
7	7

Этот пример показывает, что преимущества модульного программирования, которые были описаны для Java, также относятся и к программам на машинном языке TOY. После того как код для вычисления НОД или проверки простоты числа будет отлажен, мы сможем использовать его в других программах всего одной инструкцией TOY. Эта возможность позволила создавать уровни абстракции, быстро поднявшие программирование на машинном языке до удобного для работы уровня и приведшие к развитию многих аспектов программной инфраструктуры, которые продолжают использоваться до сих пор.

Стандартный вывод

Конечно, одним из первоочередных усовершенствований после изобретения самого компьютера стала разработка новых средств обмена данными с компьютером, более удобных, чем переключатели и индикаторы. Для этой цели применялись самые разнообразные устройства. Мы выбрали для TOY коммуникационный механизм, усеченный до минимума: бумажную перфорированную ленту. Как и сама машина TOY, этот механизм может показаться крайне примитивным, но он широко применялся не менее десяти лет.

Перфолента — простой носитель информации с наглядным кодированием двоичных чисел. Для TOY мы выбрали кодировку, очень похожую на ту, что использовалась на уже упоминавшихся старых компьютерах PDP-8. Каждое 16-разрядное двоичное число кодируется на ленте двумя рядами, каждый из которых представляет 8 битов; в позициях, соответствующих 1, в ленте пробивается отверстие, а в позициях, соответствующих 0, отверстий нет. По центру ленты идет ряд маленьких отверстий, которые в прошлом использовались для протяжки ленты через перфоратор зубчатым колесом. Перфоратор получает 16-разрядное двоичное слово от компьютера, пробивает отверстия, соответствующие этому слову (в двух рядах), и перемещает ленту, чтобы подготовиться к перфорации следующего слова. Вместо индикаторов и переключателей программист пишет программу для вывода информации на перфоленту, потом просматривает ее (или, как вы вскоре увидите, снова передает информацию с нее машине).

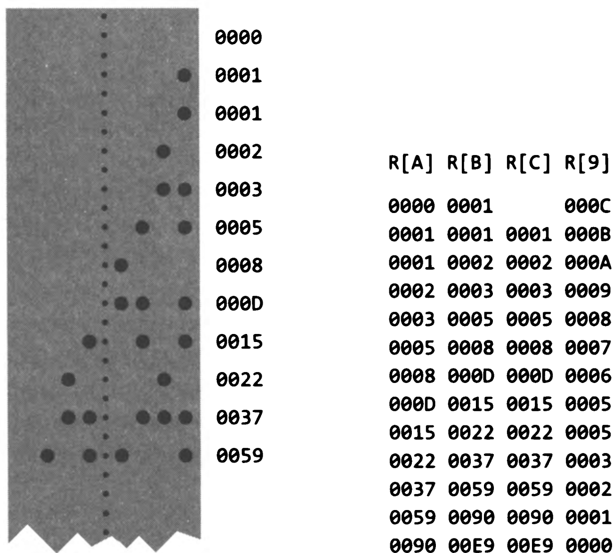


Перфолента: реальная и воображаемая

Листинг 6.3.2. Стандартный вывод: числа Фибоначчи

```
40:  7101  R[1] <- 0001
41:  7A00  R[A] <- 0000              a = 0
42:  7B01  R[B] <- 0001              b = 1
43:  894C  R[9] <- M[4C]             i = n
44:  C94B  if (R[9] == 0) PC <- 4B   while (i > 0) {
45:  9AFF  R[A] в стандартный вывод     print(a)
46:  1CAB  R[C] <- R[A] + R[B]          c = a + b
47:  1AB0  R[A] <- R[B]                 a = b
48:  1BC0  R[B] <- R[C]                 b = c
49:  2991  R[9] <- R[9] - 1             i = i - 1
4A:  C044  PC <- 44                  }
4B:  0000  останов

4C:  000C  целое значение 12_10       n
```

Первоначально счетчик PC содержит значение 40. Программа записывает в стандартный вывод первые n ненулевых чисел Фибоначчи, где n — целое значение в ячейке памяти 4C.



R[A]	R[B]	R[C]	R[9]
0000	0001		000C
0001	0001	0001	000B
0001	0002	0002	000A
0002	0003	0003	0009
0003	0005	0005	0008
0005	0008	0008	0007
0008	000D	000D	0006
000D	0015	0015	0005
0015	0022	0022	0005
0022	0037	0037	0003
0037	0059	0059	0002
0059	0090	0090	0001
0090	00E9	00E9	0000

выходная лента *трассировка для PC = 44*

Как заставить наш компьютер TOY вывести слово на ленту? Ответ на этот вопрос прост: мы резервируем для этой цели ячейку памяти FF и соединяем наши устройства так, что каждый раз при записи слова в эту ячейку перфоратор активизируется и выводит записываемое значение на ленту[1].

Программа в листинге 6.3.2 вычисляет числа Фибоначчи и выводит их на бумажную перфоленту. Вычисления вполне очевидны: два предыдущих числа Фибоначчи хранятся в R[A] и R[B], при этом R[A] инициализируется 0, а R[B] инициализируется 1. Затем мы входим в цикл, в котором следующее число Фибоначчи вычисляется суммированием R[A] и R[B] и сохранением результата в R[C], после чего происходит обновление R[A] и R[B] с копированием R[B] в R[A] и R[C] в R[B]. На каждой итерации цикла выполняется инструкция 9AFF, которая переносит содержимое R[A] на ленту. Результат на ленте изображен под листингом. Если сравнить ленту с содержимым трассировки справа, вы сможете прочитать числа Фибоначчи в двоичной записи с ленты — точно так же, как когда-то это делали люди, программировавшие похожие машины. Обратите внимание: программа TOY не управляет перфоратором явно, она только выполняет инструкции 9AFF. Эта простая абстракция позволяет заменить перфоратор другим устройством вывода: телетайпом, устройством записи на магнитную ленту и т. д. — без какого-либо изменения программы TOY! Данная схема в некоторой степени является предшественником абстракции *стандартного вывода*, которая используется до сих пор.

В частности, перфолента реализует одну из самых важных характеристик стандартного вывода: у длины ленты нет явных ограничений. И вот у нас появляется возможность писать программы, генерирующие неограниченный объем вывода. Она не только имеет полезные практические последствия (TOY — крошечная машина, но и она может производить серьезные вычисления и генерировать большой объем вывода), но она также имеет значительные последствия для теории вычислений, как было показано в главе 5.

Стандартный ввод

Конечно, параллельно с перфораторами развивались устройства для *ввода* данных с бумажной ленты. На одной стороне ленты располагается источник света, на другой — 16 фотоприемников; два ряда отверстий на ленте считываются как двоичное слово. При этом единичные биты соответствуют отверстиям, а нулевые — позициям без отверстия. И снова зубцы колеса, входящие в маленькие отверстия посередине ленты, перемещают ее в позицию готовности к чтению следующего слова.

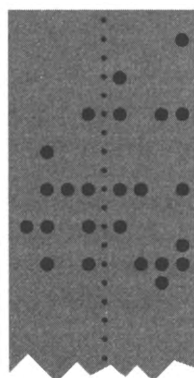
Вероятно, вы уже догадались, что для чтения слова с ленты снова используется ячейка памяти FF. В этой схеме устройство связывается с компьютером таким образом, что каждый раз, когда программа *загружает* слово из ячейки, устрой-

[1] Такую ячейку называют *портом ввода-вывода*. Если порты не включены в общее адресное пространство основной памяти (например, в архитектуре Intel x86!), то для доступа к ним служат специальные *инструкции ввода-вывода*. — *Примеч. науч. ред.*

Листинг 6.3.3. Стандартный ввод: вычисление суммы

```
50: 7800  R[8] <- 0000            int sum = 0
51: 8CFF  R[C] из станд.ввода     while ((c = read()) != 0)
52: CC55  if (R[C] == 0) PC <- 55 {
53: 188C  R[8] <- R[8] + R[C]         sum = sum + c
54: C051  PC <- 51                }
55: 98FF  R[8] в станд.вывод      print(sum)
56: 0000  останов
```

В начале работы счетчик PC содержит значение 50. Программа читает последовательность чисел из стандартного ввода, вычисляет их сумму и записывает результат в стандартный вывод. В программе соблюдается соглашение о том, что `0000` *на ленте является признаком конца последовательности.*



$0001_{16} = 1_{10}$

$0008_{16} = 8_{10}$

$001B_{16} = 27_{10}$

$0040_{16} = 64_{10}$

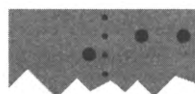
$007D_{16} = 125_{10}$

$00D8_{16} = 216_{10}$

$0157_{16} = 343_{10}$

$0200_{16} = 512_{10}$

входная лента



$0510 = 1296_{10}$

выходная лента

ство чтения активизируется, читает 16 битов с ленты и загружает их в заданный регистр.

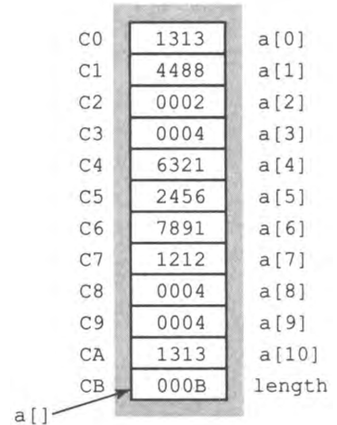
Со стандартным вводом обработка данных на машине TOY реализуется достаточно просто, как видно из листинга 6.3.3. Всего семью инструкциями TOY мы вычисляем сумму чисел на входной ленте. Приведенный пример подтверждает, что:

```
1 + 8 + 27 + 64 + 125 + 216 + 343 + 512 = 1296
```

Конечно, эта программа обобщается для любых вычислений с входными данными, и в прошлом компьютеры интенсивно использовались таким образом. Это была вполне типичная ситуация: в машину загружается небольшая программа, после чего всю работу выполняет оператор: весь день напролет он просто устанавливает ленты, запускает программу и собирает вывод на выходную ленту. Ценность эффективной обработки данных сомнений не вызывает. Еще раз подчеркнем, что объем входных данных неограничен — и в этой концепции отражены глубокие принципы теории вычислений.

Массивы

Из возможности чтения значительных объемов данных непосредственно следует необходимость их хранения в памяти для обработки. И конечно, здесь мы приходим к своей первой структуре данных в TOY — *массиву*. Мы воспользуемся естественным представлением массива, очень похожим на то, которое было описано для Java: элементы массива хранятся в непрерывной последовательности слов памяти, но длина хранится в конце массива, а не в начале, как показано справа. При обращении к массиву указывается адрес этого слова. Эта схема работает ничуть не хуже, чем схема с размещением длины в начале: с ней сохраняется важнейшая характеристика массива — чтобы вычислить адрес `a[i]`, достаточно прибавить `i` к адресу `a[0]`, а адрес `a[0]` легко вычисляется по адресу массива (вычитанием длины).



Стандартное для машины TOY представление массива

Одной из главных задач инструкций с косвенной адресацией (для машины TOY это косвенная загрузка и сохранение) является поддержка операций с массивами. Адрес `a[i]` записывается в регистр. Чтобы загрузить/сохранить значение `i`-го слова массива, вы используете инструкцию косвенной загрузки/сохранения с указанием этого регистра.

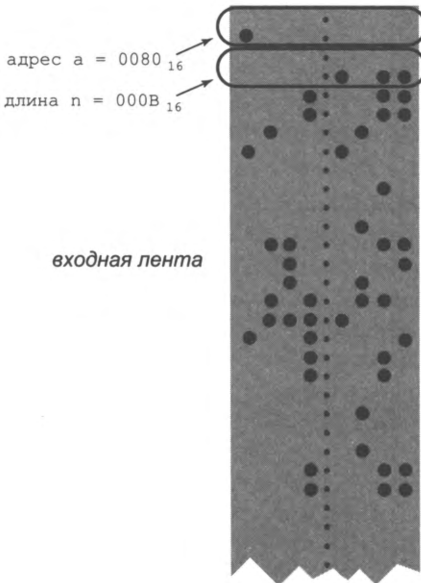
Листинг 6.3.4 демонстрирует процесс загрузки содержимого перфоленты в массив. Код оформлен в функцию, которая читает количество слов в массиве, и адрес для сохранения данных с ленты. Затем программа входит в простой цикл, который читает каждое слово, использует индекс `i` для указания следующего элемента массива и сохраняет результат в `a[i]`. После того как все данные будут прочитаны и сохранены в памяти, значение `i` станет равно длине массива — инструкция перед инструкцией останова сохраняет длину в конце массива. Возвращаемое значение в `R[A]` содержит *адрес* этого значения, находящегося в конце массива (стандартный формат TOY).

После того как данные будут загружены в массив TOY, код обработки массива может обратиться к `i`-му элементу массива точно так же, как в инструкциях `66` и `68` листинга 6.3.4: нужно прибавить индекс `i` к адресу `a[0]`, а затем воспользоваться инструкцией косвенной загрузки или сохранения элемента. Альтернативное решение — сохранить указатель на `a[i]`, увеличить индекс для перехода к следующему элементу, а затем снова воспользоваться косвенной инструкцией для обращения к элементам массива. В таблице на с. 910 приведены два примера кода, демонстрирующие пользу от передачи массивов в аргументах функций. Первый пример в таблице — типичная программа обработки массива, находящая максимальное значение в массиве, адрес которого хранится в `R[A]`. Программа

Листинг 6.3.4. Обработка массива: чтение

```
60: 8AFF R[A] из станд. ввода              a = адрес a[0]
61: 8BFF R[B] из станд. ввода              n = read()
62: 7101 R[1] <- 0001
63: 7900 R[9] <- 0000                      i = 0
64: 22B9 R[2] <- R[B] - R[9]               while (i < n)
65: C26B if (R2 == 0) PC <- 6B             {
66: 1CA9 R[C] <- R[A] + R[9]
67: 8DFF R[D] из станд. ввода
68: BD0C M[R[C]] <- R[D]                      a[i] = read()
69: 1991 R[9] <- R[9] + 1                     i = i + 1
6A: C064 PC <- 64                          }
6B: 1AA9 R[A] <- R[A] + R[9]               адрес a[] (стандарт TOY)
6C: BB0A M[R[A]] <- R[B]                   a.Length = n
6D: EF00 PC <- R[F]                        вернуть управление
```

Начиная со значения PC, равного 60, программа читает массив с ленты и сохраняет его в формате, стандартном для TOY. Конкретно она читает адрес a и целое число n из стандартного ввода, читает n целых чисел из стандартного ввода и сохраняет их в ячейках памяти M[a], M[a+1], ..., M[a+n−1]. Затем программа сохраняет n в M[a+n] (и возвращает a+n в R[a]).



80	1313	
81	4488	
82	0002	
83	0004	
84	6321	
85	2456	R[A]008B
86	7891	
87	1212	
88	0004	
89	0004	
8A	1313	
8B	000B	

результат FF60

последовательно перебирает в R[A] индексы массива, используя инструкцию косвенной загрузки для чтения каждого элемента массива и сравнения его с наибольшим из значений, обнаруженных до настоящего времени; при необходимости текущий максимум обновляется. Второй пример — клиент, который вводит массив

с перфоленты, находит максимальный элемент в массиве и выводит его на ленту. Построить набор функций для выполнения разнообразных операций с массивами не так уж сложно.

Одной из причин для разработки таких носителей, как перфолента, стала необходимость сохранения данных, полученных от устройств, занимавшихся экспериментальными измерениями разного рода. Идея *обработки данных* — сохранения данных на физических носителях (таких, как перфокарты или перфоленты) и их последующей обработки некоторой машиной — на несколько десятилетий опередила разработку компьютеров. Перфокарты применялись в учреждениях для хранения клиентских данных еще с 1900-х годов, а машина для сортировки перфокарт (с некоторым ручным вмешательством) была построена в 1901 году! Более того, одна из самых успешных компьютерных компаний современности разработала этих предшественников компьютеров за полвека до выпуска своих первых компьютеров в 1950-е годы — мы говорим о компании *International Business Machines*, которая сейчас известна под названием IBM.

Не нужно обладать выдающимся воображением, чтобы понять, что возможность выполнения таких вычислений (даже на устройствах, не особо превосходивших машину TOY по сложности) окажет громадное практическое влияние в мире, в котором люди пользуются логарифмическими линейками и калькуляторами.

функция для нахождения максимума

```
               R[A] <- адрес массива      (стандарт TOY)
70:  7101  R[1] <- 0001
71:  A90A  R[9] <- M[R[A]]               int i = a.length
72:  140A  R[4] <- R[A]
73:  7B00  R[B] <- 0                     int max = 0
74:  C97C  if (R[9] == 0) PC <- 7C       while (i > 0)
75:  2991  R[9] <- R[9] - 1              {  i = i - 1
76:  2441  R[4] <- R[4] - 1                 адрес a[i]
77:  AC04  R[C] <- M[R4]                    d = a[i]
78:  2EBC  R[E] <- R[B] - R[C]
79:  DE7B  if (R[E] > 0) PC <- 7B           if (d > max)
7A:  1B0C  R[B] <- R[C]                        max = d
7B:  C074  PC <- 74                      }
7C:  EF00  PC <- R[F]                    вернуть (max в R[B])
```

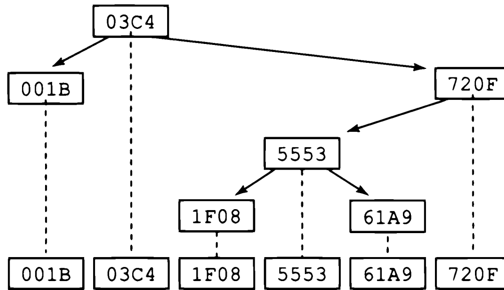
прочитать массив и вывести максимум (с использованием листинга 5.3.5 и приведенной выше функции)

```
5C:  FF60  R[F] <- 5D; PC <- 60          прочитать a[] из стандартного
                                         ввода
5D:  FF70  R[F] <- 5E; PC <- 70          R[B] =  max(a[])
5E:  9BFF  R[B] в станд.вывод            записать результат
5F:  0000  останов
```

Типичный код обработки массива

Связные структуры

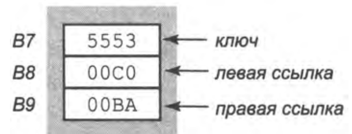
Другие структуры данных, которые мы рассматривали, тоже без особых сложностей реализуются на машине TOY. Для примера будет рассмотрено *бинарное дерево поиска* (см. листинг 4.4.3). Для простоты возьмем бинарное дерево поиска с целочисленными ключами вроде того, что изображено ниже.



Бинарное дерево поиска, представляющее множество целых чисел

Допустим, вам нужна программа для *устранения дубликатов* с перфоленты, полученной на входе, — другими словами, программа должна записывать на ленту данные, из которых исключены все повторяющиеся значения. Как решается эта задача? Программа читает новое значение, проводит *поиск* по бинарному дереву поиска, чтобы проверить, присутствует ли в нем прочитанное значение (чтобы его можно было проигнорировать), и если значение еще не встречалось прежде, то вставляет его в дерево и записывает в стандартный вывод. В листинге 6.3.5 приведена реализация функции TOY, которая может использоваться для этой цели. Сначала мы поподробнее рассмотрим функцию, а потом перейдем к клиенту.

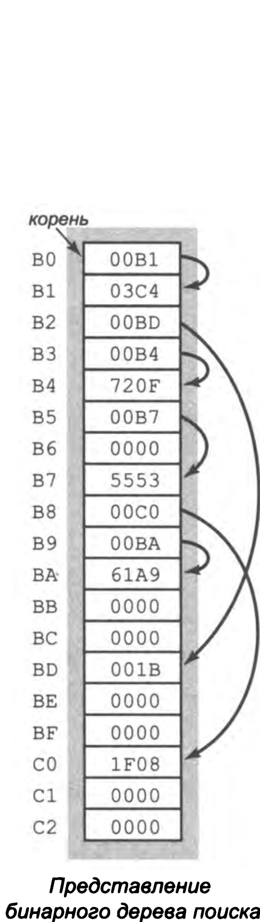
Для представления узла бинарного дерева поиска будут использоваться три слова TOY: одно для ключа, другое для левой ссылки и третье для правой ссылки (`0000` обозначает `null`-ссылки). Обычно узлы хранятся в смежных участках памяти вплотную друг к другу, хотя возможны и более общие схемы размещения.



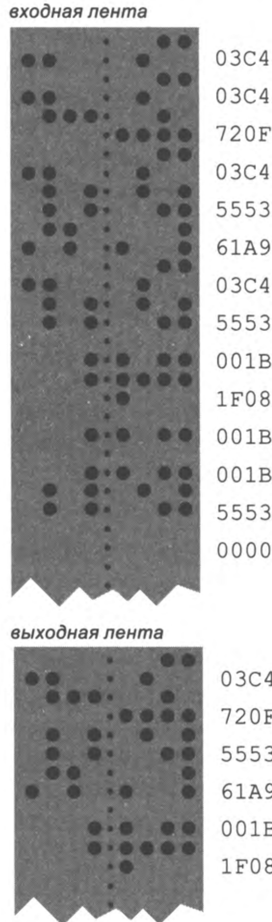
Представление узла бинарного дерева поиска

Для представления дерева используется последовательность узлов, занимающих непрерывную область памяти (см. с. 912, слева). Каждый раз, когда в дереве создается новый узел, он добавляется в конец последовательности.

Функция поиска и вставки в бинарном дереве поиска в листинге 6.3.5 получает в параметрах адрес корня дерева в `R[A]`, ключ для поиска в `R[B]` и адрес следующей ячейки для размещения нового узла в `R[E]`. Если в дереве нет ни одного узла, программа просто создает новый узел, как описано в следующем абзаце. Если дерево



Представление бинарного дерева поиска



равна `0000`, мы можем использовать инструкцию `BE08` для замены `null`-ссылки, на которой завершился поиск, ссылкой на новый узел. Чтобы создать новый узел, мы сохраняем новый ключ и две ссылки `0000` и обновляем `R[E]` (код в инструкциях по адресам `90`–`95`), после чего возвращаем управление с `R[9]` = 0.

Листинг 6.3.5 фактически представляет собой полноценную реализацию символической таблицы, которая быстро и эффективно работает в реальных ситуациях. Во многих отношениях эта реализация проще и понятнее Java-версии. По крайней мере, она поможет вам лучше понять суть реализации связных структур.

С функцией из листинге 6.3.5 реализация клиента, устраняющего дубликаты с перфоленты, требует менее 10 инструкций TOY (см. с. 914). Когда вы впервые прочитали о TOY в начале этой главы, возможно, вы и не представляли, что машина обладает мощью, необходимой для вычислений такого рода. И конечно, это только начало!

Листинг 6.3.5. Связные структуры: поиск/вставка в бинарном дереве поиска

```
80:  7101  R[1] <- 0001
81:  A90A  R[9] <- root                 x = корень
82:  180A  R[8] <- R[A]                 сохранить адрес ссылки
83:  C98F  if (R[9] == 0) PC <- 8F      while (x != 0) {
84:  AC09  R[C] <- M[R[9]]                 t = x.key
85:  2CBC  R[C] <- R[B] - R[C]             if (t == key)
86:  CC96  if (R[C] == 0) PC <- 96              вернуть управление
87:  1991  R[9] <- R[9] + 1
88:  1809  R[8] <- R[9]                    else if (t > key)
89:  A909  R[9] <- M[R[9]]                    x = x.left
8A:  DC8E  if (R[C] > 0) PC <- 8E          else
8B:  1981  R[9] <- R[8] + 1                   x = x.right
8C:  1809  R[8] <- R[9]
8D:  A909  R[9] <- M[R[9]]
8E:  C083  PC <- 83                     }
8F:  BE08  M[R[8]] <- R[E]              присвоить ссылку на новый узел
90:  BB0E  M[R[E]] <- key
91:  1EE1  R[E] <- R[E] + 1
92:  900E  M[R[E]] <- 0
93:  1EE1  R[E] <- R[E] + 1
94:  900E  M[RE] <- 0                    new node(key, 0, 0)
95:  1EE1  R[E] <- R[E] + 1
96:  EF00  PC <- R[F]                    вернуть управление
```

Будучи вызвана инструкцией `FF80` *(которая поместит в счетчик PC значение* `80`*), программа ищет в дереве с корнем R[A] ключ, хранящийся в R[B], выделяя память для новых узлов с адреса R[E]. Возвращаемое значение в R[9] отлично от 0 при успешном поиске или равно 0, если поиск не принес успеха (и узел был добавлен).*

R9	RB	RC
00B1	1F08	03C4
00B4	1F08	720F
00B7	1F08	5553
00C0	1F08	1F08

трассировка для PC = 84
поиск `1F08`

не пусто, то ключ сравнивается с ключом текущего узла (с использованием инструкции косвенной загрузки для загрузки ключа), и функция возвращает ненулевое значение в `R[9]` в случае равенства (успешный поиск). В противном случае происходит переход по левой или правой ссылке (снова с использованием инструкции косвенной загрузки), и программа выполняется в цикле — пока не найдет ключ или не достигнет ссылки `0000`.

Функция *вставки* выполняется в том случае, если поиск показал, что ключ в дереве отсутствует. Это означает, что новый узел должен быть вставлен в дерево в точке завершения поиска. Хитрость в решении этой задачи заключается в сохранении адреса последней ссылки (в `R[8]` в инструкциях по адресам `82`, `88` и `8C`). Если ссылка

устранение дубликатов с перфоленты

```
A0:  7AB0  R[A] <- B0               корень
A1:  7EB1  R[E] <- B1               свободное пространство
A2:  8BFF  R[B] - из станд.ввода    while ((b = read()) != 0)
A3:  CBA8  if (R[B] == 0) PC <- A8  {
A4:  FF80  R[F] <- PC; PC <- 80        x = searchInsert(b)
A5:  C9A2  if (R[9] == 0) PC <- A2    if (x == 0) continue
A6:  9BFF  R[B] в станд. вывод        print(b)
A7:  C0A2  PC <- A2                 }
A8:  0000  halt
```

Типичный клиент символической таблицы

Зачем изучать программирование на машинном языке?

К этому вопросу стоит вернуться сейчас — когда вы лучше разбираетесь в ситуации, но еще не перешли к сложностям самостоятельного написания программ на машинном языке. Основных причин три.

- Программы на машинном языке остаются предпочтительными во многих практических ситуациях.
- Анализ кода на столь низком уровне помогает вскрыть его суть.
- Вы лучше поймете работу программ (например, компиляторов), генерирующих программы на машинном языке вашего компьютера.

Ниже кратко объясняются все эти причины, а также подводится итог обсуждения, распределенного по этой главе.

Во-первых, программы на машинном языке остаются предпочтительными в ситуациях, критичных по быстродействию. Ученые, инженеры и прикладные программисты продолжают испытывать на прочность границы возможного, и очень часто низкоуровневая реализация критической части вычислений работает на один-два порядка быстрее кода, написанного на языках высокого уровня. Вы должны по крайней мере знать об этой возможности.

Во-вторых, код на машинном языке часто отражает сущность вычислений, так как он может отсечь системные и машинные зависимости и показать, что же происходит на самом деле. В частности, структуры данных на машинном языке часто оказываются намного более прозрачными, чем в языках высокого уровня. Только что рассмотренный пример с бинарными деревьями поиска хорошо демонстрирует это явление; другие примеры встретятся вам в упражнениях.

В-третьих, зная машинный язык, вы можете лучше представить концепцию написания программ на языках высокого уровня (таких, как Java), генерирующих программы на машинном языке. Используя свой компьютер, вы зависите от таких программ, потому что весь код, выполняемый на вашем компьютере, в конечном итоге сводится к машинному языку. Этот аспект программирования более подробно

рассматривается в следующем разделе. Написание программ, генерирующих другие программы, — интересное и достойное занятие, которое заслуживает внимания каждого программиста.

В контексте книги мы стремимся снять завесу тайны с того, что происходит внутри вашего компьютера; для этого мы описываем природу интерфейса между аппаратной и программной частью. Надеемся, что вы будете рассматривать программирование для TOY как возможность оценить эти важные идеи без привлечения более сложных реальных компьютеров. Это пример применения концепции программирования на машинном языке, на котором мы сможем продемонстрировать ряд фундаментальных идей. Также изучение TOY можно рассматривать как подготовительный этап для освоения реальных компьютеров.

В любом случае важность программирования на машинном языке в этой книге (а на самом деле и в развитии вычислительной инфраструктуры за несколько последних десятилетий) заключается в том, что оно составляет промежуточный уровень абстракции между вашими программами на языке Java и компьютером. Во-первых, вы сейчас в состоянии лучше понять связь между программами Java, которые пишете вы, и программами на машинном языке, которые выполняются на вашем компьютере. Следующий раздел посвящен более подробному анализу этой связи. Во-вторых, вы сейчас можете лучше понять потенциал создания физического устройства, способного выполнять программы на машинном языке. В главе 7 наша работа по демистификации завершится изучением процесса проектирования и построения электронных схем, способных выполнять программы на машинном языке. И снова хорошее знание низкоуровневых машин (таких, как TOY) станет важнейшим фактором для понимания того, как работают такие схемы.

Вопросы и ответы

В. Мне действительно необходимо практиковаться в написании программ TOY?

О. Пожалуй, доводить свои навыки до такой же степени, как в программировании на Java, не обязательно. Тем не менее TOY можно рассматривать как еще один язык программирования. Конечно, знание Java упростит изучение TOY[1] — каждый новый язык, который вы изучаете, упрощает изучение следующего языка.

В. Где я смогу найти дополнительную информацию о TOY?

О. На сайте книги доступна достаточно обширная информация, разработанная студентами и преподавателями за два десятилетия, в течение которых мы работали над книгой. В частности, вы найдете на сайте интерактивную модель TOY и сможете попрактиковаться во вводе программ на переключателях и индикаторах. (Мы используем ее для учебных демонстраций.)

В. Есть еще какие-нибудь источники информации о TOY?

[1] И наоборот — знание языка TOY может помочь в изучении Java. — *Примеч. науч. ред.*

О. Нет, никаких. Это воображаемая машина.

В. Так, может, мне стоит изучить другой машинный язык?

О. Конечно, вы можете изучать программирование на примере реального языка. Как уже было сказано, знание TOY хорошо подготовит вас к этому. Вы можете учиться на примере популярной архитектуры IA-32, но учтите, что полное руководство разработчика занимает тысячи страниц![1]

О. Также вы можете изучить MIX — еще один воображаемый машинный язык, разработанный Дональдом Кнутом для классической серии книг «Искусство программирования». Его доводы в пользу именно машинного языка сформулированы в предисловии к этим книгам; одна из них заключается в следующем: «Человек, сколько-нибудь серьезно интересующийся компьютерами, должен хорошо владеть машинным языком, потому что он является фундаментальной частью компьютера». А после изучения MIX станет возможным ознакомиться со всеми многочисленными алгоритмами, которые в книгах Кнута представлены только на этом языке.

Упражнения

Важно: вам будет намного проще справиться с этими упражнениями, если у вас в распоряжении имеется машина TOY для запуска и отладки программ. Прежде чем браться за упражнения, мы рекомендуем прочитать раздел 6.4.

6.3.1. Напишите программу sort3.toy, которая читает три целых числа из стандартного ввода и выводит их в стандартный вывод на перфоленту по возрастанию.

6.3.2. Напишите программу powers2.toy, которая выводит в стандартный вывод все положительные степени 2, которые могут быть представлены машинным словом TOY в дополнительном коде.

6.3.3. Напишите программу sum_1-n.toy, которая читает целое число n из стандартного ввода и выводит на ленту сумму арифметической прогрессии $1 + 2 + 3 + ... + n$.

6.3.4. Для заданного целого x следующее целое число в последовательности Коллаца вычисляется по формуле x/2, если x четное, или 3x+1, если x нечетное. Элементы последовательности генерируются до тех пор, пока значение x не станет равным 1 (см. упражнение 2.3.29). Напишите программу collatz.toy, которая читает целое число из стандартного ввода и выводит начинающуюся с него последовательность Коллаца в стандартный вывод. *Подсказка*: используйте инструкцию сдвига вправо для выполнения целочисленного деления на 2.

6.3.5. Изобразите реализацию цикла `for` на машине TOY (аналогично приведенным в тексте `if` и `while`).

[1] Или на примере какого-либо 8-разрядного микроконтроллера, который гораздо проще, чем IA-32 (но все равно заметно сложнее, чем TOY). — *Примеч. науч. ред.*

6.3.6. Напишите программу chop.toy, которая читает целое число n из стандартного ввода и выводит на ленту степени 2, сумма которых равна n. Например, если n равно 012A, то программа должна вывести на выходную ленту

```
0002
0008
0020
0100
```

потому что 012A = 0002 + 0008 + 0020 + 0100.

6.3.7. Напишите программу, которая читает целое число из стандартного ввода, возводит его в куб и выводит результат. Чтобы выполнить операцию умножения, используйте инструкцию FF90 для вызова функции умножения из упражнения 6.3.35.

6.3.8. Напишите фрагмент кода TOY, который меняет местами содержимое R[A] и R[B] без записи в основную память или в другие регистры. *Подсказка*: воспользуйтесь операцией XOR.

6.3.9. Этот вопрос проверяет, насколько хорошо вы понимаете различия между загрузкой адреса, загрузкой из памяти и косвенной загрузкой. Для каждой из следующих программ TOY приведите содержимое R[1], R[2] и R[3] после завершения программы.

```
(a)                 (b)                 (c)
10: 7211            10: 8211            10: 7211
11: 7110            11: 8110            11: A102
12: 2321            12: 2312            12: 2312
13: 0000            13: 0000            13: 0000
```

6.3.10. Рассмотрим следующую программу TOY. Какое значение будет содержать R[3] после останова?

```
10: 7101
11: 7207
12: 7301
13: 1333
14: 2221
15: D213
16: 0000
```

6.3.11. Для каждого из следующих логических выражений приведите фрагмент кода TOY, который читает целое число из стандартного ввода и записывает в стандартный вывод 0001, если условие истинно, и 0000, если оно ложно.

```
 a = 3
 a > 3
 a < 3
a != 3
a >= 3
a <= 3
```

6.3.12. Предположим, вы загрузили следующую программу в ячейки 10-17 машины TOY, занесли в PC значение 10 и нажали кнопку RUN.

```
10: 7100   R[1] <- 0000
11: 8FFF   R[F] из станд.ввода
12: 9F15   M[15] <- R[F]
13: 82FF   R[2] из станд.ввода
14: 1112   R[1] = R[1] + R[2]
15: C016   PC <- 16
16: 91FF   R[1] в станд.вывод
17: 0000   останов
```

Что будет выведено (если будет) в стандартный вывод, если стандартный ввод содержит 1112 1112?

Подсказка: первое значение сохраняется в ячейке M[15], где оно в конечном итоге будет выполнено как код.

6.3.13. Повторите предыдущее упражнение, но на этот раз со следующими данными в стандартном вводе: C011 C011 1112 1112.

6.3.14. Напишите функцию TOY, которая получает аргументы a, b и c в R[A], R[B] и R[C], вычисляет дискриминант $d = b^2 - 4ac$ и возвращает результат в R[D]. Поместите свой код в ячейки 10-... и используйте инструкцию FF90 для вызова функции умножения из упражнения 6.3.35.

6.3.15. Перечислите все входные значения от 0123 до 3210, для которых следующая программа записывает 0000 в стандартный вывод перед остановкой.

```
10: 8AFF   R[A] из станд.ввода
11: 7101   R[1] <- 0001
12: 2BA1   R[B] <- R[A] - 1
13: 3CAB   R[C] <- R[A] & R[B]
14: 9CFF   R[C] в станд.вывод
15: 0000   останов
```

Решение. 0200 0400 0800 1000 2000. 1 возвращается для всех вариантов ввода, содержащих не более одной 1 в двоичном представлении, то есть для шестнадцатеричных целых чисел 0000, 0001, 0002, 0004, 0008, 0010, ..., 8000.

6.3.16. Предположим, в ячейки 10–20 машины TOY загружена следующая программа:

```
10: 7101
11: 7A30
12: 7B08
13: 130B
14: C320
15: 1400
16: 2543
17: C51E
```

```
18: 16A4
19: A706
1A: 1717
1B: B706
1C: 1414
1D: C016
1E: 6331
1F: C014
20: 0000
```

Предположим, вы загрузили `0001 0002 0003 0004 0004 0003 0002 0001` в ячейки памяти с `30` по `37`, записали в PC значение `10` и нажали RUN. Каким будет содержимое ячеек памяти с `30` по `37` после завершения программы?

6.3.17. Преобразуйте программу TOY из предыдущего упражнения в код Java, заполнив пропуски ????.

```
for (int i = n;  i > ???? ;  i = i ????)
    for (int j = 0;  j < ???? ;  j = j ????)
        a[ ???? ] = ???? ;
```

6.3.18. Напишите программу, которая вычисляет скалярное произведение двух векторов, хранящихся в массивах TOY. Оформите ее в виде функции, которая получает адреса массивов в `R[A]` и `R[B]`, и возвращает скалярное произведение в `R[E]`. Используйте инструкцию `FF90` для вызова функции умножения из упражнения 6.3.35. *Примечание*: вам понадобится более общая схема вызова функции, чем те, которые рассматривались выше. Сохраните и восстановите адрес возврата `R[F]`, чтобы один регистр мог использоваться для адреса возврата при вызове функции умножения. См. упражнение 6.3.27.

6.3.19. Допустим, вы загрузили следующую программу в ячейки `10-1B` машины TOY, а стандартный ввод содержит значения `1CAB EF00 0000 4321 1234`. Какое значение будет направлено в стандартный вывод, если записать в PC значение `10` и нажать кнопку RUN?

```
10: 7101   R[1] <- 0001
11: 7230   R[2] <- 0030
12: 8AFF   R[A] из станд.ввода
13: CA17   if (R[A] == 0) PC <- 17
14: BA02   M[R[2]] <- R[A]
15: 1221   R[2] <- R[2] + 1
16: C012   PC <- 12
17: 8AFF   R[A] из станд.ввода
18: 8BFF   R[B] из станд.ввода
19: FF30   см. предыдущее упражнение
1A: 9CFF   R[C] в станд.вывод
1B: 0000   останов
```

6.3.20. Ответьте на предыдущий вопрос для случая, в котором стандартный ввод содержит значения `2CAB EF00 0000 4321 1234`.

6.3.21. Рассмотрим следующий код для обхода связного списка:

```
10: 7101
11: 72D0
12: 1421
13: A302
14: 93FF
15: A204
16: D212
17: 0000
```

Каждый узел состоит из двух последовательных слов в памяти: значения и ссылки (адреса следующего узла), ссылка со значением `0000` отмечает конец списка. Предположим, ячейки памяти `D0-DB` содержат значения

```
0001 00D6 0000 0000 0004 0000 0002 00DA 0000 0000 0003 00D4
```

Приведите значения, выводимые этой программой (запишите в PC значение `10` и нажмите кнопку RUN).

Решение. `1 2 3 4`.

6.3.22. Укажите, как изменить одно слово памяти в предыдущем упражнении, чтобы оно выводило `1 2 6 7` вместо `1 2 3 4 5 6 7` (удаление из связного списка).

6.3.23. Укажите, как изменить три слова памяти (заменить одно и использовать еще два), чтобы программа из упражнения 6.3.21 выводила `1 2 3 4 8 5 6 7` (вставка в связный список).

6.3.24. Допустим, память TOY содержит следующие значения (см. таблицу ниже), вы записали в счетчик адреса программы значение `30` и нажали кнопку RUN. Что будет выведено в стандартный вывод (и будет ли)? Приведите содержимое `R[2]` и `R[3]` после остановки машины.

```
    2_   3_   4_   5_   6_
_0 0000 7101 7101 0003 0002
_1 0000 7200 7200 0000 0050
_2 0000 8329 8329 0005 0000
_3 0000 1221 A403 0000 0000
_4 0000 1331 1224 0004 0000
_5 0000 A303 1331 0052 0000
_6 0000 D333 A303 0000 0000
_7 0000 92FF D343 0000 0000
_8 0000 0000 0000 0001 0000
_9 005A 0000 0000 0060 0000
_A 0000 0000 0000 0000 0000
_B 0000 0000 0000 0058 0000
_C 0000 0000 0000 0000 0000
_D 0000 0000 0000 0000 0000
_E 0000 0000 0000 0000 0000
_F 0000 0000 0000 0000 0000
```

6.3.25. Разработайте пару функций TOY, реализующих стек. Значения, заносимые и извлекаемые из стека, передаются в `R[B]`, а адрес стека в `R[A]`.

6.3.26. Напишите функцию TOY для обхода бинарного дерева поиска и вывода ключей в порядке сортировки. *Подсказка*: используйте стек.

6.3.27. На основе своего решения упражнения 6.3.25 разработайте две функции TOY: одна сохраняет регистры от `R[A]` до `R[F]` в стеке, а другая восстанавливает их из стека. Используйте функции для разработки рекурсивной программы, выводящей размеры делений на линейке (см. листинг 1.2.1).

6.3.28. Реализуйте версию функции бинарного дерева поиска (листинг 6.3.5), которая экономит память за счет размещения двух ссылок каждого узла в одном машинном слове.

Упражнения повышенной сложности

6.3.29. *32-разрядные целые числа.* Напишите функцию TOY, которая получает в `R[A]` и `R[B]` 32-разрядное целое число в дополнительном коде, а в регистрах `R[C]` и `R[D]` — второе 32-разрядное целое число в дополнительном коде, суммирует два числа и оставляет результат в `R[A]` и `R[B]`.

6.3.30. *Код Грея.* Напишите программу graycode.toy, которая читает из стандартного ввода целое число n (от 1 до 15), а затем выводит в стандартный вывод `(i >> 1) ^ i` для значений `i`, уменьшающихся от $2^n - 1$ до 0. Полученная последовательность называется *кодом Грея* порядка n (см. обсуждение, связанное с листингом 2.3.3).

6.3.31. *Скалярное произведение.* Вычислите скалярное произведение двух массивов, которые начинаются с адресов `R[A]` и `R[B]` и имеют длину `R[C]`.

6.3.32. *Axpy (Alpha X Plus Y).* Напишите функцию TOY, которая получает скалярное значение a в `R[A]`, вектор b в массиве TOY, адрес которого хранится в `R[B]`, и еще один вектор c, адрес которого хранится в `R[C]`; вычисляет вектор $ab + c$ и оставляет результат в массиве TOY, адрес которого хранится в `R[D]`.

6.3.33. *Одноразовый ключ.* Реализуйте на машине TOY генерирование одноразового ключа для шифрования и дешифрования 256-разрядных сообщений. Предполагается, что ключ хранится в ячейках памяти `30-3F`, а ввод состоит из шестнадцати 16-разрядных целых чисел.

6.3.34. *Поиск одиночного числа.* Допустим, в стандартном вводе находится последовательность из $2n + 1$ 16-разрядных целых чисел, в которой каждое из n чисел встречается ровно дважды, а одно число встречается только один раз. Напишите программу TOY, которая выводит это одиночное число. *Подсказка*: объедините все целые числа операцией XOR.

6.3.35. *Эффективное умножение.* Реализуйте алгоритм умножения двух целых чисел, который вы изучали в школе. Конкретнее, если b_i — i-й бит b, то

$$b = (b_{15} \times 2^{15}) + (b_{14} \times 2^{14}) + \ldots + (b_1 \times 2^1) + (b_0 \times 2^0).$$

Далее для вычисления $a \times b$ используется свойство дистрибутивности:

$$a \times b = (a \times b_{15} \times 2^{15}) + (a \times b_{14} \times 2^{14}) + \ldots + (a \times b_1 \times 2^1) + (a \times b_0 \times 2^0)$$

На первый взгляд задача выполнения одного умножения сводится к 32 умножениям, по 2 для каждого из 16 слагаемых. К счастью, каждое из этих 32 умножений весьма специфично, потому что $a \times 2^i$ эквивалентно сдвигу влево на i разрядов. Так как b_i равно либо 0, либо 1, i-е слагаемое равно либо `a << i`, либо 0.

Решение.

```
90: 7101   R[1] <- 0001
91: 7C00   R[C] <- 0000               c = 0
92: 7210   R[2] <- 0010               for (i = 16; i > 0; i--)
93: 2221   R[2] <- R[2] - 1           {
94: 53A2   R[3] <- R[A] << R[2]          t = a * 2^i
95: 64B2   R[4] <- R[B] >> R[2]
96: 3441   R[4] <- R[4] & 1
97: C41B   if (R[4] == 0) PC <- 99       if b[i] == 1
98: 1CC3   R[C] <- R[C] + R[3]              c += t
99: D293   if (R[2] == 0) PC <- 93    }
9A: FF00   вернуть управление
```

6.3.36. *Функция возведения в степень.* На основе функции умножения из предыдущего упражнения реализуйте функцию TOY для вычисления a^b, получающую аргументы a и b в `R[A]` и `R[B]`.

6.3.37. *Вычисление полинома.* Используя функции возведения в степень и умножения из двух предыдущих упражнений, реализуйте функцию TOY, которая получает адрес массива TOY с коэффициентами $a_0, a_1, a_2, \ldots, a_n$ и целое число x в `R[A]` и `R[B]` и вычисляет полином

$$p(x) = a_n x^n + \ldots + a_2 x^2 + a_1 x^1 + a_0 x^0$$

возвращая полученное значение в `R[C]`. Вычисление полиномов было одной из важнейших причин для появления первых компьютеров (для построения баллистических таблиц).

6.3.38. *Схема Горнера* — усовершенствованная альтернатива прямому вычислению полинома, более эффективная и проще программируемая. Основная идея заключается в рационально спланированном порядке умножения как в следующем примере:

$$p_4 x^4 + p_3 x^3 + p_2 x^2 + p_1 x^1 + p_0 x^0 = ((((p_4)x + p_3)x + p_2)x + p_1)x + p_0$$

Реализуйте на основе этой идеи функцию TOY, которая использует только n умножений для вычисления полинома степени n. Этот метод был опубликован в XIX веке британским математиком Уильямом Горнером (William Horner), но использовался он еще Исааком Ньютоном почти веком ранее (см. упражнение 2.1.31).

6.3.39. *Преобразование чисел.* Реализуйте программу TOY, использующую схему Горнера (см. листинг 6.1.1) для преобразования десятичного целого числа в двоичное представление. Прочитайте десятичные числа в шестнадцатеричной записи в форме `000x` из стандартного ввода и направьте двоичный результат в стандартный вывод по одному биту.

6.4. Виртуальная машина TOY

Если TOY — воображаемая машина, то как же запускать и отлаживать программы для нее? В этом разделе мы дадим исчерпывающий ответ на этот вопрос, а затем обсудим возможные следствия. В двух словах ответ заключается в том, что мы можем легко написать *виртуальную машину* — программу Java, которая может выполнить любую программу TOY. В самом деле, как вы увидите, виртуальная машина является определением TOY. Она точно описывает эффект каждой возможной инструкции TOY, при этом сохраняя полную информацию о состоянии машины TOY.

Виртуальная машина представляет собой определение машины: она выполняет программы, как и реальная машина, но при этом не связана напрямую с каким-либо физическим воплощением. Мы будем использовать этот термин в широком смысле: как для определения машины, так и для любой программы (или устройства), реализующей это определение.

Концепция виртуальной машины на первый взгляд связана с ограничениями. Мы приводим столь подробную информацию об этом конкретном примере для того, чтобы вы полностью поняли идею, потому что она имеет огромную практическую значимость и занимает центральное место в фундаментальных принципах теории вычислений, рассмотренных в главе 5. В самом деле, из тезиса Чёрча — Тьюринга следует, что любая вычислительная среда может выполнить те же вычисления, что и любая другая (при наличии достаточной памяти), так что каждый компьютер может стать виртуальной машиной для любого другого компьютера.

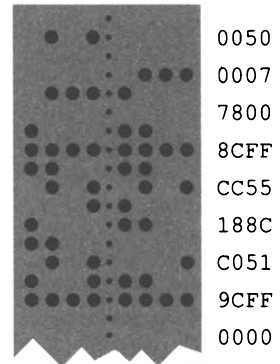
Основополагающая концепция — способность программ обрабатывать другие программы. Вспомните, о чем говорилось в доказательстве проблемы остановки: идея о том, что программа может получать на вход другую программу, на первый взгляд кажется немного странной. Но в действительности эта идея является фундаментальной концепцией информатики, которую мы подробно проанализируем в этом разделе.

Для разминки рассмотрим некоторые практические следствия этой концепции для программирования TOY, включая одну важную и неизбежную ловушку. Они сформируют контекст, необходимый для того, чтобы связать эти идеи с Java и для последующего рассмотрения виртуальной машины TOY — центральной темы этого раздела. Затем будут рассмотрены их следствия для современных и будущих вычислений.

В этом разделе рассматривается широкий круг вопросов, от TOY до последствий проблемы остановки для серверных «ферм» и облачных вычислений. Тот факт, что мы можем проделать все это на нескольких десятках страниц, свидетельствует о мощи, элегантности и непреходящей важности концепций, на которых строятся эти темы. Материал излагается по необходимости кратко, но понимание этих вопросов важно для любого специалиста, работа которого связана с вычислениями.

Начальная загрузка и дамп памяти

Начнем с важнейшей характеристики любой машины с архитектурой фон Неймана (включая TOY) — машина может обрабатывать данные любого типа, а не только числа. Более того, *инструкции одной программы могут быть данными другой программы.* Сейчас нас интересуют два практических следствия этого свойства в контексте программирования TOY.



Входные данные (?)

Начальная загрузка

Возьмем перфоленту, изображенную справа. Она не кажется вам знакомой? Возможно, к этому времени вы уже опознаете некоторые из этих шестнадцатеричных чисел из 4 цифр как инструкции TOY. В самом деле, вы видели их прежде — это последовательность инструкций для листинга 6.3.3 (вычисление суммы чисел на входной ленте), которой предшествует адрес первой инструкции и количество инструкций. Но, как подчеркивалось с самого начала этой главы, смысл заданной последовательности битов на компьютере зависит от контекста. Следовательно, программа может интерпретировать эти биты также и как числа в дополнительном коде. Эту ленту также можно подать на вход программы 6.3.3, чтобы программа вычислила сумму

$$50_{16} + 7_{16} + 7800_{16} + (-7301_{16}) + (-33AB_{16}) + 18CC_{16} + (-3FAF_{16}) + (-6301_{16}).$$

Но гораздо важнее другое: эта лента также может стать входными данными для нашей программы чтения массива из листинга 6.3.4: тогда лента рассматривается как программа, а программа 6.3.4 — как программа, загружающая другие программы в память. Собственно, мы могли бы создать такие ленты для всех программ TOY, которые мы рассматривали, и загрузить их в память аналогичным образом.

Возможен и другой вариант: создать одну ленту для всех ячеек памяти `10-FF` и заполнить всю память кодом и данными. Этот процесс называется *начальной загрузкой* (booting) компьютера — термин из прошлого, который сохранился до наших

дней. Мы используем специальный процесс (в случае TOY — переключатели) для загрузки в память маленькой программы, которая затем заполняет оставшуюся память содержимым, прочитанным с внешнего устройства.

В левой части таблицы на следующей странице приведен код из листинга 6.3.4, преобразованный из функции в код начальной загрузки, который вводится программистом после включения компьютера в начале рабочего дня. Исторически было принято резервировать несколько начальных слов памяти (ячейки `00-0F` в нашем случае) для программы-загрузчика. У такого компьютера, как TOY, распечатка этой программы была бы прикреплена к передней панели. Так как это единственная программа, которую нужно было бы вводить посредством переключателей, программисты гордились бы скоростью, с которой они щелкают переключателями, а их пальцы порхали бы, словно у пианиста (или почти так же).

Кроме того что программа-загрузчик не оформляется в виде функции, она отличается от листинга 6.3.4 в двух отношениях. Первое отличие — не обязательно хранить длину (которая была нужна для массивов). Второе (и намного более принципиальное) отличие заключается в том, что программа-загрузчик завершается инструкцией `EA00`. Так как `R[A]` содержит адрес первого загруженного слова, инструкция передает управление только что загруженным данным — классический пример превращения данных в инструкции в архитектуре фон Неймана.

Современные компьютеры используют тот же базовый процесс, так что основная терминология сохранилась до наших дней. Когда вы перезагружаете свой телефон, планшет или компьютер, операционная система и многие базовые приложения загружаются в устройство с накопителя, внешнего по отношению к процессору, маленькой программой, которая сама загружается в память процессора некоторым специальным образом. Инструкция перехода передает управление этой программе[1].

Дамп памяти

Чтобы получить программу TOY для подготовки ленты, предназначенной для программы-загрузчика, можно просто внести изменения в загрузчик, чтобы он выводил на ленту адрес и длину (вместо их чтения), а затем в цикле выводил каждое слово памяти (вместо чтения их с ленты). В программе в правой части приведенного ниже листинга эти изменения выделены жирным шрифтом. Первые две инструкции задают адрес и длину — программист может присвоить им любые значения на свое усмотрение (при помощи переключателей). Значения `10` (для адреса) и `EF` (для длины) задают «полный дамп»: на ленту записываются только слова из ячеек с `10` по `FE`, а диапазон `00–0F` резервируется для самих программ начальной загрузки и вывода дампа. Также учитывается то, что ячейка `FF` зарезервирована для стан-

[1] Обычно код загрузчика просто записывается заранее в постоянную память (ПЗУ), занимающую заранее оговоренный диапазон адресов. Процессор обычно начинает выполнение инструкций с заранее определенного адреса. — *Примеч. науч. ред.*

дартного ввода и стандартного вывода. Этот процесс называется *сохранением* (или *сбросом*) *дампа* памяти.

```
                                      чтение/запись
02: 8AFF  R[A] из станд. ввода          адрес a[]
03: 8BFF  R[B] из станд. ввода          b = длина a[]
04: 7101  R[1] <- 0001
05: 7900  R[9] <- 0000                i = 0;
06: 22B9  R[2] <- R[B] - R[9]         while (i < b)
07: C20D  if (R[2]==0) PC <- 0D       {
08: 1CA9  R[C] <- R[A] + R[9]            адрес a[i]
09: 8DFF  R[D] из станд. ввода
0A: BD0C  M[R[C]] <- R[D]                чтение/запись a[i]
0B: 1991  R[9] <- R[9] + 1               i = i + 1
0C: C006  PC <- 06                    }
0D: EA00  PC <- R[A]
```

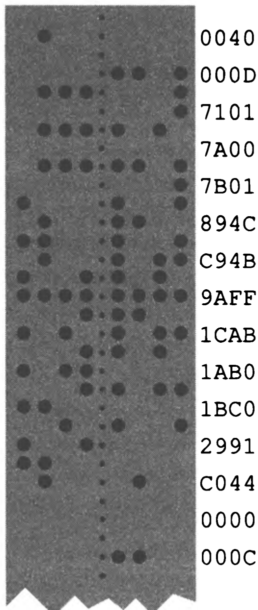
начальная загрузка (см. листинг 5.3.5)

```
00: 7A10  R[A] <- 0010
01: 7BEF  R[B] <- 239_10
02: 9AFF  R[A] в станд. вывод
03: 9BFF  R[B] в станд. вывод
04: 7101  R[1] <- 0001
05: 7900  R[9] <- 0000
06: 22B9  R[2] <- R[B] - R[9]
07: C20D  if (R[2]==0) PC <- 0D
08: 1CA9  R[C] <- R[A] + R[9]
09: AD0C  R[D] <- M[R[C]]
0A: 9DFF  R[D] в станд. вывод
0B: 1441  R[9] <- R[9] + 1
0C: C006  PC <- 06
0D: 0000  halt
```

сброс дампа

Начальная загрузка и дамп памяти

Эти две простые процедуры сильно упрощают работу программиста для таких компьютеров, как TOY. Сеанс начинается с ввода программы-загрузчика на переключателях и ее запуска для загрузки различных программ с перфоленты. Если в ходе повседневной работы вводится новый код (на переключателях), то программист вносит несколько изменений (ячейки **00-03** и **09-0A** в нашем случае) для преобразования начальной загрузки в вывод дампа, а перфоратор выводит данные на ленту для использования нового кода в будущем.



Дамп программы из листинга 6.3.2

Например, после ввода и отладки программы для вычисления чисел Фибоначчи (листинг 6.3.2) мы можем сохранить проделанную работу — для этого нужно определить, что программа состоит из **0D** слов, загружаемых в ячейки с **40** по **4C**, затем на переключателях изменить инструкцию в ячейке **00** на **7A40**, а инструкцию в ячейке **01** на **7B0D**, выставить на адресных переключателях значение **00**, нажать кнопку RUN для запуска программы сохранения дампа — и вы получите свою ленту. Далее остается провести загрузку с ленты, чтобы программа снова оказалась в памяти.

Как нетрудно себе представить, у программистов быстро накапливаются подборки перфолент с кодом. Такую подборку легко сравнить с очень ранней реализацией внешнего хранилища, а программу начальной загрузки — с очень ранней реализацией программы установки, которой вы регулярно пользуетесь для переноса программ на мобильное устройство.

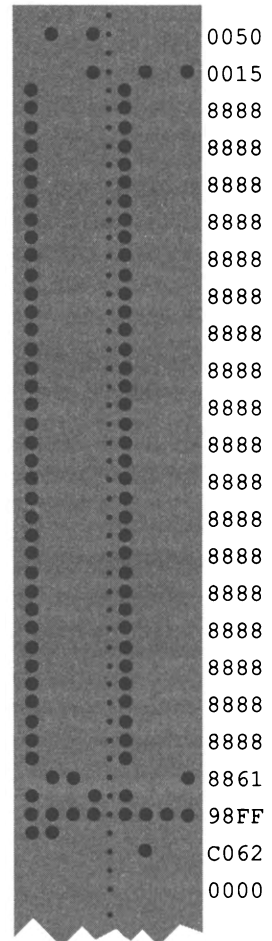
Предупреждение

Следующий пример показывает, что архитектура фон Неймана при всех своих выдающихся достоинствах может быть опасной в некоторых ситуациях.

Типичный рабочий процесс на таких машинах, как TOY, может выглядеть так: ученый разрабатывает программу для обработки экспериментальных данных, а затем использует ее неделями и месяцами для обработки данных. Так как эта работа часто сводилась к загрузке программы (либо вводом программы-загрузчика на переключателях, либо простой проверкой того, что она уже загружена ранее) и последующему вводу данных с перфоленты для их обработки программой, то для выполнения таких операций в вычислительных центрах обычно привлекались операторы (которые могли не обладать навыками программирования или научной квалификацией). Оператор загружал программу ученого в машину и запускал ее для разных наборов экспериментальных данных — обработка могла продолжаться часами.

Теперь представьте, что некто, регулярно использующий большую программу, начинает с загрузки массива программой из листинга 6.3.4 (которая занимает ячейки памяти с `60` по `6D`). В один прекрасный день коллега хочет воспользоваться программой и передает оператору ленту, изображенную справа. Оператор устанавливает ленту в считывающее устройство, нажимает RUN и (допустим) уходит на обед. Лента требует разместить в памяти массив из `21` слова, начиная с адреса `50`. Что произойдет?

Анализ этой ситуации значительно упрощается, если рассматривать ее с точки зрения *компьютера*. Какая бы инструкция ни хранилась по адресу, заданному счетчиком PC, мы знаем, что машина прочитает ее, увеличит PC и выполнит инструкцию, и этот цикл будет продолжаться вплоть до обнаружения инструкции останова. В данном примере программа-загрузчик загрузит 16 слов с ленты в память с заданным адресом и длиной; но в тот момент, когда `R[9]` содержит значение `5F` и увеличивается, к происходящему стоит присмотреться повнимательнее.



Загадочная лента

Увеличивая `5F`, мы получаем результат 60, поэтому следующее слово данных будет сохранено в `60`, следующее — в `61` и т. д., как показано в таблице на следующей странице. Процесс приводит к результатам, которые могут показаться неожиданными, потому что программа чтения массива начинает *перезаписывать себя в памяти*. В конечном итоге доходит до выполнения «инструкции», которая на самом деле была данными на ленте. В нашем примере инструкция

осуществляет переход к предыдущей инструкции (которая также была данными на ленте), и машина входит в бесконечный цикл, в котором на ленту выводится **8888**. Оператор возвращается с обеда и обнаруживает, что события развиваются, мягко говоря, неожиданно.

Мы привели эту вымышленную историю для того, чтобы в простой форме продемонстрировать идею неожиданного перехвата управления над компьютером. На месте трехстрочного цикла для вывода **8888** на ленту могла бы находиться любая программа. И такое может произойти не только с программой-загрузчиком: скорее всего, аналогичным образом можно будет перехватить управление в любой программе с вводом массива или в любой другой программе, которая читает и сохраняет данные.

эффект выполнения команд из ячеек 66–68

	R[C] ➡	60	61	62	63	64	
60:	8AFF	8888	8888	8888	8888	8888	
61:	8BFF	8BFF	8888	8888	8888	8888	*data*
62:	7101	7101	7101	8861	8861	8861	R[8] <- M[61]
63:	7900	7900	7900	7900	98FF	98FF	R[8] *в станд. вывод*
64:	22B9	22B9	22B9	22B9	22B9	C062	PC <- 62
65:	C26B	C26B	C26B	C26B	C26B	C26B	
66:	1CA9	1CA9	1CA9	1CA9	1CA9	1CA9	
67:	8DFF	8DFF	8DFF	8DFF	8DFF	8DFF	
68:	BD0C	BD0C	BD0C	BD0C	BD0C	BD0C	
69:	1991	1991	1991	1991	1991	1991	
6A:	C064	C064	C064	C064	C064	C064	
6B:	1AA9	1AA9	1AA9	1AA9	1AA9	1AA9	
6C:	BB0A	BB0A	BB0A	BB0A	BB0A	BB0A	
6D:	EF00	EF00	EF00	EF00	EF00	EF00	

Атака переполнения буфера для перехвата управления над TOY

Ситуация очень похожа на действие *вирусов*, которые десятилетиями досаждают пользователям компьютеров. Во многих ситуациях компьютерную систему можно легко «обмануть» и заставить ее передать управление в область памяти, в которой должны храниться данные, — с самыми печальными последствиями. Просто для примера: типичные программы, написанные на языке программирования С, подвержены *атакам переполнения буфера*, при которых пользователь передает строковый аргумент большей длины, чем рассчитывает получить функция. Так как код функции располагается сразу же после буфера, в котором должна храниться строка в памяти, злоумышленник может закодировать в длинной строке *программу*, как в нашем примере. Система передаст управление в ячейку памяти, в которой должна находиться функция, но тем самым она передает управление вредоносному коду. В случае *вируса* этот код связывается с другими компьютерами и заражает их, причем ситуация развивается стремительно. Задокументированы случаи заражения вирусом компьютеров миллионов пользователей, и они происходят снова и снова.

Нельзя ли написать программу, которая бы проверяла такую возможность? И как насчет антивирусных программ? К сожалению, такие программы просто проверяют известные вирусы, не пытаясь выяснить, что делает заданная последовательность инструкций. В общем случае из неразрешимости проблемы остановки (см. раздел 5.4) следует, что написать программу, которая бы проверяла, является ли любая заданная программа вирусом, невозможно[1].

Программы, обрабатывающие другие программы

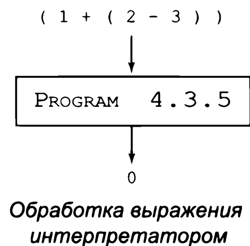
Также можно взглянуть на ситуацию с другой, гораздо более позитивной точки зрения: во многих случаях программы, которые получают на вход (или генерируют на выходе) другие программы, приносят огромную пользу. Мы неформально обсуждали такие программы еще тогда, когда говорили о компиляторе Java и виртуальной машине Java в разделе 1.1, но в контексте TOY можно привести более подробную информацию.

Ассемблер

Программирование на уровне шестнадцатеричных чисел неудобно и ненадежно, поэтому одной из первых разработок для многих компьютеров был *язык ассемблера*, который позволял использовать символические имена для операций и машинных адресов. Ассемблер — программа, которая получает на входе программу на языке ассемблера и выдает программу на машинном языке. Написать ассемблер для TOY на Java несложно (см. упражнение 6.4.13). Проделать то же самое на языке TOY посложнее, но первые программисты сталкивались с таким испытанием на самых разных компьютерах. Программирование на языке ассемблера широко распространено и в наши дни.

Интерпретатор

Интерпретатор — программа, которая напрямую выполняет программы, записанные на некотором языке программирования. Вы уже видели простой пример интерпретатора: программу для вычисления арифметических выражений (листинг 4.3.5). Арифметическое выражение описывает некоторые вычисления на очень простом языке программирования, а программа 4.3.5 выполняет эти вычисления. Собственно, сами вычисления достаточно просты, чтобы вы могли представить их реализацию на машине TOY (см. упражнение 6.4.15). Аналогично, но в гораздо большем масштабе многие современные языки программирования рассчитаны на обработку интерпретатором.



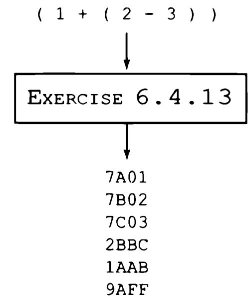
Обработка выражения интерпретатором

[1] Известны даже вирусы с изменяющимся кодом (*полиморфные*). Вместе с тем многие современные антивирусы могут *с некоторой вероятностью* распознавать *предположительно* вредоносный программный код. — *Примеч. науч. ред.*

Основная причина для использования интерпретируемых языков — их интерактивность, то есть вы можете последовательно вводить код и видеть результат его выполнения. Основная причина для того, чтобы *не использовать* интерпретируемые языки, — потенциальная неэффективность, так как системе приходится заново разбирать и обрабатывать каждую строку кода на исходном языке каждый раз, когда она встречается.

Компилятор

Компилятор — программа, преобразующая исходный код на одном компьютерном языке на другой компьютерный язык (часто машинный), обычно с целью создания исполняемой программы. Чтобы вы лучше поняли эту концепцию, мы рекомендуем опробовать упражнение 6.4.14, в котором вам предлагается преобразовать «вычислитель» арифметических выражений в компилятор. Например, если интерпретатор встречает знак +, он выполняет сложение; компилятор в такой ситуации генерирует машинную инструкцию, которая выполняет сложение. После того как вся исходная программа будет обработана компилятором, результатом является программа на машинном языке. Многие системы программирования промышленного уровня основаны на компиляции, потому что современные компиляторы могут генерировать программы на машинном языке, не уступающие по эффективности написанным вручную (и даже превосходящие их).



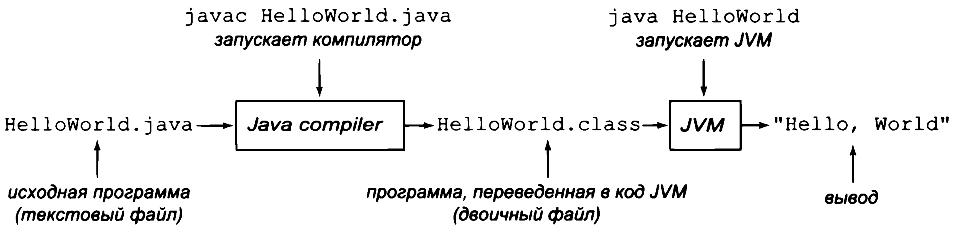
Обработка выражения компилятором

Виртуальная машина

Виртуальная машина — определение машины, которая выполняет программы, как и настоящая машина, но не обязательно соответствует какому-либо реальному аппаратному обеспечению. Конечно, TOY является виртуальной машиной! Исторически этот термин прошел значительное развитие. Чтобы программистам не приходилось переписывать существующие программы, каждая новая версия компьютерной системы включает программный или аппаратный *эмулятор* для запуска программ, написанных для ее предшественника. Одним из самых ранних применений виртуальных машин были *системы с разделением времени* — программы, которые, работая на единственном компьютере, создавали иллюзию запуска нескольких «экземпляров» вычислительных машин. Еще одним ранним применением, которое дожило до наших дней, стала промежуточная абстрактная прослойка между языками высокого уровня и аппаратурой компьютера (именно так действует Java — см. ниже). В современных вычислительных технологиях под термином «виртуальная машина» обычно объединяются все эти значения.

JVM

JVM (Java Virtual Machine) — классический пример виртуальной машины. Вместо того чтобы разрабатывать компилятор Java заново для каждого процессора, создатели системы Java решили, что будет намного лучше определить виртуальную машину со многими характеристиками реальных машин (регистры, память, счетчик адреса программы, инструкции для выполнения арифметических и логических операций, передача информации между регистрами и памятью, реализация переходов), но со своим набором инструкций, который называется *байт-кодом* и проектируется с расчетом на эффективное исполнение программой-интерпретатором, а не непосредственно процессором. Далее они направили все усилия на разработку компилятора с языка Java в язык JVM. Процесс, которому они следовали, изображен на диаграмме ниже. Чтобы язык Java работал на любом конкретном компьютере, достаточно написать интерпретатор для JVM — это намного проще, чем разрабатывать новый компилятор для Java. И хотя язык Java был разработан пару десятилетий назад, он успешно работает на новых машинах даже сегодня. О практической полезности этой идеи свидетельствует то, что появились новые языки, компилирующиеся в JVM. После компиляции эти языки работают на любом устройстве, способном выполнять программы Java.



Программы, обрабатывающие другие программы

Все эти разновидности программ чрезвычайно интересны, и вы с большой вероятностью встретите их в том или ином виде, когда начнете более серьезно заниматься программированием. Единственное различие между программой и данными — контекст. В случае машины TOY, если PC содержит адрес слова памяти, то содержимое этого слова считается инструкцией; в противном случае оно считается данными. Эта важнейшая характеристика архитектуры фон Неймана не просто какой-то фокус, а ключевой аспект современных вычислительных технологий. Тьюринг выдвинул идею, фон Нейман осознал ее практическую важность, а мир с тех пор пользуется ее плодами.

В текущем контексте нас интересует прежде всего связь между Java и TOY, поэтому сейчас мы рассмотрим эту тему.

TOY на языке Java

Программу на языке Java на следующей странице стоит рассмотреть повнимательнее. Она в полном смысле этого слова *является* машиной TOY, так как именно ее мы использовали при написании и отладке всех программ TOY в этой книге (и рекомендуем вам использовать ее для реализации и отладки ваших программ TOY). Вероятно, вас удивит, насколько проста и понятна эта программа. В самом деле, одно из первых требований к проектированию самой машины TOY заключалось в том, чтобы эта программа помещалась на одной странице. Перед вами полная программа, если не считать кода стандартного ввода и стандартного вывода, который был опущен, чтобы вам было проще сосредоточиться на логике работы машины.

Разбор инструкций TOY

Допустим, имеется инструкция TOY в переменной `IR` типа `int`. Это 32-разрядное значение, но инструкции TOY состоят всего из 16 битов, поэтому имеют значение только младшие 16 разрядов. При помощи операций сдвига и маскирования, рассмотренных в разделе 6.1, можно выделить код операции, регистры и адрес для последующего использования:

```
int op   = (IR >> 12) & 0xF;
int d    = (IR >>  8) & 0xF;
int s    = (IR >>  4) & 0xF;
int t    = (IR >>  0) & 0xF;
int addr = (IR >>  0) & 0xFF;
```

В каждой конкретной инструкции используются либо `s` и `t`, либо `addr`, но проще сразу вычислить все значения для каждой инструкции[1].

Состояние машины

С самого начала мы отметили, что поведение машины TOY полностью определяется содержимым регистров (в частности, PC) и содержимым памяти. Этот факт естественным образом приводит к выбору переменных экземпляров в программе 6.4.1: массив с 16 значениями `int` используется для представления содержимого регистров, массив из 256 значений `int` — для содержимого памяти и одно значение `int` для PC. Еще раз напомним, что фактически используются только младшие 16 разрядов этих значений (см. «Вопросы и ответы» в конце раздела).

Важно, что с учетом стандартного ввода размер состояния машины становится бесконечным, хотя сама машина является конечным автоматом. Машина имеет всего 4160 битов в оперативной памяти и в регистрах, но количество битов на входной ленте не ограничено.

[1] В некоторых языках (например, в C) предусмотрены так называемые *объединения*, позволяющие интерпретировать содержимое одной и той же области памяти как несколько разных значений *одновременно*. — *Примеч. науч. ред.*

Листинг 6.4.1. Виртуальная машина TOY (без стандартного ввода и вывода)

```
public class TOY
{
   private int[] R  = new int[16];
   private int[] M = new int[256];
   private int PC;
   public TOY(String filename)  // Конструктор (см. в тексте)
   public void run()
   {
      while (true)
      {
         int IR = M[PC];          // Выборка.
         PC = (PC + 1) & 0xFF;    // Инкремент.
         int op   = (IR >> 12) & 0xF;
         int d    = (IR >>  8) & 0xF;
         int s    = (IR >>  4) & 0xF;
         int t    = (IR >>  0) & 0xF;
         int addr = (IR >>  0) & 0xFF;
         if (op == 0) break;
         switch (op)
         {
           case  1: R[d] = R[s] +  R[t];              break;
           case  2: R[d] = R[s] -  R[t];              break;
           case  3: R[d] = R[s] &  R[t];              break;
           case  4: R[d] = R[s] ^  R[t];              break;
           case  5: R[d] = R[s] << R[t];              break;
           case  6: R[d] = (short) R[s] >> R[t];      break;
           case  7: R[d] = addr;                      break;
           case  8: R[d] = M[addr];                   break;
           case  9: M[addr] = R[d];                   break;
           case 10: R[d] = M[R[t] &0xFF];             break;
           case 11: M[R[t] & 0xFF] = R[d];            break;
           case 12: if ((short) R[d] == 0) PC = addr; break;
           case 13: if ((short) R[d] >  0) PC = addr; break;
           case 14: PC = R[d] & 0xFF;                 break;
           case 15: R[d] = PC; PC = addr;             break;
         }
         R[d] = R[d] & 0xFFFF;
         R[0] = 0;
      }
   }
   public static void main(String[] args)
   {  /* См. упражнение 6.4.2. */  }
}
```

R[]	регистры
M[]	память
PC	счетчик адреса программы

op	код операции
d	регистр результата
s	регистр аргумента
t	регистр аргумента
addr	поле адреса

Загрузка машины

Чтобы слегка упростить процесс запуска модели TOY, мы загружаем машину в конструкторе (см. код на с. 934). Клиент передает имя файла и начальное значение PC как аргументы конструктора; файл должен содержать последовательность инструкций, каждая его строка состоит из адреса памяти и соответствующей инструкции,

разделенных двоеточием и пробелом. Справа приведен файл fib.toy — программа из 13 слов (см. листинг 6.3.3), которая загружается в `M[40-4C]` и запускается с присваиванием PC значения 40. Иначе говоря, мы сохраняем программу TOY в файле, а затем производим начальную загрузку из этого файла, передавая имя файла и исходное значение PC в конструктор. Конструктор осуществляет загрузку памяти TOY, записывая последовательность инструкций в заданных ячейках памяти. После завершения конструктора программа TOY готова к запуску вызовом метода `run()`.

```
% more fib.toy
40: 7101
41: 7A00
42: 7B01
43: 894C
44: C94B
45: 9AFF
46: 1CAB
47: 1AB0
48: 1BC0
49: 2991
4A: C044
4B: 0000
4C: 000C
```

```
public TOY(String filename, int pc)
{
   PC = pc & 0xFF;
   In in = new In(filename;
   while (in.hasNextLine())
   {
      String line = in.readLine();
      String[] fields = line.split("[:\\s]+");
      int addr = Integer.parseInt(fields[0], 16) & 0xFF;
      int inst = Integer.parseInt(fields[1], 16) & 0xFFFF;
      M[addr] = inst;
   }
}
```

Конструктор для виртуальной машины TOY

На самом деле описанный процесс начальной загрузки довольно точно воспроизводит подход, применявшийся в реальных компьютерах, в которых машины загружаются специальным образом, отличным от механизмов ввода, используемых программами. Конкретная реализация механизма загрузки не столь важна; нас интересует лишь то, что происходит после заполнения памяти и инициализации PC заданным адресом. В нашем примере для этого вызывается метод `run()`. Вызов метода имитирует эффект нажатия кнопки RUN оператором после ввода (или загрузки) программы и выставления стартового адреса на переключателях.

Метод run()

Метод `run()` занимает центральное место в модели TOY, а его реализация чрезвычайно проста. Программа производит выборку инструкции, адрес которой хранится в `PC`, в переменную `IR` и увеличивает значение `PC` (в одной строке программы). Затем мы декодируем все составляющие инструкции (код операции, регистр результата, регистры аргументов и адрес) в соответствии с приведенным выше описанием. После извлечения информации изменения в состоянии машины описываются однострочными реализациями, объединенными в конструкцию `switch`. Вы сразу видите, что делает каждая инструкция, потому что код Java для каждой инструкции полностью соответствует описанию этой инструкции там, где она была впервые представлена.

Все, что происходит дальше, полностью зависит от инструкций и определяемых ими изменений в состоянии машины. Точно так же, как TOY выполняет инструкции в соответствии со значениями своего регистра PC, виртуальная машина выполняет инструкции в соответствии со значениями своей переменной `PC` и продолжает работу до тех пор, пока не встретит инструкцию останова (код операции `0`). В примере программы fib.toy (справа) результатом является вывод чисел Фибоначчи в стандартный вывод, как и предполагалось.

```
% java TOY fib.toy
0000
0001
0001
0002
0003
0005
0008
000D
0015
0022
0037
0059
```

```
private void stdin(int addr, int op, int t)
{
   if ((addr == 0xFF && op == 8) || (R[t] == 0xFF && op == 10))
      M[0xFF] = Integer.parseInt(StdIn.readString(), 16) & 0xFFFF;
}
private void stdout(int addr, int op, int t)
{
   if ((addr == 0xFF && op == 9) || (R[t] == 0xFF && op == 11))
      StdOut.printf("%04X\n", M[0xFF]);
}
```

Стандартный ввод и вывод для виртуальной машины TOY

Стандартный ввод и вывод

Наш процесс начальной загрузки читает загружаемую программу из файла, чтобы мы могли использовать `StdIn` для стандартного ввода и `StdOut` для стандартного вывода. Говоря конкретнее, мы должны прочитать значение из стандартного потока ввода в тот момент, когда инструкция загрузки (код операции `8`) или косвенной загрузки (код операции `A`) обращается к ячейке памяти `FF`, и записать значение в стандартный поток вывода, когда к этой же ячейке `FF` обращается инструкция сохранения (код операции `9`) или косвенного сохранения (код операции `B`). Этот код инкапсулируется в методах `stdin()` и `stdout()` на предыдущей странице. Чтобы добавить стандартный ввод и стандартный вывод в листинг 6.4.1, просто добавьте вызов `stdin(addr, op, t)` перед основной инструкцией `switch` и вызов `stdout(addr, op, t)` после этой инструкции. Также можно было выполнить эти проверки непосредственно перед обращением к памяти; в действительности этот вариант ближе к тому, как может действовать само аппаратное обеспечение (см. главу 7).

```
% more sum.toy
50: 7800
51: 8CFF
52: CC55
53: 188C
54: C051
55: 98FF
56: 0000
```

```
% more sum.txt
0001
0008
001B
0040
007D
00D8
0157
0200
0000
```

И снова подробности реализации несущественны — мы всего лишь хотим точно моделировать поведение машины. Также обратите внимание на то, что мы не стали затруднять вашу задачу двоичным вводом и выводом, как при работе с перфолентой (но см. упражнение 6.4.3).

```
% java TOY sum.toy 50 < sum.txt
0510
```

На с. 935 (справа) приведено содержимое файлов sum.toy (программа из 7 слов из листинга 6.3.3) и sum.txt (данные для этой программы, которые подаются на стандартный ввод и моделируют их присутствие на перфоленте). При вызове тестовый клиент из TOY.java загружает эту программу в ячейки M[50-56], присваивает PC значение 50 и вызывает run(). Смоделировать стандартный ввод TOY несложно — достаточно перенаправить стандартный поток ввода на файл sum.txt, как показано справа. Программа вызывает stdin(), чтобы ячейка M[FF] заполнялась из стандартного ввода при каждом выполнении инструкции 8CFF; таким образом, программа прочитает все числа и просуммирует их. Наконец, программа записывает результат в стандартный вывод инструкцией 98FF и останавливается. Как можно видеть, описанный процесс будет работать с любой программой TOY.

Разработка программ TOY

При необходимости мы сможем легко доработать программу 6.4.1, чтобы она обеспечивала трассировку содержимого PC, регистров и задействованных ячеек памяти в ходе выполнения и выдавала дамп памяти при необходимости (см. упражнение 6.4.3). В 1980-е годы программисты проводили значительную часть времени за изучением дампов памяти, потому что для многих ошибок только так можно было разобраться в том, что же происходит внутри машины. Собственно, программу 6.4.1 можно рассматривать как расширяемую среду разработки TOY — она может использоваться для написания и отладки программ TOY. Более того, вы можете доработать ее так, как считаете нужным, чтобы она выдавала любую информацию, необходимую для понимания работы вашей программы. Все это делается намного проще, чем на реальной машине (что в данном случае вообще невозможно, потому что машина TOY физически не существует). Добавьте в свою версию программы 6.4.1 ту функциональность, которая, как вы считаете, упростит разработку программ TOY. На сайте книги имеется программа (написанная таким же студентом, как вы) с графическим интерфейсом, в котором отображается состояние всех переключателей и индикаторов, с поддержкой трассировки и вывода дампа; пошагового выполнения программы по одной инструкции и множеством других функций.

Закон Мура

По крайней мере за последние 60–70 лет скорость и память современного компьютера удваиваются приблизительно каждые 18 месяцев. Это правило, называемое *законом Мура*, создает постоянную проблему: как строить новый компьютер, чтобы все усилия, потраченные ранее на разработку программного обеспечения, не пропали зря? Виртуальные машины играют в этом деле важнейшую роль, потому что одним из первых шагов (если не самым первым) проектирования нового компьютера становится построение для него виртуальной машины (такой, как в листинге 6.4.1) на старом. Данный подход обладает следующими преимуществами.

- Разработка программ для нового компьютера может начаться еще до того, как он появится на свет.

- Когда компьютер будет построен, его реальное поведение можно сравнивать с поведением виртуальной машины.
- Одним из первых программных продуктов для нового компьютера (если не самым первым) может стать виртуальная машина для старого компьютера! В этом случае любая программа, разработанная для старого компьютера, будет работать на новом компьютере.

Например, можно ли создать программу TOY, реализующую саму виртуальную машину TOY? Конечно! Мы без особых проблем преобразовали многие программы Java в эквивалентные программы TOY, а программа 6.4.1 относительно проста (см. упражнение 6.4.10). А когда в вашем распоряжении появится виртуальная машина TOY, вы можете усовершенствовать ее: добавить новые инструкции, расширить набор регистров, перейти на другой размер слова — все, что захотите. Полученная виртуальная машина TOY станет «более мощной», чем исходная. Таким образом, после того как вы построили одну машину, ее можно будет использовать для создания «более мощных» машин. Эта фундаментальная идея десятилетиями играла важную роль в проектировании компьютеров. Ее хорошо передает одна знаменитая цитата (за достоверность которой мы не ручаемся): «Сеймур Крэй, основатель Cray Research и создатель нескольких поколений суперкомпьютеров, услышал, что компания Apple купила суперкомпьютер Cray для моделирования архитектуры своих новых компьютеров. Крэй заметил: “Забавно, а я использую Apple для моделирования Cray-3”».

А теперь вопрос, над которым стоит задуматься: существует ли машина TOY? Физической машины TOY не существует, но мы все равно можем писать и отлаживать программы TOY. В этом смысле TOY не отличается от Java: мы пишем и отлаживаем программы Java, хотя физическая машина Java тоже не существует[1]. Собственно, программа 6.4.1 доказывает, что машина TOY реальна ровно в той же степени, как и Java. Могут существовать миллиарды реальных машин, реализующих JVM, — каждая из них также реализует TOY. Другими словами, если существует Java, то существует и TOY. Java может работать на миллиардах устройств; то же можно сказать и о TOY.

Каждая программа, описанная в этом разделе (и вообще любая программа TOY), может быть легко сохранена в файле, а в стандартном вводе могут быть представлены любые данные. Конструктор в листинге 6.4.1 загружает программу на виртуальную машину Java, моделирует ее работу для заданного ввода и направляет результаты ее работы (если они есть) в стандартный вывод.

Что еще важнее, любой программист (и вы тоже) может реализовать любую машину, спроектированную им самим, и написать программы для нее. Эта плодотворная идея подняла вычислительную инфраструктуру на ее сегодняшний уровень — и она же поведет нас в будущее, о чем будет рассказано далее.

---

[1] Вычислительные устройства, программируемые непосредственно на Java, существуют. Если не вдаваться в детали внутренней реализации, то с точки зрения программиста их можно считать физическими Java-машинами. — *Примеч. науч. ред.*

TOY как семейство воображаемых компьютеров

Наша воображаемая машина содержит всего 256 16-разрядных слов памяти. И хотя мы продемонстрировали, что теоретически на машине TOY можно реализовать любую программу, которая может быть реализована на Java, это ограничение вызывает естественное предположение: TOY наверняка не обладает достаточной памятью для выполнения каких-либо серьезных вычислений в реальном приложении.

Photo: NASA

Apollo 17 на лунной орбите

Это предположение полностью ошибочно. Компьютеры, похожие на TOY, годами использовались для всевозможных важных вычислений. Хотя бы один пример: навигационный компьютер Apollo, с помощью которого люди побывали на Луне шесть раз, располагал 1024 16-разрядными словами памяти — всего в 4 раза больше, чем у TOY!

И хотя устройства внешней памяти совершенствовались от перфолент и перфокарт до магнитных лент и дисковых накопителей, программисты понимали, что они могут обойтись относительно небольшим объемом (дорогостоящей) встроенной основной памяти — для этого нужно разбить программы на фазы (так называемые *оверлеи*), которые размещаются в памяти, выполняют свою работу, а затем загружают код следующей фазы из внешней памяти. К 1970-м годам на основе этого метода была разработана концепция ***виртуальной памяти***: операционная система создает иллюзию того, что программам доступна память, объем которой значительно превышает объем физической памяти компьютера. Эта концепция по-прежнему играет центральную роль в современных вычислительных технологиях.

Тем не менее развитие технологий постоянно открывает нам доступ к компьютерам с большими объемами памяти и большим быстродействием. Современным компьютерам доступны миллиарды битов памяти. Как можно их сравнить с крошечной машиной TOY?

При невероятных масштабах технологических достижений на всех фронтах остается неизменным принципиальный факт: программы на машинном языке для современных компьютеров отличаются от программ TOY намного меньше, чем можно было бы предположить.

TOY-64

Чтобы связать TOY с современными компьютерами, рассмотрим воображаемую 64-разрядную машину TOY, которую мы будем называть TOY-64. Описание такой машины может содержать точно такие же инструкции, какие мы уже рассматривали, но с выделением большего количества разрядов для идентификации регистров и ячеек оперативной памяти. А именно мы можем выделить 40 разрядов для адресов памяти и 20 разрядов для номеров регистров. Это будет означать, что TOY-64

сможет работать более чем с 68 миллиардами 64-разрядных слов и использовать более 250 тысяч регистров — конечно, такие показатели намного ближе к возможностям современных компьютеров, чем у исходной TOY или у реальной PDP-8.

Воображаемый 64-разрядный компьютер

Программирование для такой машины ничем не отличается от программирования для TOY, если не считать существенно большей разрядности слова, большего количества регистров и большего объема памяти. Даже если не углубляться в подробности, вы видите, что все значения будут представляться 16 шестнадцатеричными цифрами, а все рассмотренные представления расширяются естественным образом. Например, значение `40544F592D363421` может интерпретироваться как число 4 635 417 160 900 293 665_{10}, строка символов `"@TOY-64!"` или инструкция поразрядного XOR содержимого регистров `592D3` и `63421`, результат которой сохраняется в регистре `0544F`. Легко понять, как программа, разработанная для TOY, преобразуется в программу для машины TOY-64.

С учетом простоты такого преобразования возможность использования всей постоянно растущей горы старых программ скоро стала одной из важных целей при проектировании новых компьютеров. В самом деле, большая часть программных продуктов, используемых в наши дни, была разработана десятилетия назад, и мы можем использовать их, потому что новые машины сохраняют совместимость со старыми.

Скорее всего, у машины TOY-64 не будет переключателей и индикаторов — только беспроводной интерфейс и кнопка «вкл/выкл». Технические подробности несущественны; важно лишь то, что машина имеет доступ к потокам ввода/вывода, не ограниченным по длине с точки зрения программы.

Самое важное отличие TOY-64 от современных компьютеров — набор инструкций. На типичной машине под код операции выделяется большее количество битов, чтобы машина поддерживала больше инструкций для выполнения на аппаратном уровне различных операций, от вычислений с плавающей точкой до операций с памятью и поддержки внешней памяти. Тем не менее в области проектирования аппаратного обеспечения компьютеров существует и противоположное течение. Относительная простота написания программ по сравнению с крайней сложностью разработки надежных высокопроизводительных аппаратных решений привела к появлению компьютеров с сокращенным набором инструкций (*RISC*, Reduced Instruction Set Computer), которые продолжают существовать и в наше время. Так что набор инструкций типичного современного компьютера может оказаться ненамного — всего в 2–4 раза — больше, чем у TOY-64[1].

[1] Альтернативный подход называют *CISC* (Complex Instruction Set Computer) — максимально разнообразный набор инструкций, аппаратно реализующих различные операции. Не всегда, однако, архитектуру можно классифицировать как CISC или RISC только по количеству инструкций. — *Примеч. науч. ред.*

Другое значительное отличие TOY-64 от современных компьютеров заключается в том, что очень немногие современные компьютеры поддерживают столько регистров или инструкции типа *RR*. Мы не будем задерживаться на этом отличии, а лишь заметим, что во всех компьютерах существуют иерархии памяти: от дорогостоящей, быстрой и ограниченной по емкости до дешевой, медленной и емкой. В нашем наборе регистров отражается идея о том, что в архитектуре любого компьютера должны учитываться различия в технологиях памяти.

На этом уровне ситуация несколько отличается от уже знакомой модели «клиент — API — реализация», характерной для модульного программирования. Где должна проходить граница, «интерфейс» между программным и аппаратным обеспечением? Очень многие компьютеры остановились на интерфейсе, достаточно близком к TOY, так что теперь вы сможете начать писать программы для них. Необходимые для этого усилия сравнимы с теми, которые потребуются для изучения нового языка программирования.

TOY-8

Чтобы связать TOY с электронными схемами, из которых собран компьютер, в главе 7 мы расмотрим 8-разрядную машину TOY с 32 словами памяти и единственным регистром, которую будем называть *TOY-8*. Конечно, программирование для такой машины кажется серьезным испытанием, но заметьте: все программы, которые были рассмотрены в этом разделе, занимали в памяти менее 32 слов. А с учетом возможности программно подгружать дополнительный код с ленты становится ясно, что даже на такой крошечной машине можно кое-что сделать.

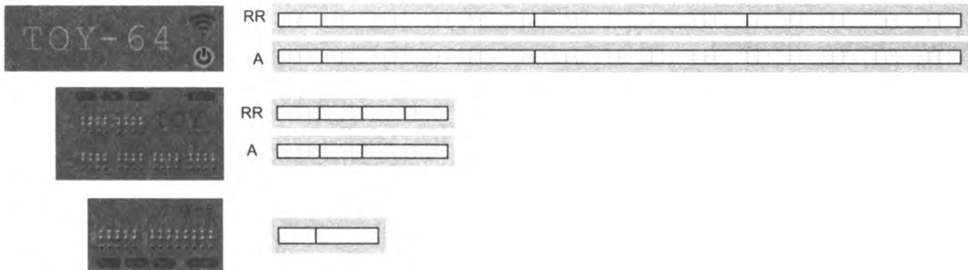
Стоит осознать, что даже у TOY-8 количество возможных состояний памяти равно 2^{256} — даже без учета внешней памяти. Таким образом, возможности TOY-8 неисчерпаемы (а абсолютное большинство возможных состояний TOY-8 не будет наблюдаться в этой Вселенной).

Даже с учетом сказанного основной целью TOY-8 является демонстрация полной машины, обладающей почти всеми характеристиками TOY, просто с меньшими значениями всех показателей. Мы описываем TOY-8 для того, чтобы представить полную электронную реализацию для компьютера, обладающего всеми важнейшими элементами TOY (и многих других компьютеров). Зная, как программируется TOY, вы сможете представить, как работают программы на таких машинах, как TOY-64 и ваш собственный компьютер; зная, как строится TOY-8, вы представите, как строятся TOY, TOY-64 и ваш собственный компьютер.

Сводка параметров, определяющих наше семейство воображаемых компьютеров, приведена в таблице и на иллюстрации на следующей странице. Конечно, между TOY и TOY-64 остается огромный пробел, потому что мы не приняли во внимание 32-разрядные компьютеры, широко применявшиеся в течение многих десятилетий. Оставляем вам эту тему для самостоятельной работы.

	TOY	TOY-64	TOY-8
Разрядность слов	16	64	8
Количество регистров	16	262 144	1
Количество слов оперативной памяти	256	68 719 476 736	32
Разрядность кода операции	4	4	3
Разрядность адреса регистра	4	20	0
Разрядность адреса оперативной памяти	8	40	5

Параметры семейства TOY



Семейство воображаемых компьютеров TOY

Виртуальная память

Может показаться, что одним из самых серьезных недостатков TOY является ограниченный объем памяти. Возникает вопрос: как смоделировать машину с бóльшим объемом памяти, чем на используемом компьютере? Из-за высокой стоимости памяти решением этой проблемы стали заниматься уже в самый ранний период. Представить себе это с бумажной перфолентой сложновато, но спустя короткое время появились магнитные ленты и диски, которые могли обеспечить значительный объем внешней памяти. С такими устройствами вскоре появилась и концепция *виртуальной памяти*. Многие программы в любой момент времени работают с относительно небольшой областью памяти, поэтому программы получают доступ к большой виртуальной памяти, которая в основном располагается во внешнем хранилище. Операционная система должна позаботиться о том, чтобы часть памяти, с которой должна работать программа, в нужный момент оказалась в физической оперативной памяти. Конечно, в таком случае расширение физиче-

ской памяти повышает эффективность работы программ, так как оно сокращает объем передачи данных между физической и внешней памятью.

Обратная совместимость

После смены многих поколений вычислительной техники мы приходим к удивительному факту: значительная часть программного обеспечения, используемого сегодня, была написана давно на менее мощной машине. Как правило, о программе никто ничего не знает, кроме того, что она делает. Когда вы покупаете новый компьютер, лишь малая часть используемых на нем программ написана именно для этого компьютера. Это сильно ускоряет прогресс: если виртуальная машина работает, то работает и все программное обеспечение! Но по прошествии времени могут возникнуть сложности. Известный пример такого рода — «проблема 2000 года», когда старые программные системы приходилось переписывать до наступления 2000 года, потому что для представления года в них было зарезервировано всего две цифры; в результате людям присваивался бы отрицательный возраст и возникали другие нежелательные последствия.

Серверные фермы

Зачем останавливаться на одном компьютере? Наша реализация TOY в программе 6.4.1 является типом данных, поэтому мы можем легко написать для них клиентское приложение, которое создает тысячу или миллион компьютеров TOY и запускает их. А современные технологии облачных вычислений позволяют запускать их одновременно на *серверных фермах*, состоящих из большого количества реальных процессоров. В наши дни сколько-нибудь значительные вычисления все чаще реализуются на виртуальной машине на серверной ферме. Когда ваше мобильное устройство распознает ваш голос или подвергает изображение цифровой обработке, скорее всего, ему в этом помогает виртуальная машина на серверной ферме.

Безусловно, наша воображаемая вычислительная машина — более чем скромный представитель вычислительных устройств, изобретенных человеком, но благодаря ее простоте мы можем лучше оценить суть фундаментальных вопросов, связанных с природой вычислений из главы 5. Что именно представляет собой компьютер или компьютерная программа? Какие аспекты компьютера принципиальны? Существуют ли новые технологии вычислений, способные значительно расширить наши представления? Существует ли скрытое объяснение того, почему эмуляция связывает воедино столь широкий спектр разных вычислительных устройств? Тщательный анализ вопросов такого рода приводит к практическому применению программ в форме интерпретаторов, компиляторов, программ установки и т. д.

Чтобы окончательно снять завесу тайны с TOY, остается разобраться в том, как спроектировать аппаратную реализацию этой машины. Этой теме посвящена следующая глава.

Серверная ферма TOY-64

Вопросы и ответы

В. Почему бы не использовать `short` для `R[]` и `M[]` в листинге 6.4.1? Это 16-разрядные значения, как и `short`.

О. При выполнении арифметических операций Java преобразует значения `short` в значения `int`, так что игра не стоит свеч[1]. В результате для моделирования поведения `short` иногда приходится выполнять явное преобразование или применять маску `0xFF`.

В. Что произойдет, если занести в PC значение `FF`? Какое значение находится в `FF`?

О. Конечно, мы можем реализовать любое поведение по своему усмотрению, но чтобы ответить на эти вопросы, необходимо проанализировать код программы 6.4.1 — виртуальной машины TOY. Очевидно, `M[FF]` изначально содержит 0, но затем в этой ячейке памяти хранится последнее значение, прочитанное вызовом `stdin()` или записанное `stdout()`. И конечно, злоумышленники могут воспользоваться этой особенностью поведения в своих целях! Возможно, будет лучше

[1] Более «короткий» тип данных мог бы (в зависимости от реализации выравнивания элементов массива) дать уменьшение занимаемого объема памяти. Но для современной техники такая экономия обычно неактуальна, а преобразования типа требовали бы дополнительных затрат времени. — *Примеч. науч. ред.*

обнулять значение этой ячейки в `stdin()` и `stdout()`, чтобы машина в этом случае останавливалась.

В. Является ли машина TOY универсальной (эквивалентной машине Тьюринга)?

О. В том виде, в котором мы ее описали, — не совсем, хотя проблема решается заменой устройств чтения/записи с использованием перфоленты двунаправленным накопителем на магнитной ленте с поддержкой чтения/записи[1]. Собственно, это стало одним из первых усовершенствований на многих старых компьютерах.

Упражнения

6.4.1. Сколько объема памяти (в байтах) доступно на серверной ферме TOY-64, изображенной в конце главы?

6.4.2. Реализуйте метод `main()` для TOY.java (листинг 6.4.1).

Решение

```
public static void main(String[] args)
{
   String filename = args[0];
   int pc = Integer.parseInt(args[1], 16);
   TOY toy = new TOY(filename, pc);
   toy.run();
}
```

6.4.3. Добавьте в программу TOY.java использование аргумента командной строки, в котором передается адрес памяти и номер регистра, а затем дополните код программы, чтобы виртуальная машина выводила содержимое заданного регистра непосредственно перед выполнением инструкции по заданному адресу памяти каждый раз, когда счетчик PC принимает это значение. Если аргумент равен 0, то при завершении выводится дамп памяти в форме, приведенной в тексте (в начале раздела 5.3.1).

6.4.4. Разработайте для TOY.java версии `stdin()` и `stdout()`, имитирующие вывод на перфоленту: для каждого 16-разрядного значения должны выводиться два ряда из 8 символов, в которых каждый нулевой бит представлен пробелом, а каждый единичный бит — символом *.

6.4.5. Модифицируйте программу TOY.java, чтобы инструкция вычитания была заменена инструкцией умножения. Обязательно предусмотрите то, что результат умножения двух 16-разрядных чисел является 32-разрядным.

6.4.6. Модифицируйте программу TOY.java, чтобы она поддерживала память TOY из 2^{16} слов; для этого измените способ адресации во всех инструкциях доступа к памяти на косвенный, через первые 256 слов памяти. Например, инструкция

[1] Имеется в виду, что перфолента, в отличие от ленты машины Тьюринга, не позволяет стереть или перезаписать уже записанные на нее значения. — *Примеч. науч. ред.*

8A23 должна загружать в R[A] слово памяти, 16-разрядный адрес которого хранится в M[23]. Реализуйте версию sum.toy для этой машины, которая может просуммировать 10 тысяч 16-разрядных значений в дополнительном коде.

Упражнения повышенной сложности

6.4.7. *Однорегистровая машина.* Спроектируйте 16-разрядный компьютер с одним регистром, 16 инструкциями и 4096 словами памяти. Каждая инструкция с двумя операндами получает один операнд из регистра, а другой из памяти и оставляет результат в регистре. Напишите программу для моделирования своей машины.

6.4.8. *Виртуальная память.* Представьте, что машина оснащена новым устройством внешней памяти с 2^{32} адресуемыми 32-разрядными словами, а интерфейс устройства с TOY реализует виртуальную память следующим образом: две последовательные операции записи в FF формируют 32-разрядный адрес. Затем, если следующие два обращения к FF являются операциями сохранения (или косвенного сохранения), они формируют инструкцию записи 32-разрядного значения по этому адресу; если следующие два обращения к ячейке FF являются операциями загрузки (или косвенной загрузки), они формируют инструкцию чтения 32-разрядного значения по этому адресу. Напишите версию sum.toy, которая может суммировать до 1 миллиона 32-разрядных значений в дополнительном коде.

6.4.9. *Параллельная машина TOY.* Измените программу TOY.java, чтобы она получала целое число n как аргумент командной строки и моделировала машину с n регистрами PC, пронумерованными от 0 до $n - 1$. В каждом цикле машина выполняет операции «выборка-инкремент-исполнение» для всех регистров PC одновременно. Если в каждом цикле два PC требуют внесения изменений в один регистр, то предпочтение отдается PC с меньшим индексом.

6.4.10. *TOY в TOY.* Разработайте программу TOY.toy, которая реализует виртуальную машину TOY. Для начала предположите, что машина оснащена 32 словами памяти, 8 регистрами и не поддерживает стандартный ввод/вывод. Остаток памяти (и стандартный ввод/вывод) могут использоваться для разработки программы. Затем добавляйте стандартный ввод/вывод, дополнительную память и дополнительные регистры до тех пор, пока на машине не станет возможно запускать любые программы из раздела 6.3.

6.4.11. *Строковая машина TOY.* Спроектируйте и постройте модель воображаемой 16-разрядной строковой машины с такими же регистрами и памятью, как у TOY, но со строковыми операциями. Предполагается, что строки хранятся в виде последовательности слов, по два ASCII-символа на слово, и завершаются байтом 00 (нуль-терминированные строки). Включите операции для поиска, извлечения подстрок и стандартного ввода/вывода. Напишите программу сортировки массива строк (точнее, ссылок на строки) с использованием сортировки методом вставки.

6.4.12. *Быстродействие.* Что работает быстрее: TOY или Java? Проведите тесты с удвоением размера задачи, чтобы определить соотношение времени выполнения

нашей программы TOY и программы Java, использующих поиск в ширину для удаления дубликатов в файлах 16-разрядных целых чисел.

6.4.13. *Ассемблер.* Напишите программу Java, которая получает на вход программу TOY, написанную на языке чуть более высокого уровня (*языке ассемблера*), и генерирует программу TOY для программы-загрузчика, приведенной в тексте. Язык ассемблера поддерживает использование *символических имен* для адресов, кодов операций и регистров. Например, ниже приведена версия листинга 6.3.4 на языке ассемблера (программа выводит числа Фибоначчи в стандартный вывод):

```
     LA one, 1
     LA a, 0
     LA b, 1
     L  i, N
loop BZ i, done
     ST a, stdout
     A  c, a, b
     A  a, b, 0
     A  b, c, 0
     S  i, i, one
     BZ 0, loop
done H
N    000C
```

Между строками кода на языке ассемблера и инструкциями TOY должно существовать однозначное соответствие, но остальные подробности остаются на ваше усмотрение. Одно из больших преимуществ языка ассемблера перед машинным языком заключается в том, что программа может быть загружена с любого адреса (ассемблер сам вычисляет адреса), поэтому ваша программа должна получать начальный адрес как аргумент командной строки.

6.4.14. *Компилятор выражений.* Измените алгоритм Дейкстры (листинг 4.3.5), чтобы он выводил программу TOY для вычисления заданного выражения.

6.4.15. *Интерпретатор выражений.* Разработайте реализацию алгоритма Дейкстры (листинг 4.3.5) для машины TOY, опустив функцию квадратного корня. Считайте, что входное выражение находится в стандартном вводе, все операнды являются неотрицательными 15-разрядными числами без знака, а для кодирования операторов и разделителей используются отрицательные числа: `8001` для `+`, `8002` для `-`, `8003` для `*`, `8004` для `/`, `8005` для `(` и `8006` для `)`. Используйте свою реализацию стека из упражнения 6.3.25 и реализацию умножения из упражнения 6.3.35.

6.4.16. *32-разрядная машина TOY.* Спроектируйте 32-разрядный компьютер TOY. Обоснуйте каждое принятое решение. Реализуйте виртуальную машину для своей версии TOY-32.

6.4.17. *Отражающийся шар.* Разработайте программу TOY, которая генерирует команды для внешнего устройства — чертежной машины (плоттера), получающей 16-разрядные команды. Первая шестнадцатеричная цифра содержит код операции; остальные могут содержать информацию. В контексте этого упражнения

нас интересуют всего два кода операций: код 0 заносит 12-разрядное значение из оставшейся части слова в стек, а код 1 извлекает три 12-разрядных значения из стека (r, затем y, затем x) и строит окружность с центром в точке (x, y) и радиусом r. Например, код внизу выдает в стандартный вывод последовательность команд, результатом выполнения которых устройством будет движущийся шар (шар движется слева направо и возвращается к левому краю при выходе за правый край). Доработайте эту программу, чтобы она генерировала команды вывода шара, отражающегося от краев экрана (см. листинг 1.5.6).

```
10: 7AEE  x
11: 7BFF  y
12: 710F  dx
13: 7C32  r
14: 9CFF  push r
15: 9BFF  push y
16: 1AA1  x += dx
17: 831E  R[3] = маска
18: 3AA3  x &= маска
1A: 9AFF  push x
1B: 871D  R[7] = команда
1C: 97FF  push команда
1D: C014  цикл
1E: 1010
1F: 0FFF
```

Генерирование команд для устройства графического вывода

6.4.18. *Виртуальная рисующая машина.* Напишите программу Java DrawingTOY.java, которая использует `StdDraw` для моделирования рисующего устройства, описанного в предыдущем упражнении. Расширьте машину так, чтобы она могла рисовать квадраты, линии и многоугольники, а затем напишите код TOY для построения желаемых изображений.

Глава 7

Построение вычислительного устройства

Вероятно, вы считаете, что проектирование процессора компьютера — грандиозная задача, требующая участия целой армии людей с высочайшей квалификацией. Несомненно, в этом представлении есть зерно истины, но общая архитектура типичного процессора на удивление проста, причем она остается неизменной со времен первых компьютеров. В этой главе во всех подробностях рассматривается процесс проектирования конкретного компьютера общего назначения. Описывая этот процесс, мы хотим помочь вам в поиске ответов на вопросы типа: «Как устроены компьютеры?» и «Как работают компьютеры?»

На концептуальном уровне компьютер может рассматриваться как «черный ящик», связанный с устройствами ввода и вывода. Но что находится внутри «черного ящика»? Если вы откроете корпус своего компьютера, вы увидите несколько электронных блоков, связанных друг с другом на системной плате. Что они делают? Бóльшая часть из них управляет устройствами ввода и вывода, но один из них — *центральный процессор* (CPU, Central Processing Unit) — является сердцем, мозгом и душой компьютера, потому что в компьютере почти ничего не происходит без сигнала от центрального процессора. А если бы вам удалось заглянуть внутрь процессора (вероятно, в микроскоп), вы бы увидели, что он представляет собой не что иное, как множество крошечных блоков, соединенных проводниками. Ранние компьютеры физически сильно отличались от современных (и занимали целые машинные залы), но в основном потому, что это блоки и проводники между ними занимали больше места.

Наша цель не только объяснить, как работают компьютеры, но и убедить вас в том, что проектирование компьютера — вполне посильная задача. Как и многие задачи из области компьютерных наук, она требует внимания к подробностям, но на концептуальном уровне процесс проектирования компьютеров на удивление прост. Этот пример наглядно демонстрирует мощь абстракции.

7.1. Булева логика

Вы уже знакомы с концепцией математических функций — например, мы подробно рассматривали их в контексте реализующих их программ. *Логическая,* или *булева, функция* (называемая также *переключательной функцией*) — математическая функция, отображающая аргументы на значение, у которой *область определения* (аргументы функции) и *область допустимых значений* (значение функции) включают всего два возможных значения. Концепция остается неизменной, какие бы два значения ни были выбраны: `true` и `false`, «да» и «нет», 0 и 1. Теория логических функций называется булевой логикой.

Джордж Буль (1815–1864)

Булева логика была создана английским математиком Джорджем Булем (George Boole) в XIX веке. С тех пор она служит основой для логических рассуждений. Подробное изложение булевой логики заняло бы целую книгу (хотя чтение такой книги или прохождение учебного курса принесло бы большую пользу). В этом разделе мы начнем с первых принципов и сосредоточимся на концепциях, связанных с вычислениями, прежде всего на реализации цифровых схем.

Логические функции уже неоднократно встречались вам в книге.

- В главе 1 вы познакомились с типом данных Java `boolean` и там же научились применять его для принятия решений в конструкциях `if` и `while` в ваших программах.
- В главе 5 рассматривалась важная роль задачи выполнимости логических уравнений в теории вычислений.
- В главе 6 было продемонстрировано применение логических функций при работе с двоичным представлением данных.

Материал этого раздела очень важен, поэтому мы включили в книгу несколько опережающих ссылок. Может оказаться, что вы читаете этот раздел, не ознакомившись с частью предшествующего материала. Проблем не будет, потому что основная информация, необходимая в предыдущих главах, усваивается достаточно легко, а этот раздел мы писали так, чтобы он не зависел от других. Мы выбрали нелинейную структуру книги, потому что между логическими функциями и электронными схемами, реализующими вычислительные операции, существует тесная связь; это один из фундаментальных принципов, который привел к формированию современной вычислительной инфраструктуры. Мы постараемся привлечь ваше внимание к этой связи, а заодно помочь вам оценить красоту вклада Буля в математику. Вряд ли Буль сознавал, что почти через два столетия его работа послужит основой для вычислительных технологий. Понимание этой связи эквивалентно пониманию вопроса: «Как электрические цепи выполняют вычисления?»

Логические функции

Начнем с простых и уже знакомых логических функций: отрицания, конъюнкции и дизъюнкции. Чтобы определить любую логическую функцию, достаточно задать ее значение для каждой возможной комбинации ее входных значений. В этом разделе логические значения будут обозначаться 0 и 1. Для представления логических значений в символических выражениях используются *логические переменные*. Например, функция отрицания NOT (или НЕ) является функцией одной логической переменной и определяется следующим образом:

```
         0 если x равно 1
NOT(x) =
         1 если x равно 0
```

Аналогичным образом функции AND (И), OR (ИЛИ) и XOR (исключающее ИЛИ) являются функциями двух переменных и определяются следующим образом:

```
         0 если x или y (или оба сразу) равны 0
AND(x,y) =
         1 если x и y равны 1

         0 если x и y равны 0
OR(x,y) =
         1 если x или y (или оба сразу) равны 1

         0 если x и y равны
XOR(x,y) =
         1 если x и y различны
```

Чтобы усвоить эти определения на интуитивном уровне, можно интерпретировать x и y как логические утверждения вида «небо синее» или «светит солнце», а затем интерпретировать 1 как «истину» и 0 как «ложь». Например, результат AND(x,y) будет истинным, если истинно как x, так и y, и ложным в любом другом случае; в сущности, это формальное выражение нашего интуитивного представления о том, что утверждение вида «небо синее И светит солнце» истинно только в том случае, если истинны обе его составляющие. В математическом контексте, если x является утверждением «целое число v больше либо равно 0», а y — утверждением «целое число v меньше либо равно 0», AND(x,y) фактически утверждает, что v равно 0 (с соответствующими аксиомами относительно целых чисел). Практические применения такого рода заставили Буля взяться за изучение таких функций.

Обозначения

За два столетия своего существования булева логика стала играть важную роль в множестве контекстов, поэтому появилось много разных систем обозначений даже для элементарных операций. Более того, некоторые из них уже встречались вам в книге. В этой главе запись x' соответствует NOT(x); логическое произведение xy соответствует AND(x,y); а логическая сумма x+y соответствует OR(x,y).

Произведение согласуется с интуитивным представлением об умножении 0 и 1, но в случае с суммой имеем 1 + 1 = 1 (поскольку все значения должны быть равны 0 или 1).

Эта запись чрезвычайно компактна, и она широко применяется в этой главе. Чтобы избежать недоразумений, приведем сводку других обозначений, встречавшихся в книге.

	логические выражения	*логические операции в Java*	*битовые операции в Java*	*схемотехника*
NOT	¬x	!x	~x	x'
AND	x∧y	x && y	x & y	xy
OR	x∨y	x || y	x | y	x+ y
XOR	x⊕y	x ^ y	x ^ y	x ⊕ y

Обозначения элементарных логических функций

Таблицы истинности

Как упоминалось ранее, один из способов определения логической функции основан на указании ее значения для всех возможных наборов значений ее аргументов. Структурировать такое определение позволяют *таблицы истинности*. Таблица истинности содержит столбец для каждой переменной, строку для каждой возможной комбинации значений переменной и столбец, определяющий значение функции для этой комбинации. Например, вот как выглядят таблицы истинности для определенных выше элементарных функций:

NOT

x	x'
0	1
1	0

AND

x	y	x y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

Определения таблиц истинности для элементарных логических функций

Таблица истинности для функции *n* переменных содержит 2^n строк, поэтому для больших *n* таблицы истинности не используются. Как вы вскоре увидите, таблицы истинности позволяют не только определять функции, но и проверять коррект-

ность (валидность) различных операций, потому что это выполняется тоже путем систематического перебора всех возможных вариантов.

Существуют ровно 16 логических функций двух переменных, и мы можем перечислить их в одной таблице.

x	y	0	AND	xy'	x		y	XOR	OR	NOR	EQ	y'		x'		NAND	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Все логические функции двух переменных

Некоторые из этих функций часто встречаются как в математической логике, так и в проектировании цифровых схем — даже те, для которых не указаны названия (см. упражнения 7.2.2). В частности, обратите особое внимание на функции *NOR* (*NOT OR*, *ИЛИ-НЕ*), *NAND* (*NOT AND*, *И-Н*Е) и xy' (A*ND NOT*).

Булева алгебра

Логическим (или *булевым*) *оператором* называется символическое обозначение логической (булевой) функции; термином «булева алгебра» обозначаются преобразования выражений, состоящих из логических переменных и логических операторов. Как вы вскоре увидите, из-за ограниченного набора значений 0 и 1 булева алгебра отличается от алгебры с вещественными числами, которую вы изучали в школе (булева алгебра намного проще); впрочем, между ними существует и некоторое сходство.

Общая концепция *алгебры* является предметом изучения в обширной области математики; это еще одна нетривиальная тема, выходящая за рамки книги. Для булевой алгебры тщательное определение аксиом (неопровержимых истин) позволяет логически выводить тождества и теоремы, которые могут применяться для изучения и вывода свойств логических функций. В тексте аксиомы, тождества и теоремы неформально называются *законами*. Основные законы булевой алгебры имеют простые формулировки, и многие из них выглядят знакомо. Привычные законы коммутативности, дистрибутивности и ассоциативности из традиционной алгебры переходят в булеву алгебру, как показано в таблице на с. 953. Кроме того, легко доказываются многие другие законы, относящиеся только к булевой алгебре. Например, в последней строке таблицы приведены два специальных тождества для функций NAND и NOR, известные как *законы де Моргана*.

Аксиомы

Тождество	$1x = x$ $x + 0 = x$
Дополнение	$xx' = 0$ $x + x' = 1$
Коммутативность	$xy = yx$ $x + y = y + x$
Дистрибутивность	$x(y + z) = xy + xz$ $x + yz = (x + y)(x + z)$
Ассоциативность	$(xy)z = x(yz)$ $(x + y) + z = x + (y + z)$

Тождества и теоремы

Отрицание	$0' = 1$ $1' = 0$
Двойное отрицание	$(x')' = x$
Аннулирование	$0x = 0$ $1 + x = 1$
Поглощение	$x(x + y) = x$ $x + xy = x$
Законы де Моргана	$(xy)' = x' + y'$ $(x + y)' = x'y'$

Базовые законы булевой алгебры

Все эти законы легко проверяются по таблицам истинности. Собственно, этот способ уже использовался в качестве примера в разделе 1.2. Из-за важности законов де Моргана мы повторим их доказательство методом таблиц истинности, используя запись этой главы.

NAND										NOR									
x	y	xy	(xy)'	x	y	x'	y'	x'+y'		x	y	x+y	(x+y)'	x	y	x'	y'	x'y'	
0	0	0	1	0	0	1	1	1		0	0	0	1	0	0	1	1	1	
0	1	0	1	0	1	1	1	1		0	1	1	0	0	1	1	0	0	
1	0	0	1	1	0	0	1	1		1	0	1	0	1	0	0	1	0	
1	1	1	0	1	1	0	0	0		1	1	1	0	1	1	0	0	0	

Доказательство законов де Моргана с использованием таблиц истинности

Булева алгебра в Java

Как вы, вероятно, уже знаете, булева алгебра может включаться в программы Java двумя разными способами. Различия между ними сбивают с толку новичков (и дают преподавателям материал для контрольных вопросов), поэтому на них стоит остановиться подробнее.

- Тип данных Java `boolean`: в разделе 1.2 были представлены логические операции со значениями `true` и `false`, а также операции AND, OR и NOT с использованием операторов `&&`, `||` и `!` соответственно. С тех пор мы использовали логические выражения с переменными типа `boolean` и этими операторами (и выражения со смешанными типами, возвращающими значение `boolean`) для управления последовательностью выполнения в программе. Java поддерживает логические выражения произвольной сложности, как показывает наш первый пример `LeapYear` (листинг 1.2.4), но в остальных программах почти всегда используются предельно простые выражения.
- Битовые операции с целочисленными значениями: в разделе 6.1 были описаны битовые операции Java, при которых операторы AND, OR, NOT и XOR применяются к каждому биту двоичного представления целых чисел, для чего используются операторы `&`, `|`, `~` и `^` соответственно. И снова язык Java поддерживает выражения произвольной сложности, позволяющие работать с отдельными битами данных. Классический пример такого рода — виртуальная машина TOY (листинг 6.4.1).

Каждому программисту в конечном итоге приходится применять булеву алгебру обоими способами, и такие операции стали неотъемлемым элементом современных языков программирования.

Практическое применение

Чтобы показать, как эти концепции применяются на практике, рассмотрим задачу из области криптографии.

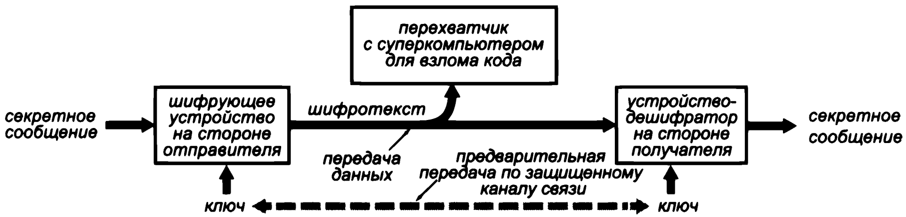
Фундаментальная задача криптографии — передача секретного сообщения от отправителя к получателю таким образом, чтобы в случае перехвата оно не могло быть прочитано посторонними. Для этого отправитель при помощи шифровального устройства создает зашифрованный текст (*шифротекст*, или *криптограмму*), а получатель расшифровывает при помощи устройства-дешифратора. *Криптографическая система* представляет собой протокол для решения этой задачи.

На с. 955 изображена элементарная криптографическая система. Она основана на применении криптографического ключа для обеспечения защищенной передачи данных. Идея заключается в том, что отправитель и получатель заранее обмениваются ключами каким-либо защищенным способом, чтобы отправитель мог использовать ключ для шифрования сообщения, а получатель мог воспользоваться тем же ключом для его расшифровки. Например, в XX веке в ходе мировых

войн на море и на суше командиры брали с собой так называемые шифровальные книги; такие же книги хранились в штабах. Для каждого дня в таких книгах был указан ключ, который должен использоваться для защищенной передачи данных в этот день.

Особенно простой метод шифрования/дешифрования основан на прямом применении булевой логики. Чтобы отправить сообщение, мы преобразуем его в двоичную строку, а затем шифруем, применяя поразрядную операцию XOR с ключом:

```
сообщение                S       E       C       R       E       T
сообщение (дв.)   m      010100110100010101000011010100100100010101010100
ключ              k      110010010011110110111001011010111001100010111111
шифротекст        m⊕k    100110100111100011111010001110011101110111101011
```



Элементарная криптографическая система

Криптографический ключ представляет собой обычную последовательность битов, длина которой равна длине сообщения. Конечно, «устройство» шифрования в наши дни представляет собой программу, поэтому вы можете легко написать программу Java, которая выполняет необходимые вычисления с двумя последовательностями битов произвольной длины для получения шифротекста (см. упражнение 7.1.18). Если биты ключа будут выбираться случайно с равномерным распределением, то перехватчику не удастся понять шифротекст (его биты также будут случайными). Безопасность криптографической системы зависит от степени «случайности» ключа — наука и искусство криптографии заключаются в создании и безопасном распространении ключей, которые бы в максимально возможной степени обладали свойствами случайных ключей, чтобы сторона, перехватившая зашифрованное сообщение, не смогла его прочитать, даже если в ее распоряжении будет целый вычислительный центр с суперкомпьютерами.

Но как получатель расшифровывает сообщение? Как это ни удивительно, используя тот же ключ, получатель может применить для дешифрования сообщения тот же процесс, который использовался для его шифрования:

```
шифротекст        c      100110100111100011111010001110011101110111101011
ключ              k      110010010011110110111001011010111001100010111111
сообщение (дв.)   c⊕k    010100110100010101000011010100100100010101010100
сообщение                S       E       C       R       E       T
```

На первый взгляд творится какое-то волшебство, но мы можем раскрыть его секрет, *доказав*, что $(c \oplus k) = ((m \oplus k) \oplus k) = m$. Эти операции выполняются с каждым битом сообщения, и из формулы следует, что получатель успешно восстановит каждый бит.

Для доказательства этого тождества можно воспользоваться таблицей истинности (см. упражнение 7.1.4), но это превосходный пример для демонстрации того, что с булевой алгеброй можно доказывать тождества, не прибегая к таблицам истинности. Следующая таблица содержит алгебраическое определение и тождества, необходимые для доказательства:

Определение	$x \oplus y = xy' + x'y$
Тождество	$x \oplus 0 = x$
Аннулирование	$x \oplus x = 0$
Ассоциативность	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$

Законы тождества, аннулирования и ассоциативности легко доказываются на основании определения и базовых законов булевой алгебры на с. 953 (см. упражнение 7.1.5). С этими законами утверждение доказывается тривиально:

$(m \oplus k) \oplus k = m \oplus (k \oplus k)$	Ассоциативность
$= m \oplus 0$	Аннулирование
$= m$	Тождество

Этот на удивление простой механизм широко применялся в течение очень долгого времени, и он продолжает широко применяться в современных системах. Конечно, в наши дни люди редко пользуются услугами доверенных курьеров — современная криптография базируется на более сложных и гораздо более удобных методах распространения ключей (самый, вероятно, распространенный метод RSA основан на сложности разложения числа на простые множители). И конечно, в области генерирования «случайных» ключей ведутся активные исследования (см. упражнение 7.1.15).

Это всего лишь один небольшой пример практического применения булевой алгебры. Тем не менее, как и многие практические применения математической теории, встречавшиеся в этой книге, он доказывает важность фундаментальных исследований. Почти два столетия назад Буль вел исследования в этой области, не подозревая о том, что его алгебра сыграет такую роль в нашем понимании криптографической инфраструктуры, окружающей интернет-коммерцию.

Логические функции трех и более переменных

С ростом числа переменных количество возможных функций резко возрастает. С тремя переменными существуют 2^8 разных функций, у 4 переменных — 2^{16} функций, у 5 переменных — 2^{32} функций, у 6 переменных — 2^{64} функций и т. д. Исходя

из этого и из нашего обсуждения экспоненциального роста в разделе 5.5 следует, что большинство возможных логических функций с большим количеством переменных никогда не встретится вам на практике. Тем не менее некоторые функции играют критическую роль в вычислениях и проектировании электронных схем, поэтому мы их сейчас рассмотрим.

Определения функций AND и OR с несколькими аргументами естественным образом обобщаются от определений с двумя аргументами:

```
                 0 если хотя бы один аргумент равен 0
AND(x1,...xn) =
                 1 если все аргументы равны 1

                 0 если все аргументы равны 0
OR(x1,...xn) =
                 1 если хотя бы один аргумент равен 1
```

В таблице истинности функции AND все значения, кроме нижнего, равны 0. В таблице истинности функции OR все значения, кроме верхнего, равны 1.

Количество возможных комбинацией огромно. В самом деле, логические функции могут охватывать любую вычислительную задачу. Например, можно определить функцию PRIME(x_1,...x_n) равной 1 в том и только в том случае, если двоичное число x_1 x_2 x_3 ... x_n является простым; или функцию TOY_k(x_1,...x_n) равной 1 в том и только в том случае, если k правых индикаторов на передней панели машины TOY светятся при остановке TOY после того, как оператор инициализирует все биты в памяти и в счетчике адреса программы (PC) значениями x_1 x_2 x_3 ... x_n и нажмет кнопку RUN. Такие функции прекрасно определяются, хотя их таблицы истинности могут быть невообразимо огромными.

А пока рассмотрим еще два примера, встречающихся при проектировании цифровых схем: *мажоритарная функция* (MAJ) и функция нечетности (ODD):

```
                 1 если количество 1 в аргументах (строго) больше 0
MAJ(x1,...xn) =
                 0 в противном случае

                 1 если нечетное число аргументов равно 1
ODD(x1,...xn) =
                 0 в противном случае
```

Такие функции, как AND и OR, являются *симметричными*: они не зависят от порядка своих аргументов.

Логические выражения

Как и в случае с логическими функциями двух переменных, мы можем явно задать значения переменных любой логической функции в таблицах истинности. Для трех переменных такая таблица состоит из $2^3 = 8$ строк. Например, таблица со значениями функций AND, OR, MAJ и ODD с тремя переменными выглядит так.

x	y	z	AND(x, y, z)	OR(x, y, z)	MAJ(x, y, z)	ODD(x, y, z)
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	0	1	1	0
1	1	0	0	1	1	0
1	1	1	1	1	1	1

Некоторые логические функции с тремя переменными

Это представление получается слишком громоздким, и оно быстро перестает работать для функций с большим количеством переменных, потому что, как упоминалось ранее, количество строк, необходимых для n переменных, равно 2^n. Вместо этого для определения логических функций часто бывает удобнее использовать логические выражения. Например, нетрудно убедиться в истинности следующих двух тождеств:

$$\text{AND}(x_1, \ldots x_n) = x_1 x_2 \ldots x_n$$
$$\text{OR}(x_1, \ldots x_n) = x_1 + x_2 + \ldots + x_n$$

С точки зрения булевой алгебры эти функции можно было бы определить при помощи таких выражений. Но какие логические выражения соответствуют MAJ, ODD или любой другой логической функции, которую нужно будет рассмотреть?

Представление суммы произведений

Один из фундаментальных результатов булевой алгебры гласит, что любая логическая функция может быть представлена выражением, в котором используются операторы AND, OR и NOT (конкатенация, + и ') и никакие другие. Этот факт на первый взгляд выглядит удивительно, но, как ни странно, он так же легко достигается. Например, возьмем следующую таблицу истинности:

x	y	z	MAJ	x'	y'	z'	x'yz	xy'z	xyz'	xyz	x'yz + xy'z + xyz' + xyz
0	0	0	**0**	1	1	1	0	0	0	0	**0**
0	0	1	**0**	1	1	0	0	0	0	0	**0**
0	1	0	**0**	1	0	1	0	0	0	0	**0**
0	1	1	**1**	1	0	0	1	0	0	0	**1**
1	0	0	**0**	0	1	1	0	0	0	0	**0**
1	0	1	**1**	0	1	0	0	1	0	0	**1**
1	1	0	**1**	0	0	1	0	0	1	0	**1**
1	1	1	**1**	0	0	0	0	0	0	1	**1**

Доказательство представления MAJ(x, y, z) в виде суммы произведений (ДНФ)

Так как значения равны для всех значений переменных, столбцы, выделенные в этой таблице жирным шрифтом, доказывают следующее равенство:

$$MAJ(x, y, z) = x'yz + xy'z + xyz' + xyz$$

Логическое выражение, которое мы строим, называется *логической суммой произведений*, или *дизъюнктивной нормальной формой функции (ДНФ)*.

Эта конструкция наглядно показывает, как вывести такое выражение для любой логической функции из ее таблицы истинности: для каждой строки таблицы истинности, у которой значение функции равно 1, создается слагаемое, равное 1, если входные переменные имеют значения из этой строки, и 0 в противном случае. Каждое слагаемое является произведением всех входных переменных (если соответствующий элемент в рассматриваемой строке равен 1) или их отрицаний (если элемент равен 0). Например, слагаемое `xyz'` соответствует строке `1 1 0` и равно 1 в том и только в том случае, если `x`, `y` и `z` имеют значения 1, 1 и 0 соответственно. Очевидно, что сумма всех этих слагаемых дает искомую функцию[1].

Этот метод подходит для любой логической функции. Количество слагаемых равно количеству значений, для которых значение функции равно 1. В качестве очередного примера рассмотрим таблицу функции нечетности.

x	y	z	ODD	x'	y'	z'	x'y'z	x'yz'	xy'z'	xyz	xyz+xy'z'+x'yz'+x'y'z'
0	0	0	**0**	1	1	1	0	0	0	0	**0**
0	0	1	**1**	1	1	0	1	0	0	0	**1**
0	1	0	**1**	1	0	1	0	1	0	0	**1**
0	1	1	**0**	1	0	0	0	0	0	0	**0**
1	0	0	**1**	0	1	1	0	0	1	0	**1**
1	0	1	**0**	0	1	0	0	0	0	0	**0**
1	1	0	**0**	0	0	1	0	0	0	0	**0**
1	1	1	**1**	0	0	0	0	0	0	1	**1**

Вывод представления ODD(x, y, z) в виде ДНФ по таблице истинности

Эти и многие другие примеры встретятся нам в ходе разработки цифровых схем для вычислительных задач. А в завершение этого раздела мы приведем более формальный вариант только что описанной конструкции, выражающий фундаментальное свойство логических функций.

Теорема (Буль, 1847). Любая логическая функция может быть представлена в виде логической суммы произведений ее аргументов и их отрицаний, то есть в ДНФ.

Доказательство. См. текст выше на этой же странице.

[1] Полученный таким образом вид записи функции принято называть *совершенной дизъюнктивной нормальной формой — СДНФ*. СДНФ часто бывает избыточной и поэтому подвергается *минимизации*. — *Примеч. науч. ред.*

Булева логика закладывает теоретическую основу для построения цифровых схем, как будет показано позднее. В частности, представление функции в виде ДНФ играет чрезвычайно важную роль при построении цифровых схем для вычисления логических функций, а именно: использование такого представления сводит задачу построения цифровой схемы для *произвольной* логической функции к задаче построения схемы, реализующей AND, OR и NOT. После небольшого вводного курса в разделе 7.2 мы всерьез займемся этой темой в разделе 7.3.

Упражнения

7.1.1. Укажите в каждом пустом столбце таблицы всех логических функций двух переменных в тексте соответствующее логическое выражение (по аналогии с выражением `xy'`, приведенным в таблице для функции AND NOT).

7.1.2. Сохранится ли истинность законов де Моргана, если заменить OR операцией XOR? Докажите или приведите контрпример.

7.1.3. Приведите доказательство тождества `x + yz = (x + y)(x + z)` в виде таблицы истинности.

7.1.4. Приведите доказательство тождества `((m ⊕ k) ⊕ k) = m` в виде таблицы истинности.

7.1.5. Докажите законы тождества, аннулирования и ассоциативности для функции XOR по определению `x ⊕ y = xy' + x'y` и основных законов булевой алгебры (см. с. 953 и 956.

7.1.6. Используя приведенный в тексте ключ, приведите шифротекст, полученный в результате шифрования сообщения `A N S W E R` (тем же методом, что в тексте). Проверьте свой ответ — дешифруйте шифротекст с тем же ключом и тем же методом.

7.1.7. Докажите, что `MAJ(x, y, z) = xy + xz + yz`.

7.1.8. Приведите доказательство `MAJ(x, y, z) = (x + y + z)(x' + y + z)(x + y' + z)(x + y + z')` в виде таблицы истинности.

7.1.9. Покажите, что каждая логическая функция может быть представлена в виде *логического произведения сумм*, включающих каждую входную переменную или ее отрицание, как в примере из предыдущего упражнения. Такое логическое выражение называется *совершенной конъюнктивной нормальной формой* (*СКНФ*).

7.1.10. Приведите логическое выражение, эквивалентное `MAJ(w, x, y, z)`.

7.1.11. Приведите в виде ДНФ представление логической функции, которая равна 1, если `xy=z`, и равна 0 в противном случае.

7.1.12. Приведите в виде ДНФ представление логической функции с тремя аргументами, которая равна `x` для `z=0` или равна `y` для `z=1`.

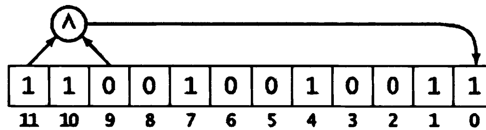
7.1.13. Приведите таблицу истинности для функции (x + y)(x + z)(y + z), а затем приведите более простое выражение для той же функции.

7.1.14. Докажите, что законы де Моргана обобщаются для *n* переменных. Конкретнее, докажите, что для любого положительного целого числа *n* выполняются следующие два условия:

$$(x_1 x_2 \ldots x_n)' = x_1' + x_2' + \ldots + x_n'$$
$$(x_1 + x_2 + \ldots + x_n)' = x_1' x_2' \ldots x_n'$$

Упражнения повышенной сложности

7.1.15. *LFSR*. Напишите программу Java, которая эмулирует абстрактный регистр сдвига с линейной обратной связью (LFSR, Linear Feedback Shift Register) для генерирования «случайной» последовательности битов. А именно ваша программа должна моделировать работу 12-разрядного регистра следующим образом.



Регистр на каждом шаге вычисляет результат операции XOR битов 11 и 9, помещает результат в бит 0 и выводит его в качестве очередного бита генерируемой последовательности, после чего сдвигает все биты на одну позицию влево. В своей программе используйте для представления битов символы 0 и 1, как в упражнении 7.1.18, и действуйте следующим образом: 11 битов передаются в строковом аргументе; возьмите целое число *n* из второго аргумента командной строки и создайте строку длины *n*, выполнив следующую операцию *n* – 11 раз: чтобы вычислить *i*-й бит (символ), примените операцию XOR к битам (символам) в позициях (*i* – 11) и (*i* – 9). Программа должна работать примерно так:

```
% java LFSR 11001001001 48
110010010011110110111001011010111001100010111111
```

Решение.

```
public class LFSR
{
   public static void main(String[] args)
   {
      String fill = args[0];
      int n = Integer.parseInt(args[1]);
      for (int i = 11; i < n; i++)
         if (fill.charAt(i-11) == fill.charAt(i-9))
             fill += "0";
```

```
            else fill += "1";
        StdOut.println(fill);
    }
}
```

7.1.16. *Эффективная реализация LFSR.* Решение, приведенное для упражнения 7.1.15, неэффективно для больших n, потому что оно выполняется за квадратичное время (как и любая программа, присоединяющая по одному символу к строке), а последовательность повторяется после $2^{11} - 1$ битов. Разработайте версию программы, которая решает вторую проблему, заменяя 11 на 63, а 9 на 62 (и получая 63 символа 0/1 в первом аргументе), а первую проблему — сохраняя только последние выведенные 63 бита.

7.1.17. *Таблицы истинности.* Напишите для любой заданной логической функции клиентское приложение, которое выводит таблицу истинности этой функции. Предполагается, что функция получает массив логических значений в своем единственном аргументе. *Подсказка*: см. `SATsolver` (листинг 5.5.1).

7.1.18. *Машина для шифрования/дешифрования.* Напишите программу Java `Crypto`, которая читает из стандартного ввода две строки одинаковой длины, состоящие из символов 0 и 1, интерпретирует эти символы как биты и записывает строку 0/1, соответствующую результату поразрядной операции XOR с двумя входными значениями. Другими словами, ваша программа должна работать следующим образом:

```
% java Crypto
010100110100010101000011010100100100010101010100
110010010011110110111001011010111001100010111111
100110100111100011111010001110011101110111101011
```

7.1.19. *Функционально полная система логических функций.* Множество элементарных логических функций называется *функционально полной системой*, или *функционально полным базисом*, если любая логическая функция может быть реализована с использованием только функций из этого множества. Конструкция с суммой произведений, ДНФ, описанная в тексте, показывает, что множество `{AND, OR, NOT}` является функционально полной системой (как и конструкция произведения сумм, КНФ, из упражнения 7.1.9). Предполагая, что NOT имеет всего один аргумент, а все остальные упоминаемые функции — два аргумента, покажите, что все множества элементарных функций из следующего списка, кроме одного, являются функционально полными системами. Докажите, что множество-исключение не является функционально полной системой.

a. NOT и AND

b. NOR

c. NAND

d. AND и OR

e. NOT и OR

f. AND и XOR

7.2. Базовая модель электронной схемы

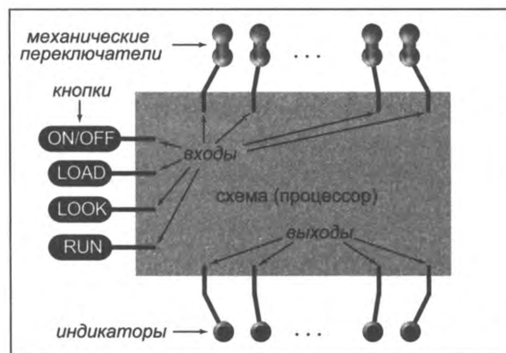
Чтобы понять, как строятся компьютеры, мы рассмотрим схемы, состоящие из компонентов трех простейших типов:

- проводники;
- подключения к источнику питания;
- управляемые переключатели.

Проводники используются для соединения с источником питания, передачи данных и соединения элементов схемы; управляемые переключатели могут создавать или прерывать соединения. Некоторые проводники назначаются *входными*, то есть используются для ввода исходных данных; другие могут назначаться *выходными*, то есть используются для вывода результатов[1]. В этом случае наша абстракция схемы определяется следующим образом.

Схема *представляет собой множество взаимосвязанных проводников, подключений к источнику питания и управляемых переключателей, которое преобразует значения на входах в значения на выходах.*

Модель имеет достаточно общий характер для описания любого вычислительного устройства. Например, с ее помощью можно описать поведение нашего компьютера TOY из главы 6: процессор машины TOY представляет собой схему, входы которой соединяются с механическими переключателями и кнопками на передней панели, а выходы — с индикаторами на передней панели. Требуется спроектировать схему с индикаторами, которые должны включаться для любой заданной истории изменений состояния переключателей и нажатий кнопок.



Упрощенная модель компьютера

У элементов модели существуют различные физические состояния, которые мы интерпретируем как аналоги дискретных двоичных значений. Переключатели

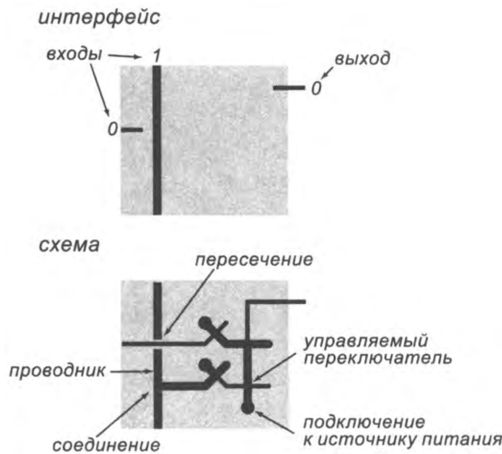
[1] Далее для краткости «входы» и «выходы». — *Примеч. пер.*

и индикаторы находятся либо во включенном, либо в выключенном состоянии; проводник либо соединен с источником питания, либо не соединен. Изменения в состоянии соответствуют перемещению информации по схеме.

Чтобы связать модель с реальным миром, мы воспользуемся двумерным геометрическим представлением. Проводники изображаются отрезками прямой на плоскости; управляемые переключатели — проводниками, пересекающимися определенным образом; схемы элементов и узлов — проводниками в прямоугольнике. Чтобы показать схему, мы изображаем прямоугольник, обозначающий ее границы. Входные проводники начинаются от границы схемы; выходные выходят за ее границу. Если подробности реализации схемы нас не интересуют, то изображается только ее *интерфейс* (прямоугольник, входные проводники, выходные проводники и подписи), как показано на иллюстрации внизу.

Использование явного геометрического представления вместо абстракции, отделенной от какой-либо геометрической структуры, приводит к некоторой потере гибкости, но зато мы сразу приходим к конечному результату — конкретному представлению любой схемы в виде чертежа с пересекающимися линиями. Этот момент важен, потому что схемы в наши дни могут быть изготовлены непосредственно по таким чертежам.

Уровень детализации при объединении схем необходимо тщательно продумать. С абстрактной точки зрения эти подробности несущественны, но с практической точки зрения необходимо соблюдать простые соглашения — своего рода аналоги «правил проектирования», управляющих построением реальных больших интегральных схем (БИС), лежащих в основе современных процессоров. Возможно, после рассмотрения нескольких схем вам стоит перечитать этот раздел и убедиться в том, что вы понимаете базовые определения.



Представление схемы и основные термины

Проводники

Схемы состоят из проводников, которые соединяются друг с другом, источником питания или *управляемым переключателем*. Управляемый переключатель представляет собой пересечение двух проводов, один из которых завершается сразу же после пересечения. Работа управляемых переключателей подробно рассматривается далее. Проводники представляются отрезками, обычно горизонтальными или вертикальными (иногда диагональными). Они могут пересекаться (проходить выше или ниже друг друга) или соединяться. Мы будем предполагать, что каждый проводник всегда находится в одном из двух состояний, что позволяет интерпретировать состояние проводника как двоичное значение (0 или 1). Соединенные проводники всегда имеют одинаковое значение. Если мы захотим отследить значения проводников в схеме, проводники в состоянии 1 будут обозначаться толстыми линиями, а проводники в состоянии 0 — тонкими линиями.

Подключения к источнику питания

Чтобы не загромождать чертеж, мы будем предполагать, что один из входов схемы всегда равен 1, и мы будем обозначать его жирной точкой как подключение к источнику питания. В реальной интегральной схеме такие точки могут представлять собой межслойное соединение — подключение к отдельному слою питания. Проводник, подключенный к источнику питания, находится в состоянии 1, пока это подключение сохраняется. Любой проводник, соединенный с проводником в состоянии 1, также находится в состоянии 1. На следующей странице мы опишем, как управляемый переключатель может разорвать соединение, в результате чего разомкнутый проводник переходит в состояние 0. Проводник, соединенный с входом схемы в состоянии 0 или с любым другим проводником в состоянии 0, также находится в состоянии 0[1].

Входы

Любое устройство ввода на компьютере передает схеме набор дискретных значений. Когда вы нажимаете клавишу на клавиатуре, значение символа Юникода, соответствующего нажатой клавише, передается на вход процессора вашего компьютера[2]; если вы проводите по сенсорному экрану или сенсорной панели, процессору передаются двоичные числа, описывающие относительное смещение; переключатели TOY напрямую соответствуют двоичным значениям и т. д. Изменения во входных значениях приводят к изменениям в состоянии проводников и переклю-

---

[1] Подразумевается, что любой проводник соединен не более чем с одним источником сигнала. Непосредственное соединение выходов двух источников (так называемые *монтажные* И либо ИЛИ) и возможные при этом конфликты значений сигнала не рассматриваются. — *Примеч. науч. ред.*

[2] Точнее, клавиатура персонального компьютера передает так называемый *scan-код* нажатой клавиши, а затем процессор (программа) преобразует его в значение символа в нужной кодировке. — *Примеч. науч. ред.*

чателей внутри схемы, а в конечном итоге к изменению выходных значений. Для простоты будем считать, что изменения состояния в схемах происходят намного быстрее внешних изменений — когда вы нажимаете клавишу на клавиатуре своего компьютера или щелкаете переключателем TOY, можно ожидать, что компьютер отреагирует немедленно.

Выходы

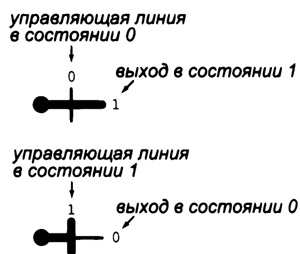
Состояния выходов могут интерпретироваться любым устройством, внешним по отношению к схеме. Появление значения, соответствующего символу Юникода, как набора выходных значений на вашем компьютере может привести к тому, что этот символ будет напечатан принтером; передача координат и цветового значения как другого набора выходных значений может привести к появлению точки заданного цвета на экране монитора; индикаторы TOY напрямую соответствуют двоичным выходам и т. д. Чаще всего выходы наших схем являются входами для других схем.

Соглашения

Наша модель передачи и получения информации от схем основана на простой идентификации некоторых проводников как входных или выходных. В геометрическом представлении эти входы и выходы просто должны находиться на границах схемы, где они могут соединяться с внешними устройствами. Обозначения интуитивно понятны: на наших чертежах любой проводник, который только соприкасается с границей схемы, является входом, а любой проводник, заметно заходящий за границу, является выходом. Входные проводники представляют набор двоичных значений, которые создаются за пределами схемы и передаются ей для обработки; выходные проводники представляют набор двоичных значений, которые вычисляются схемой и передаются наружу для последующей обработки, сохранения или отображения. Обычно мы стараемся размещать входы на верхней и левой стороне схемы, а выходы — на правой и нижней стороне. Это соглашение всего лишь помогает понять предназначение проводов. В ряде случае входы и другие проводники могут проходить через всю схему по вертикали или горизонтали.

Управляемые переключатели

Ключевым моментом для понимания наших схем становится понимание принципа действия *управляемых переключателей* — тех точек, где *управляющая линия переключателя* пересекается с другим проводником и завершается. Изменение в состоянии управляющей линии может, например, разомкнуть соединение проводника с источником питания, тем самым изменяя его состояние.



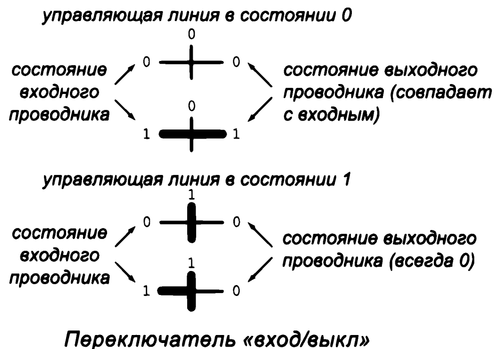
Переключатель «вкл/выкл»

Переключатели «вкл/выкл»

Многие переключатели представляют собой проводник, соединенный с 1, который пересекается с управляющей линией переключения. Если управляющая линия находится в состоянии 0, она ни на что не влияет. Но если линия находится в состоянии 1, это приводит к разрыву соединения, а состояние проводника на другой стороне соединения меняется с 1 на 0[1]. Этот проводник становится выходом переключателя. Таким образом, если управляющая линия переключателя находится в состоянии 0, то выходной проводник находится в состоянии 1; если управляющая линия находится в состоянии 1, то выходной проводник находится в состоянии 0.

Переключатели «вход/выкл»

В более широком смысле управляемый переключатель может рассматриваться как имеющий входное значение (не обязательно 1). Вход и выход соединяются, образуя прямую линию, а управляющая линия входит в точку пересечения. С логических позиций принцип работы переключателя прост: если управляющая линия находится в состоянии 0, то входной и выходной проводники соединены и поэтому имеют одинаковое значение (0 или 1); если управляющая линия находится в состоянии 1, то входной и выходной проводники разъединены, и, следовательно, значение выходного проводника равно 0 (независимо от состояния входного проводника). Таким образом, управляющая линия может рассматриваться как механизм автоматического размыкания соединения входа с выходом (по включению управляющей линии). Как вы увидите, эта простая возможность управления соединением служит основой для построения довольно сложных схем. Эти схемы используются для построения компьютеров и других электронных устройств.



Переключатель «вход/выкл»

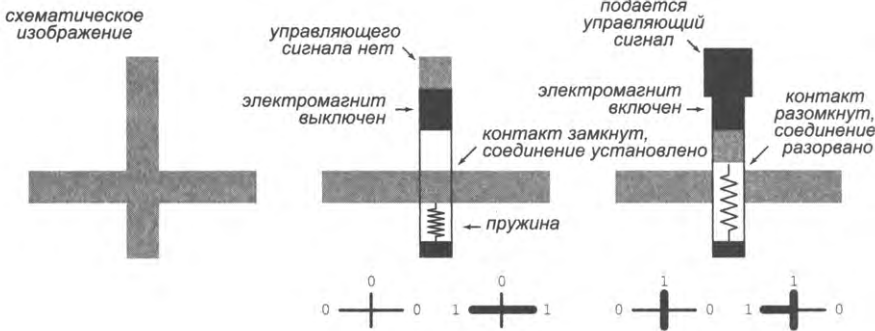
[1] Здесь и далее используемая модель переключателя наиболее соответствует ключу на полевом транзисторе со «встроенным» проводящим каналом. Заметим, что во всех описываемых схемах выходное значение 0 физически не гарантируется, так как управляемое соединение с «землей» не предусмотрено. — *Примеч. науч. ред.*

Соглашения

Вход управляемого переключателя может быть входом схемы или выходом другого переключателя. Выход управляемого переключателя может быть управляющей линией, или входом другого управляемого переключателя, или же выходом схемы. Мы не обозначаем явно вход и выход при изображении управляемых переключателей. Во всех наших схемах это различие очевидно, потому что входом всегда является проводник, соединенный с источником питания или выходом другого переключателя, причем входы обычно располагаются у левой или верхней стороны схемы, а выходы — у правой или нижней стороны.

Физический пример

Как построить управляемый переключатель? Чтобы дать вам некоторое представление о процессе, мы рассмотрим устройство, называемое *реле*. В конструкции реле управляющая линия подключена к электромагниту, притягивающему подвижный контакт (небольшой отрезок проводника), который может соединять входной проводник с выходным и к которому также присоединена пружина. Если электромагнит выключен, то контакт под действием пружины соединяет вход с выходом; если электромагнит включен, то его воздействие превышает усилие пружины, контакт отходит и разрывает соединение между входом и выходом. Реле такого рода продолжают использоваться во многих физических устройствах, от выключателей в тостерах и радиоприемниках до звуковых сигналов и дверных звонков.



Строение реле (управляемого переключателя)

В современных компьютерах механические реле не применяются, потому что они слишком велики, слишком медленны, слишком дороги и недостаточно эффективны для соединения миллионов и миллиардов проводников. В наше время в компьютерах на физическом уровне функции переключателей чаще всего выполняют крошечные электронные приборы — транзисторы. Некоторые транзисторы на самом деле представляют собой пару пересекающихся проводников, изготовленных из разных материалов, как на наших иллюстрациях. Ранние компьютеры строились с другими типами переключателей: электронными лампами, реле и иными прибора-

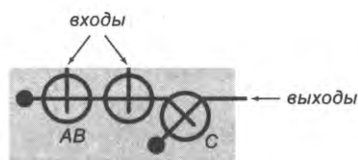
ми. Почти с самого момента изобретения компьютера потребность в создании более быстрых, компактных и дешевых компьютеров в значительной мере сводилась к созданию более быстрых, компактных и эффективных переключателей.

Схемы

Чтобы построить электронную схему, мы соединяем входы с управляющими линиями переключателей, а выходы переключателей — с другими управляющими линиями или выходами. Изменения в состоянии управляющих линий распространяются по схемам и реализуют вычисления. Эти изменения могут быть достаточно сложными, поэтому каждое соединение необходимо тщательно продумать.

Анализ переключений

Чтобы понять логику работы схем, необходимо понять, что делают управляемые переключатели и в каких местах схемы управляющие линии пересекаются с другими проводниками. Как упоминалось выше, изменение состояния управляющей линии может приводить к изменению состояния на выходе переключателя.



на обоих входах 0

на одном или на обоих входах 1

Анализ переключений для схемы (логический элемент OR)

Для примера рассмотрим схему, изображенную справа. Схема состоит из трех переключателей A, B и C и реализует *логический элемент* (или *вентиль*) OR — один из основных элементов, которые мы вскоре рассмотрим более подробно. Значение на выходе этого элемента является результатом применения к двум входным значениям операции OR. Чтобы понять, как это происходит, следует проанализировать реакцию переключателей на все возможные комбинации входных значений. Когда переключатель управляется входными значениями, мы знаем выходное состояние этого переключателя. Если его выход управляет другим переключателем, мы знаем выходное значение другого переключателя и т. д.

Первой изображена ситуация, когда оба входных значения равны 0. В этом случае у переключателей A и B выход соединен с входом схемы, поэтому на вход C подается значение 1. Тогда соединение переключателя C с источником питания прерывается, поэтому на выход передается 0. Во всех остальных случаях переключатель A и/или переключатель B разрывает соединение с источником питания слева, на управляющую линию переключателя C подается 0, соединение с источником пи-

тания не прерывается, и на выход передается 1. Честно говоря, анализ такого рода получается довольно запутанным; к счастью, его необходимо проделать только для небольшого количества очень простых схем (таких, как эта).

Комбинационные схемы

Между разными типами схем существует принципиальное отличие, неочевидное из базовых определений. Это отличие связано с наличием обратных связей в схеме. Как правило, если схема не содержит обратных связей, анализ, подобный приведенному выше, всегда приводит к однозначным выходным значениям. *Комбинационная* схема не содержит обратных связей и, как следствие, обладает тем свойством, что ее выходные значения зависят только от входных значений, но не от порядка их представления. Более интересный пример комбинационной схемы — *сумматор* — получает $2n$ входных значений, представляющих два n-разрядных двоичных числа, и выдает $n + 1$ выходных значений, представляющих их сумму. При любом порядке подачи входных значений мы должны получить одну и ту же сумму (после короткой задержки, необходимой для распространения значений в схеме). Полная схема сумматора будет рассмотрена в разделе 7.3.

Последовательностные схемы

С другой стороны, *последовательностная схема* (называемая также *схемой с памятью*) может содержать обратные связи, соответственно у многих таких схем выходные значения зависят от временной последовательности входных значений. Пример последовательностной схемы — *счетчик* с одним входом и n выходами, содержащий двоичное представление количества изменений состояния входа с 0 на 1 или наоборот. Отличительная особенность последовательностных схем состоит в том, что состояние их внутренних элементов зависит не только от текущего состояния входов, но и от прошлых состояний. Вы лучше поймете суть этого отличия после того, как мы рассмотрим несколько конкретных примеров. Комбинационные схемы проще последовательностных, поэтому они будут подробно рассмотрены в разделе 7.3, прежде чем мы займемся последовательностными схемами в разделе 7.4.

В этой главе мы сначала построим несколько небольших структурных элементов вроде только что описанного. Затем мы используем интерфейсы к этим схемам для построения нескольких более сложных модулей. Наконец, использование интерфейсов к этим модулям позволит нам построить полноценный процессор. Интересно, что для перехода от переключателя к процессору достаточно всего двух промежуточных уровней абстракции!

Логическое проектирование и реальный мир

Наша модель ориентирована на логическое строение процессора, а не на физический компьютер. Модель ничего не говорит о том, сколько будет весить такая машина, из каких материалов она будет изготовлена, какую мощность она будет потреблять, в какой цвет она будет покрашена и какими физическими свойствами

будет обладать. Она даже не требует, чтобы схемы были электрическими. Например, можно без особого труда представить управляемые переключатели в схемах, построенных из труб, по которым подаются вода или сжатый воздух[1]. Такие устройства интересно придумывать, причем размышления над ними не так уж далеки от размышлений над новыми технологиями транзисторов.

Разнообразие типов переключателей, которые использовались для построения компьютеров, свидетельствует о том, насколько эффективна абстракция управляемых переключателей. Исследователи продолжают искать возможности для существенных улучшений в вычислительных технологиях, просто изобретая усовершенствованный переключатель.

Безусловно, в реальном мире приходится учитывать различия между разными типами переключателей и проводников, поэтому наша абстракция станет лишь отправной точкой. Например, мы не учитываем в полной мере такие факторы, как мощность, необходимая для управления переключателями; время, необходимое переключателю для разрыва соединения, или физический размер переключателей. Если вы действительно хотите производить компьютеры, вам придется принять во внимание все эти (и многие другие) факторы.

Для каждого физического проводника и переключателя нужно будет найти подходящее место, и это еще один аспект, с которым безусловно придется считаться в реальном мире. Мы имеем это в виду с самого начала, задавая расположение каждого элемента. Представьте, что каждая схема, которую мы разрабатываем, будет соответствовать физическому устройству, от которого отходят проводники; чтобы соединить их, необходимо определиться с физическим расположением входов и выходов. В современных схемах больше свободы для выбора, но в конечном итоге все должно занять какое-то определенное место. Мы не учитываем в полной мере такие факторы, как толщина провода, пространство между проводами или свойства пересечений, но, если потребуется, наш подход можно легко расширить.

Несмотря на свою простоту, абстракции проводника, источника питания, управляемого переключателя и схемы весьма полезны. Они обладают достаточной мощью, чтобы их можно было применить для проектирования машин произвольной сложности. Совместно они представляют очень узкий «мостик» между реальным миром и абстрактным миром вычислений, который дает нам возможность пользоваться технологическими усовершенствованиями для улучшения всех этих машин.

Вопросы и ответы

В. Что вы понимаете под термином «переключатель» — нечто абстрактное, что может находиться в одном из двух состояний, или реальное устройство вроде того, которым мы пользуемся, чтобы включать свет?

---

[1] Гидравлические цифровые вычислительные устройства реально изготавливались и использовались. — *Примеч. науч. ред.*

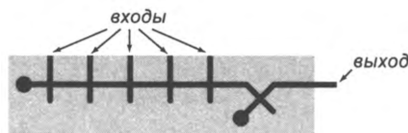
О. И то и другое. Переключатели и проводники — основные абстрактные структурные элементы, используемые для проектирования схем, и они же являются физическими компонентами компьютеров. С точки зрения компьютерных технологий способ изготовления переключателей непринципиален; анализ схем превращается в анализ групп взаимосвязанных переключателей. Мы уже рассматривали две разные физические реализации переключателей: переключатели на передней панели компьютера TOY и реализация управляемых переключателей в наших схемах на базе пересечений проводников. С точки зрения ученого или инженера, переключатели исключительно важны, потому что изобретение новых видов переключателей в реальном мире может привести к новым знаниям или дать толчок технологиям. Переключатели напрямую связаны со всем, что мы знаем о вычислениях. Переключателем может быть реле, или транзистор, или какой-либо объект реального мира: фрагмент генетического материала, нейрон, молекула или «черная дыра».

В. А почему «цепь»? Это, кажется, обычно связано с замкнутым контуром, а мы только присоединяем к ней что-нибудь?

О. Как и «проводник», термин «цепь» пришел из электрических цепей, которые действительно всегда замыкаются через источник питания. Чтобы зажечь лампочку, нам нужно получить замкнутый контур, поэтому и электрическая вилка для подключения ее к сети имеет два штыря. Но подробности функционирования электрических цепей для нашей модели не так важны, и вам не обязательно полностью знать электротехнику, чтобы понимать эту модель. Конечно, значение электротехники как основы работы компьютера бесспорно, и понимание таких аналогий будет, конечно, полезным, но в конечном итоге вам достаточно следовать нашим упрощенным формальным правилам, чтобы определить, при каком наборе входных данных (при каком состоянии физических переключателей) какая выходная линия окажется подключенной к источнику питания и тем самым зажжет соответствующий индикатор.

Упражнения

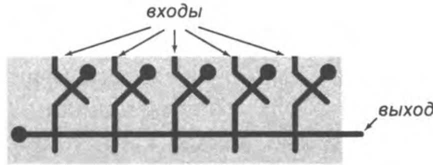
7.2.1. При каких условиях на выход следующей схемы выдается 0?



Решение. В том и только в том случае, если на все входы подается 0.

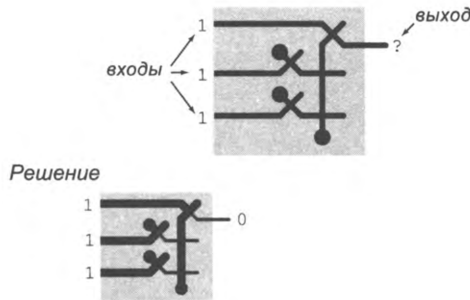
7.2.2. Укажите условия, при которых на выход в упражнении 7.1.1 выдается 1.

7.2.3. При каких условиях на выход следующей схемы выдается 1?



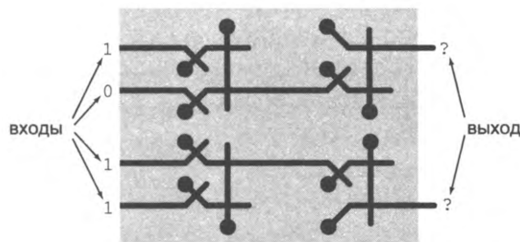
7.2.4. Укажите условия, при которых на выход в упражнении 7.1.3 выдается 0.

7.2.5. Обведите проводники, находящиеся в состоянии 1, и укажите выход следующей схемы.



Примечание: в наших схемах выходные проводники в состоянии 1 всегда считаем подключенными к источнику питания.

7.2.6. Обведите проводники, находящиеся в состоянии 1, и укажите выход следующей схемы.



7.3. Комбинационные схемы

Многие вычислительные задачи сводятся к нахождению значений математических функций для заданных входных значений. В этом разделе мы займемся построением схем для таких задач. Например, мы знаем, что любому компьютеру необходима схема, которая складывает два двоичных числа, — как построить такую схему?

При поиске ответа на этот вопрос мы с большой вероятностью обнаружим два неожиданных обстоятельства. Первая неожиданность заключается в том, что существует простой систематический метод для построения схем, способных вычислить любую логическую функцию, которая имеет явное определение. Этот метод будет описан позднее в этом разделе. Но когда вы его изучите, станет очевидно, что он работает только с небольшим количеством входов, потому что в противном случае придется использовать слишком много переключателей и проводников. Как и в случае с программами, необходимо учитывать не только теоретическую возможность построения схем для необходимых функций, но и ее практическую реализуемость в реальном мире. Вторая неожиданность заключается в том, что схемы, необходимые для реализаций функций, которые необходимы для такого компьютера, как TOY, могут быть разработаны с небольшим количеством уровней абстракции, и они достаточно просты для понимания. Этот раздел посвящен изучению важного класса таких схем, а именно: мы рассматриваем задачу построения схем, соответствующих логическим функциям. Такие схемы, отличительное свойство которых заключается в том, что их выходные значения зависят только от входных значений, называются *комбинационными схемами*. Комбинационные схемы являются исключительно важной отправной точкой для понимания работы вашего компьютера.

Мы начнем с базовых конструкций для создания *логических элементов* (*вентилей*, gates) — схем, реализующих базовые логические функции NOT (НЕ), NOR (ИЛИ-НЕ), OR (ИЛИ) и AND (И). Затем мы перейдем на более высокий уровень абстракции и рассмотрим схемы для разных (более сложных) логических функций, состоящие из базовых элементов. После этого будет рассмотрена общая конструкция, которая преобразует таблицу истинности произвольной логической функции в реализующую ее схему. Поднимаясь на более высокий уровень абстракции, мы ответим на только что поставленный вопрос, рассмотрев построенную из таких компонентов схему для суммирования двух двоичных чисел. В этом разделе мы определим набор интуитивно понятных правил для модульного конструирования схем: соединения простых схем и использования их для построения более сложных схем на более высоких уровнях абстракции.

Логические элементы

Как было показано в разделе 7.1, булева логика представляет собой абстрактную математическую систему, которая играла центральную роль в системе математического мышления в течение почти двух столетий. Но как она связана с построением компьютеров? Эту глубинную связь выявил Клод Шеннон (Claude Shannon), когда он был студентом Массачусетского технологического института в 1930-х годах. Шеннон понял, что считавшийся сугубо теоретическим и формальным аппарат булевой алгебры *напрямую* применим к компьютерам, потому что он позволяет строить *цифровые* схемы, реализующие логические функции. В то время некоторые люди (к числу которых относился и научный руководитель

Шеннона) рассматривали возможность построения компьютеров на базе *аналоговых* схем, с представлением вещественных чисел уровнями напряжения[1]; другие работали над цифровыми схемами, но без формального базиса, предоставляемого булевой логикой.

Следуя рассуждениям Шеннона (как это делали все с момента публикации его работы), мы поднимаемся в своей модели схемы на следующий уровень абстракции, на котором из проводников и переключателей строятся небольшие устройства, называемые *логическими элементами*, или *вентилями* (gates). В частности, мы построим элементы, реализующие логические функции AND, OR, NOR и NOT, включая их версии с несколькими аргументами. Как вы увидите, логические элементы достаточно просты, чтобы их можно было построить из нескольких переключателей, но при этом достаточно функциональны, чтобы их можно было использовать для построения любого вычислительного устройства.

Клод Шеннон (1916–2001)

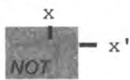
Как и все схемы, каждый вентиль будет изображаться заключенным в маленький прямоугольник и состоять из входных проводников (доходящих до границы прямоугольника), выходных проводников (которые заходят за границу прямоугольника) и переключателей, соединенных другими проводниками. Все соединения в логических элементах просты: входы управляют переключателями, и, если входные значения не изменяют состояния, также не изменяется и состояние на выходе. Но если входное состояние изменяется, то состояние на выходе изменяется по прошествии короткого времени, называемого *задержкой переключения*. Другими словами, логические элементы представляют собой простые схемы, у которых состояние выходов по прошествии незначительного времени является элементарными логическими функциями.

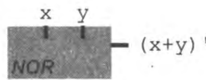
Логический элемент NOT (НЕ)

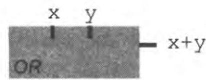
Возможно, вы уже заметили, что переключатель «вкл/выкл» является логическим элементом, реализующим функцию отрицания NOT (если рассматривать управляющую линию как вход). Если на вход подается 0, то на выход поступает 1; если на вход подается 1, то на выход подается 0. Таким образом, для входного значения x выходное значение равно x'. В дальнейшем мы будем называть переключатели «вкл/выкл» логическими элементами NOT. Два разных случая работы NOT изображены на следующей странице в левой части рисунка. Логические элементы NOT также называют *инверторами*.

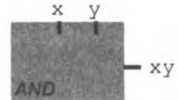
[1] Аналоговые электронные вычислительные машины реально создавались и достаточно долго использовались в различных областях. — *Примеч. науч. ред.*

интерфейсы

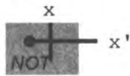


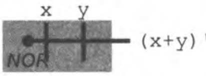


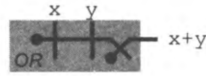


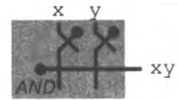


схемы

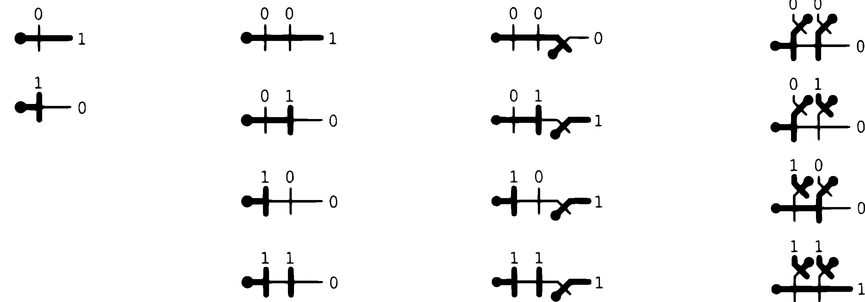








анализ переключений



Логические элементы NOT, NOR, OR и AND, построенные из переключателей

Логический элемент NOR (ИЛИ-НЕ)

В результате соединения выхода переключателя «вкл/выкл» с входом переключателя «вход/выкл» мы получаем логический элемент, выход которого равен 1 в том и только в том случае, если оба входа (управляющие линии переключения) равны 0, как показано во втором столбце. Выходной проводник соединен с 1 в том и только в том случае, если на оба входа подается 0, потому что если хотя бы один вход находится в состоянии 1, соединение будет прервано. Сверяясь с таблицей истинности, мы видим, что на выход передается логическая функция NOR.

Логический элемент OR (ИЛИ)

Чтобы вычислить функцию OR, достаточно инвертировать NOR, поэтому вентиль OR создается объединением элемента NOR с элементом NOT. Выход равен 0 в том и только в том случае, если оба входа находятся в состоянии 1. Вы также можете проанализировать подробности операции переключения в третьем столбце, но это простое объяснение более понятно.

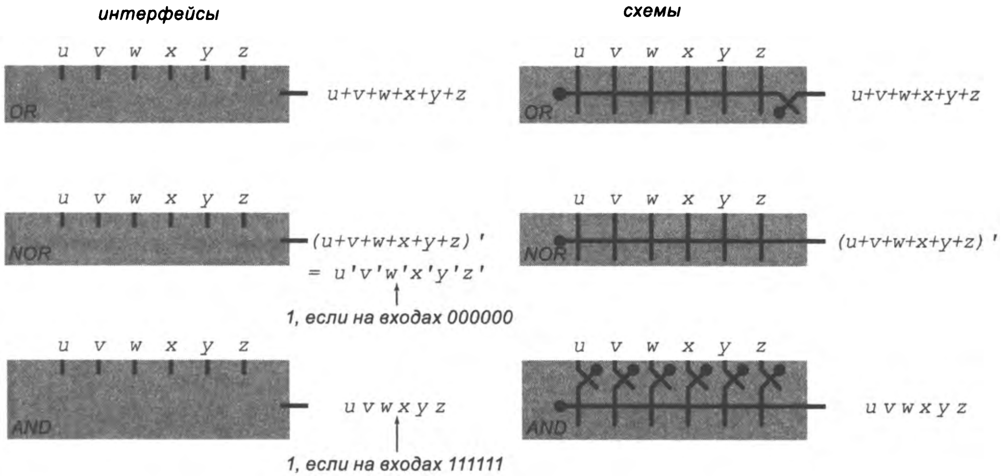
Логический элемент AND (И)

Законы де Моргана дают основу для построения логического элемента AND на базе элемента NOR. Оба входа инвертируются. Если x и y — входы, то вентиль вы-

числяет `(x' + y')' = xy`. Выход равен 1 в том и только в том случае, если оба входа находятся в состоянии 1. И снова вы можете убедиться в этом, проанализировав работу переключателей в четвертом столбце, но проще воспользоваться булевой алгеброй.

Многовходовые логические элементы

Все наши конструкции для логических элементов NOR, OR и AND естественным образом расширяются для любого количества входов. Чтобы построить элемент NOR с n входами, мы выстраиваем цепочку из n переключателей (переключатель «вкл/выкл», за которым следуют $n - 1$ переключателей «вход/выкл»): выход последнего является результатом применения NOR к входам. Как и в случае с двухвходовыми логическими элементами, элемент OR строится из элемента NOR инвертированием выхода, а элемент AND — из элемента NOR инвертированием входов.



Многовходовые логические элементы OR, NOR и AND, построенные из переключателей

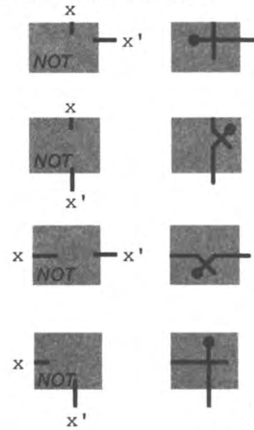
Обратите внимание: согласно обобщенному закону де Моргана (см. упражнение 7.1.14), значение на выходе логического элемента NOR может формироваться двумя способами.

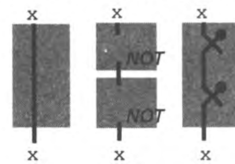
Геометрия, повороты, отражения и двойное инвертирование

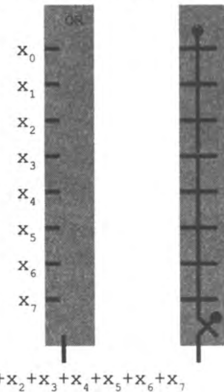
На этой стадии стоит кратко припомнить наши соглашения, относящиеся к геометрии схем. С одной стороны, в любой момент времени следует работать на подходящем уровне абстракции. Например, компонентами высокоуровневых схем становятся не другие схемы как таковые, а лишь интерфейсы. С другой стороны, мы

хотим показать, что схема имеет вполне конкретное строение, поэтому мы будем регулярно «поднимать капот» и показывать, из каких компонентов строится схема, — даже на высоких уровнях абстракции. Одно из следствий такого подхода заключается в том, что во всех наших проектах будут учитываться геометрические аспекты, даже если это не обязательно. Например, все наши логические элементы AND, OR и NOT изображаются в виде прямоугольников со слегка различающимися размерами, тогда как при традиционном подходе к проектированию схем используются блоки разной формы, но одинакового размера[1].

Как следствие, при построении более сложных схем иногда требуется поворачивать, переворачивать или растягивать элементы. Некоторые примеры показаны на иллюстрации справа. В первой группе примеров представлены четыре разных способа реализации логических элементов NOT в рамках наших соглашений: вход может располагаться сверху или слева, а выход — справа или внизу. Второй пример показывает, что соединение выхода одного элемента NOT с входом другого эквивалентно отсутствию операции. Этот пример наглядно свидетельствует о том, что мы работаем не с физическими, а с логическими соединениями между входами и выходами. Для пары последовательно соединенных элементов NOT значение на выходном проводнике равно значению на входном проводнике, хотя явного физического соединения между ними нет. На нижнем примере изображен многовходовый логический элемент OR, развернутый вертикально. Элементы такого типа достаточно часто будут встречаться в наших схемах — они принимают значения, которые могут поступать от нескольких источников.







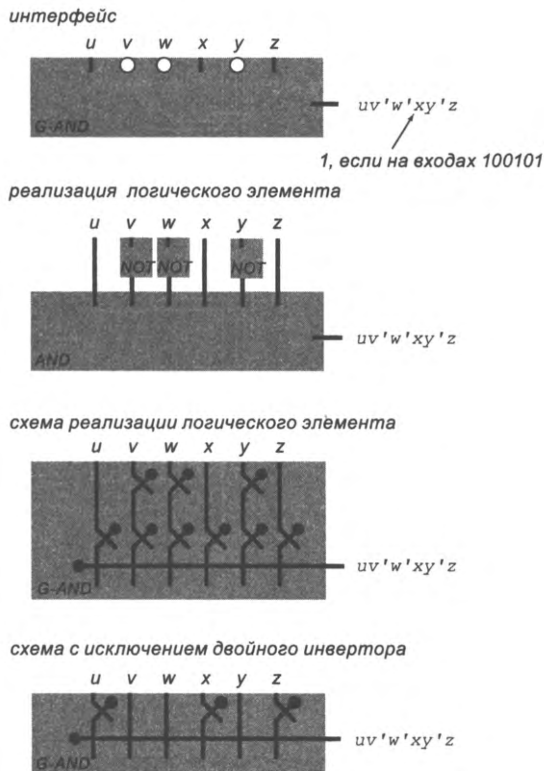
Примеры эквивалентности схем (слева: интерфейсы, справа: схемы)

[1] Имеются в виду зарубежные схемотехнические нотации. В отечественной практике логические элементы и микросхемы на принципиальных схемах изображались прямоугольниками, функции элемента и его отдельных выводов обозначались соответствующими символами, а размер прямоугольника зависел от числа выводов. На функциональных схемах использовались обозначения разной формы в зависимости от функций элементов. — *Примеч. науч. ред.*

Конечно, перемещение входов/выходов с верхней/правой стороны на левую/нижнюю, исключение двойных элементов NOT и повороты не влияют на значения, выдаваемые схемами в ответ на заданные входные наборы. Мы упоминаем эти вариации, чтобы избежать путаницы, когда вы столкнетесь с ними в больших схемах. И хотя поворот схемы на 180° никак не повлияет на ее работу, мы всегда следим за тем, чтобы входы располагались слева или наверху, а выходы — справа или внизу. Это делается по единственной причине: чтобы вам было проще понять, что делают схемы.

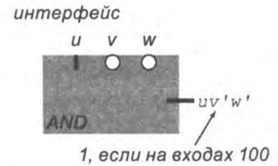
Обобщенные многовходовые логические элементы

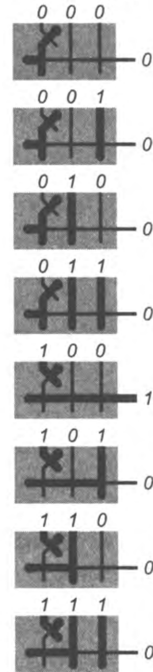
Обобщая идею многовходовых логических элементов, мы можем инвертировать только часть входов, чтобы получить обобщенные логические элементы AND для вычисления функций вида `uv'w'xy'z`. Такие элементы исключительно полезны; кроме того, они заслуживают вашего внимания как пример низкоуровневого анализа и реализации комбинационных схем.



Обобщенный логический элемент AND (пример)

Для начала заметим, что многовходовый логический элемент AND на с. 977 обладает одним важным свойством: *существует только один набор входных значений, для которого выходное значение равно 1*. Если инвертировать часть выходов, при сохранении этого свойства можно вычислить любую функцию, выражаемую как результат операции AND нескольких аргументов.





Анализ переключений для обобщенного логического элемента AND

На с. 979 приведен пример интерфейса, используемого для таких логических элементов, — интерфейс обобщенного элемента AND для вычисления функции `uv'w'xy'z`. Правила просты: кружок вместо линии на входе означает, что вход инвертируется. Вторая сверху иллюстрация показывает, как такие схемы реализуются на базе элементов NOT и AND, а на третьей раскрыта внутренняя реализация элементов. Как видите, любой инвертированный вход соответствует двойному элементу NOT, поэтому все их можно убрать, и у вас остается компактная реализация схемы, изображенная слева внизу (с. 979). Если вы не уверены в том, как работает эта схема, обратитесь к иллюстрации справа — на ней представлен анализ переключений для всех возможных вариантов входных данных обобщенного логического элемента AND с тремя входами, вычисляющего функцию `uv'w'`.

Такие вентили отличаются компактностью и гибкостью, и на их основе будут строиться схемы, которые мы будем рассматривать в оставшейся части этого раздела. Если вы все еще не уверены в том, как они работают, перечитайте этот раздел позднее. Однако учтите, что конкретная функция таких логических элементов легко может быть прочитана по их интерфейсу: для каждого входа его значение, необходимое для получения 1 на выходе, показано непосредственно в точке входа (отрезок обозначает 1, а кружок — 0)[1].

Логические элементы предоставляют более высокий уровень абстракции, чем переключатели. По сути, это качественное изменение, которое позволяет нам игнорировать подробности работы с переключателями и работать на уровне булевой логики. Для ясности и удобства мы рассмотрели конструкции для элементов NOT,

[1] Иными словами, в символике на входах «читается» единственный входной набор, обеспечивающий для функции AND выходную 1. Для функции OR, очевидно, обозначения входов будут символизировать *инверсию* единственного набора, обеспечивающего выходной 0. — *Примеч. науч. ред.*

NOR, AND и OR и их обобщения для нескольких входов, но на самом деле с учетом того, что элемент NOT представляет собой NOR с одним входом, для построения всех остальных элементов можно ограничиться только элементами NOR.

Наше применение управляемых переключателей для построения логических элементов в действительности не использует их возможности в полной мере: каждый переключатель либо выполняет функции инвертора, либо управляет соединением между 1 и выходным проводником. Можно сконструировать элементы с меньшим количеством переключателей, но наше консервативное решение является уступкой реальному миру: его реализация проще, когда речь идет об изготовлении миллионов и миллиардов переключателей. Например, каждый выходной проводник связывается с питанием внутри своего элемента.

Разумно (всегда) спросить себя: что мы теряем от перехода на более высокий уровень абстракции? Априорно мы этого знать не можем: управляемые переключатели могут соединяться между собой множеством разных конфигураций, разобраться в поведении которых может быть достаточно сложно. Однако в данном случае можно легко воспользоваться булевой алгеброй, чтобы доказать, что логические элементы могут использоваться для построения любых схем, которые строятся на базе управляемых переключателей, и наоборот.

Также разумно (всегда) подумать и над тем, что мы приобретаем от перехода на более высокий уровень абстракции. В данном случае анализ схемы упрощается и становится более систематичным, но еще важнее другое: этот уровень абстракции принципиален тем, что он отделяет физический мир от абстрактного.

Переход к использованию логических элементов позволяет отвлечься от особенностей поведения конкретных сложных физических устройств. Возможно, вы перейдете на другой тип пружин, на более мощные магниты или что-нибудь в этом роде. А может, вы выберете совершенно иную реализацию элементов, в которой переключатели вообще не используются. Эта возможность полного изменения физической реализации на столь низком уровне была фундаментальной движущей силой прогресса в вычислительных технологиях почти с первых дней их существования. Логические элементы делались на базе реле, электронных ламп, транзисторов и множеством других физических способов. Мы уже рассмотрели один из способов улучшения всего — построение улучшенного переключателя; другой способ заключается в создании улучшенного элемента NOR (см. упражнение 7.2.2).

Если же рассматривать ситуацию на более высоком уровне, переход к использованию логических элементов связывает наши схемы с булевой алгеброй — полностью разработанной математической системой, надежность которой не вызывает сомнений. В этом и заключалось озарение Шеннона. Формальные математические утверждения относительно свойств логических функций также помогают строить цифровые схемы, поведение которых нам полностью понятно.

Каждый логический элемент сам по себе является схемой, поэтому мы непосредственно приходим к полезному рекурсивному определению абстракции схемы.

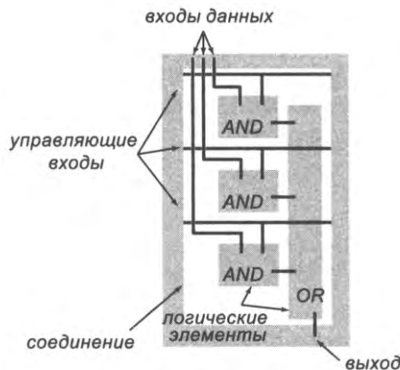
Схема *представляет собой несколько логических элементов или схем (более простых), соединенных между собой проводниками, часть из которых определяется как входные или выходные.*

Несмотря на заманчивую простоту, для ясности мы слегка уточним это определение, когда начнем строить все более сложные схемы на следующих уровнях абстракции.

Построение схем из логических элементов

Из опыта написания программ на Java вам хорошо известна мощь концепции построения больших программ из меньших: вы определяете интерфейсы и соблюдаете соглашения вызова. Аналогичный принцип распространяется и на оборудование. С этого момента мы будем разрабатывать схемы, соединяя логические элементы с соблюдением следующих правил.

- Входы располагаются слева и сверху, выходы располагаются справа и снизу (как и прежде).
- Некоторые входы, называемые *управляющими линиями*, окрашены в синий цвет и проходят по всей длине схемы.
- Элементы помечаются их функцией и соединяются так, как предписывает их интерфейс.
- Все переключатели скрываются под прямоугольниками, представляющими элементы.



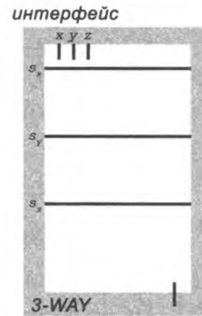
Строение схемы на уровне логических элементов

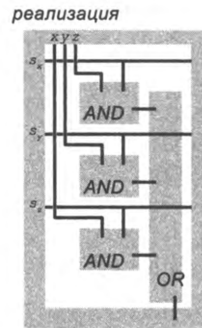
Например, схема вверху состоит из трех элементов AND и одного элемента OR; она имеет три входа, три управляющие линии и один выход. Полезно различать входы, управляющие перемещением данных (управляющие линии), и входы, по которым

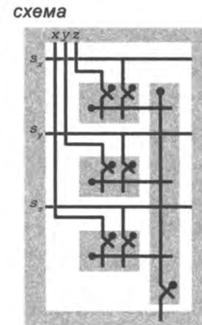
передаются данные. Для удобства управляющие линии проходят через всю схему, чтобы мы могли соединяться с ними с любой из сторон. Иногда это нарушает обычное соглашение о том, что входы должны располагаться слева, но не противоречит интуиции, потому что в отличие от входов данных мы не представляем себе «перетекание» данных по управляющим линиям на выходы.

Пример: мультиплексор

K-канальный мультиплексор (или *мультиплексор K-в-1*) представляет собой комбинационную схему с k парами входных проводников и одним выходным проводником, который передает одно из входных значений на выход по следующим правилам: у каждой пары имеется *вход данных* (*сигнальный* или *информационный*, вход) и *управляющая линия*. В любой момент времени в состоянии 1 находится не более чем одна управляющая линия (такие двоичные комбинации называют *унитарным кодом*), что служит признаком выбора соответствующего входа данных для передачи его значения на выход. Таким образом, схема служит своего рода *логическим переключателем*: ее выходное значение совпадает с выбранным входным значением. Мы используем уточнение «логический», потому что проводники не обязаны соединяться физически: достаточно совпадения передаваемых значений.







Мультиплексор «3-в-1»

Изображенный вверху справа интерфейс соответствует 3-канальному («3-в-1») мультиплексору. Входные значения помечены x, y и z, а управляющие линии — s_x, s_x и s_z. Схема должна установить на выходе значение x, если s_x находится в состоянии 1, и т. д. Интерфейс также задает размер и форму схемы, позиции входов и выхода. Как обычно, эти геометрические ограничения несущественны для понимания того, что делает схема, но их соблюдение — не большая цена за простоту понимания схемы и ее соединения с другими схемами.

Эту задачу решает реализация на среднем изображении. По ней ясно видно, что схема вычисляет логическую функцию $s_x x + s_y y + s_z z$. (В формуле не отражено условие, что не более чем одна управляющая линия может находиться в состоянии 1 — см. упражнение 7.3.4.)

В нижней части изображена непосредственная реализация логических элементов. Мы будем время от времени приводить полную низкоуровневую реализацию, чтобы указать на различные свойства всей схемы. Например, из нее очевидно, что между входами и выходами схемы нигде нет явного физического соединения.

Как вы увидите, мультиплексор «3-в-1» — не просто учебный пример. Такие схемы используются для переключения логических соединений между блоками процессора. На интуитивном уровне можно представлять себе мультиплексор как механизм, аналогичный тому, который используется для подключения разных входных устройств к вашему телевизору или экрану монитора (скорее всего, у вас есть такой мультиплексор).

Проектирование на уровне логических элементов в сочетании с простыми соглашениями, описанными ранее, существенно упрощает процесс построения схем. Большинство людей начинает изучение проектирования схем именно на уровне логических элементов. Как и в случае с программными продуктами, вы вскоре привыкнете работать на более высоких уровнях абстракции.

Наш подход слегка усложняет соблюдение геометрических соглашений в целом, но эта дополнительная работа немедленно окупается, потому что в его основе всегда лежат полные схемы.

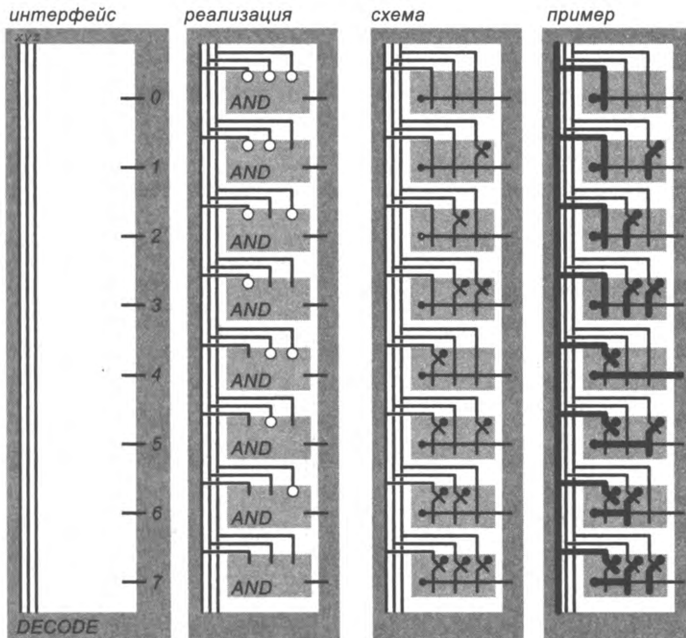
Дешифраторы, адресные демультиплексоры и мультиплексоры

Чтобы рассмотреть дополнительные примеры проектирования на уровне логических элементов, мы познакомимся с несколькими полезными комбинационными схемами, построенными на базе элементов AND (и возможно, одного элемента OR). Такие схемы станут хорошим первым шагом в изучении реальных схем на уровне логических элементов. На первый взгляд схемы выглядят устрашающе, но логика их поведения на самом деле достаточно проста. У всех таких схем n входов, проходящих сверху вниз, и 2^n n-входовых элемента AND, по одному для каждой из возможных комбинаций инвертирования n входов. Схемы отличаются по выходным сигналам и типу использования дополнительного входа.

Дешифратор

Дешифратор представляет собой комбинационную схему с n входами и 2^n выходами; входной набор интерпретируется как двоичное число, которое определяет, какой выходной проводник следует активизировать. А конкретнее, если интерпретировать n входных значений как n-разрядное двоичное число i и пронумеровать выходные проводники от 0 до $2^n - 1$, то i-й выход дешифратора будет находиться в состоянии 1, а все остальные выходы — в состоянии 0; таким образом, дешифратор преобразует позиционный двоичный код в унитарный. В левой части рисунка на с. 985 изображены интерфейс и внутренняя реализация 3-разрядного дешифратора, а также пример состояния его входов и выходов.

Как показано в правой части рисунка, дешифратор строится простым подключением ко всем входам каждого из 2^n обобщенных элементов AND с n входами. Каждый выход находится в состоянии 1 только для одного набора входных значений — того, в котором 0 соответствуют инвертированным вводам, а 1 —



3-разрядный дешифратор

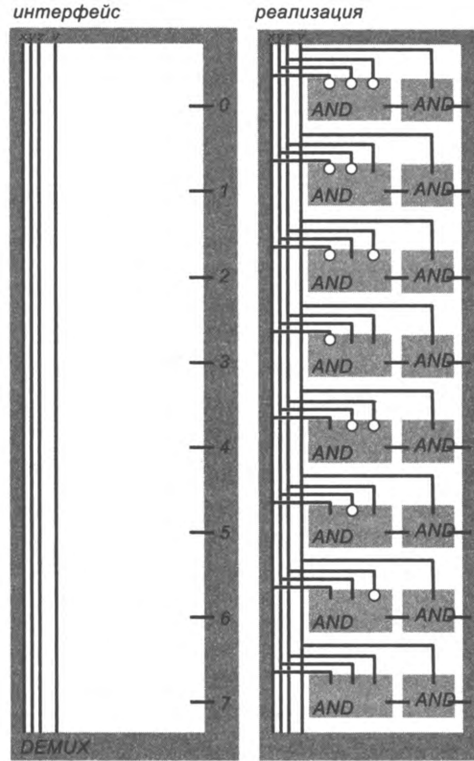
неинвертированным. При раскрытии реализации элементов мы получаем схему, изображенную в третьей части иллюстрации. В примере анализа переключений схема интерпретирует входное значение 100 как двоичное число 4 и переводит выход 4 в состояние 1 (а все остальные — в состояние 0).

Как будет показано в разделе 7.5, дешифраторы играют важнейшую роль в процессорах: они позволяют преобразовывать двоичные коды, содержащиеся в машинных инструкциях, в единичные значения на проводниках, которые активизируют схемы, определяемые двоичными кодами.

Мы привели пример с анализом на уровне переключателей, чтобы вы видели, как каждый из возможных логических элементов AND реагирует на заданный входной набор. Мы не будем повторно показывать подробно внутреннее устройство таких схем, так как их поведение достаточно очевидно следует из реализации уровня логических элементов.

Адресный демультиплексор

Адресный демультиплексор (см. с. 986) добавляет к дешифратору еще один вход (который мы называем входным значением) и действует как логический переключатель 1-в-2^n, передающий входное значение на выбранный выход. Другими словами, все выходы равны 0, кроме одного, заданного двоичным значением адресных входов; он равен 0, если входное значение равно 0, и равен 1, если входное



3-разрядный адресный демультиплексор

значение равно 1. Как вы видите, это поведение достигается простым добавлением логического элемента AND к каждому выходу дешифратора, чтобы ни один выход демультиплексора не был равен 1, если входное значение не равно 1. Для экономии мы используем элемент AND, у которого один из входов располагается слева (см. упражнение 7.3.2). Полное изображение схемы опущено, потому что оно очень похоже на схему дешифратора на предыдущей странице.

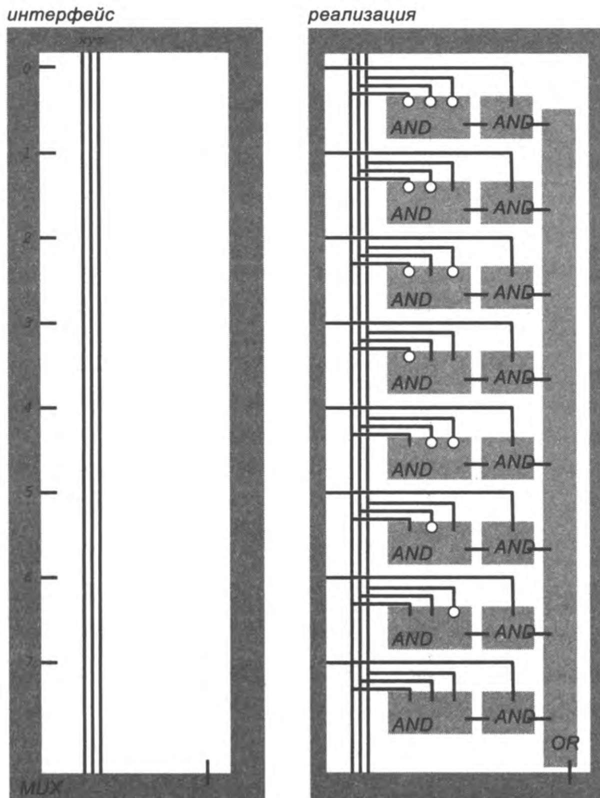
И снова следует заметить, что демультиплексор является *логическим* переключателем — между входом и выбранным выходом не существует физического соединения. Мы просто обеспечили соответствующее значение на выбранном выходе.

Как будет показано в разделе 7.5, такие демультиплексоры также играют очень важную роль в процессорах, потому что они позволяют передавать *значения* в другие части схемы в соответствии с двоичными кодами в машинных инструкциях.

Адресный мультиплексор

Адресный мультиплексор представляет собой комбинационную схему с $n+2^n$ входами и одним выходом, которая передает на выход одно из входных значений; таким образом, он действует как логический переключатель 2^n-в-1.

Как видно из реализации внизу, эта функциональность реализуется использованием дополнительного логического элемента AND для каждого выхода (как и в случае с демультиплексором) и подачей на них входов с последующим использованием (вертикального) многовходового логического элемента OR для формирования выхода. В этом случае только один из входов элемента OR может быть равен 1 (единственная возможность — выбранная линия, которая равна 1 в том и только в том случае, если вход на этой линии также равен 1), так что выход будет иметь нужное значение. Например, если на входах *xyz* подаются значения 110, то на выход будет подано значение входной линии 6. Как обычно, схема является ***логическим*** переключателем — не существует физического соединения между выбранным входом и выходом.



3-разрядный адресный мультиплексор

Как вы, вероятно, уже догадались, такие мультиплексоры тоже играют важную роль в процессорах, потому что они позволяют использовать двоичные значения в позиционном коде в машинных инструкциях для выбора значений данных на выходе.

Вертикальный многовходовый логический элемент OR с одним активным входом

Вертикальные многовходовые логические элементы OR в наших мультиплексорах очень легко анализировать, потому что их входы всегда удовлетворяют одному особому условию. А именно ровно один вход в многовходовом элементе OR находится в состоянии 1 (активен). В принципе, можно построить специальную схему, использующую эту свойство; мы используем обычный многовходовый элемент OR. Это условие выполняется, потому что один и только один из обобщенных логических элементов AND может выдать 1 на выходе — тот, который соответствует выбранному входу данных. С этого момента каждый раз, когда мы упоминаем «элемент OR», имеется в виду именно такой логический элемент.

Мы рассмотрели четыре полезные комбинационные схемы, построенные на основе вертикального столбца логических элементов AND. Изучите эти элементы и убедитесь в том, что вы полностью понимаете их. Они демонстрируют правила, которые будут применяться в этой главе для построения более сложных схем; в частности, эти правила чрезвычайно важны для процессора, который будет спроектирован в разделе 7.5. Но что еще важнее, полное понимание архитектуры этих схем поможет вам в полной мере осознать возможности их обобщения — до такой степени, какую вы сейчас и представить не сможете.

Схемы для вычисления суммы произведений

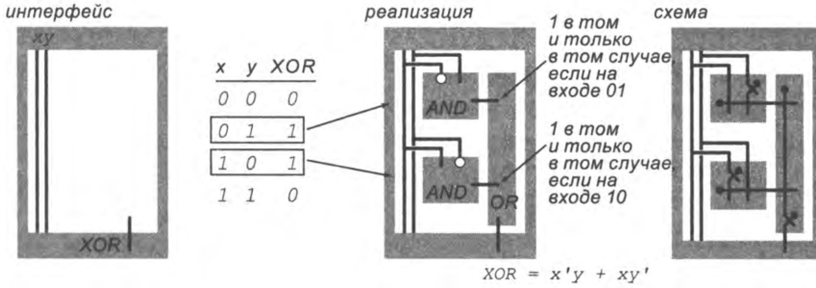
Базовый метод, который мы только что использовали, расширяется и позволяет объединить логические элементы AND в столбец для построения схемы, на выходе которой формируется любая заданная логическая функция ее входных значений. Такая схема может быть построена непосредственно по таблице истинности, задающей функцию. Это делается так.

- Отберите строки таблицы истинности, для которых функция равна 1.
- Подайте входы каждой такой строки на обобщенный вентиль AND, который выдает 1 в том и только в том случае, если на входы поданы значения этой строки.
- Подайте выходы всех этих вентилей на (вертикальный) многовходовый логический элемент OR.
- На выходе элемента OR появится значение функции.

Эта конструкция полностью эквивалентна нашей конструкции вычисления логической суммы произведений для логической функции в дизъюнктивной нормальной форме.

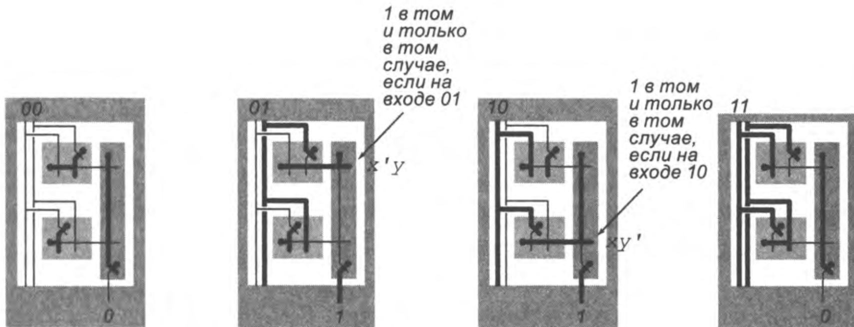
XOR

Для начала рассмотрим конструкцию, которая реализует схему из трех логических элементов для функции XOR двух переменных.



Построение схемы XOR по таблице истинности

Мы построили обобщенные логические элементы AND для x'y и xy', а их выходы подключили к (вертикальному) элементу OR для вычисления x'y + xy'. Если вы хотите лишний раз убедиться, что схема работает так, как задумано, выполните анализ всех возможных вариантов ввода на уровне переключателей на рисунке ниже.



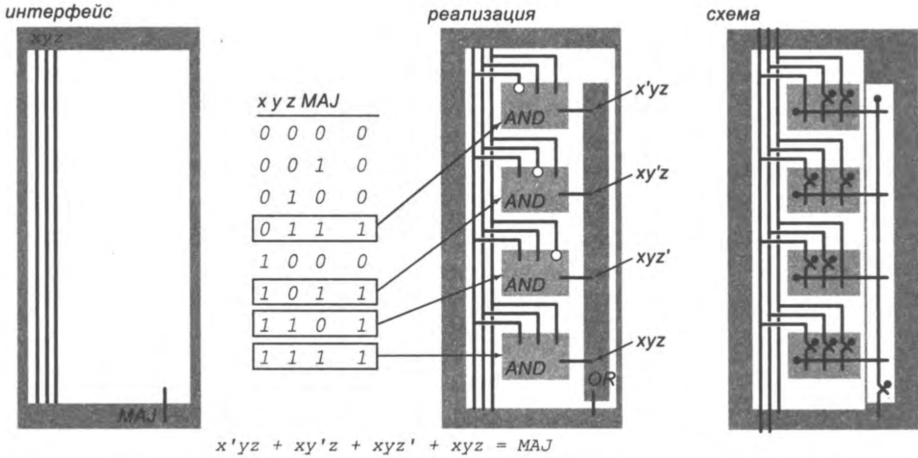
Конечно, мы воспользуемся этой схемой позднее, для реализации операции поразрядного XOR в нашем процессоре.

MAJ и ODD

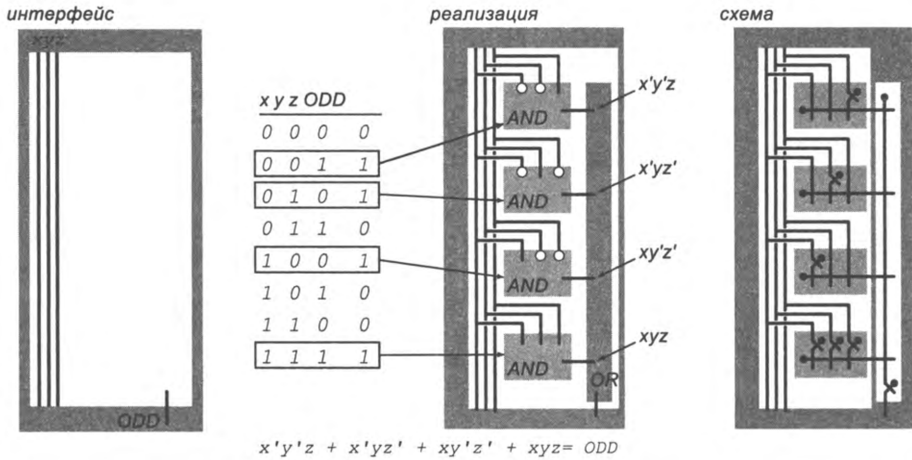
Аналогичным образом, чтобы построить схему для вычисления мажоритарной функции, мы можем взять логические элементы для вычисления x'yz, xy'z, xyz' и xyz, а затем объединить их выходы по OR.

И конечно, мы можем построить схему для вычисления функции нечетности ODD для трех входных переменных; для этого нужно взять логические элементы для x'y'z, x'yz', xy'z' и xyz, а затем подать их выходные значения на вертикальный элемент OR:

Очевидно, этот же метод может быть применен для любой логической функции. Если у вас есть полностью определенная логическая функция, вы можете сформировать ее таблицу истинности; при наличии таблицы истинности вы сможете



Построение схемы MAJ по таблице истинности



Построение схемы ODD по таблице истинности

построить схему для вычисления определяемой ею функции. Идея о том, что вы можете так легко построить схему для вычисления любой логической функции, чрезвычайно глубока.

Практическое ограничение

Однако существует одно ограничение — возможно, вы его уже заметили: схемы, напрямую реализующие СДНФ функции, обычно мало полезны на практике при большом количестве входов, потому что необходимое количество логических

элементов может расти экспоненциально. Например, если вы захотите воспользоваться этим методом для вычисления мажоритарной функции с 64 входами, вам потребуется более 2^{63} элементов — нереально, особенно для такой простой функции. Аналогичным образом дело обстоит со схемой мультиплексора, рассмотренной в предыдущем разделе: n-разрядный адресный мультиплексор («2^n-в-1») имеет $2^n + n$ входов, поэтому для использования этой конструкции придется рассматривать 2^{2^n+n} строк. Конечно, такая схема будет ужасающе непрактичной — особенно если учесть, что у нас уже есть схема, использующая всего $2^n + 1$ элементов. Например, 4-разрядный мультиплексор состоит из 17 логических элементов, но полная таблица истинности этой функции содержит более 2 миллиардов строк.

Тем не менее схемы с суммой произведений приносят огромную практическую пользу. Представьте, что у вас есть ящик логических элементов, помеченных значениями строк таблицы истинности, и вы строите схему для вычисления произвольной логической функции, выбирая элементы из ящика и соединяя их в соответствии со строками таблицы истинности. Или представьте, что у вас есть большой дешифратор и вы просто подключаете элементы NOT в местах, соответствующих 0 таблицы истинности. Или представьте, что ваша программа автоматически преобразует таблицу истинности в схему, вычисляющую сумму произведений, потому что это всего лишь разные воплощения одной абстракции. Все эти представления не так уж далеки от реальных способов построения компьютеров в разные моменты их истории.

Как правило, эффективность проектирования можно улучшить: начните с СДНФ как канонического логического выражения и упростите его, применяя базовые правила булевой алгебры. Обычно можно найти схемы, использующие меньшее число логических элементов, чем схемы, реализующие СДНФ. Например, тождество

$$MAJ(x, y, z) = xy + xz + yz$$

(которое можно проверить по таблице истинности) непосредственно дает схему, которая использует всего три 2-входовых элемента AND — по сравнению с четырьмя 3-входовыми элементами AND, используемыми в реализации непосредственно по таблице истинности (см. упражнение 7.2.4). В области минимизации схем велись интенсивные исследования на заре компьютерной эпохи (когда каждый логический элемент был отдельным физическим объектом), поэтому для многих часто используемых логических функций известны схемы, использующие минимально возможное количество элементов. Чтобы не вносить путаницу, мы вынесли такие упрощения в раздел упражнений[1].

Схемы с логической суммой произведений предоставляют еще один уровень абстракции. Мы знаем, что, если использование до 2^n логических элементов допустимо, мы сможем построить схему для вычисления любой логической функции с n переменными. Этот факт — большой шаг вперед как от самого начала пути,

[1] Существует ряд методов минимизации логических функций, в том числе весьма сложных. Их подробное рассмотрение выходит за рамки этой книги. — *Примеч. науч. ред.*

когда мы представляли схемы в виде «черных ящиков», так и от первого уровня абстракции, на котором схемы строились из переключателей и проводников. Во-первых, у нас появляется систематичный метод практического построения схем для логических функций с малым числом аргументов. Во-вторых, мы знаем, что для любой логической функции можно построить схему. Следовательно, теперь можно просто вычислить необходимую ширину и высоту и затем изобразить для произвольной логической функции интерфейс наподобие того, который был показан для функций MAJ и ODD (при большом количестве входов приходится искать возможности построения схем с меньшими затратами ресурсов).

Все схемы, рассмотренные в этом разделе, функциональны, но они выполняют относительно простые вычисления. Пора рассмотреть схему для выполнения вычислений, которые обычно ассоциируются с компьютером: схему для суммирования двух n-разрядных чисел.

Сумматор

Тщательно проанализируем процесс суммирования двух двоичных чисел с применением метода, который вы изучали в школе. Справа приведен пример с подробным описанием вычисления 5 + 6 = 11, или в виде 4-разрядных двоичных чисел, `0101 + 0110 = 1011`. В таблице под этим примером показано, как всем битам, участвующим в суммировании, присваиваются символические имена. В общем случае 4-разрядный сумматор имеет 8 входов $x_3\, x_2\, x_1\, x_0$ и $y_3\, y_2\, y_1\, y_3$, четыре выхода $z_3\, z_2\, z_1\, z_0$ и пять «внутренних» значений переноса $c_4\, c_3\, c_2\, c_1\, c_0$. Бит c_0 удобно рассматривать как дополнительное входное значение (которому присваивается 0), а c_4 — как дополнительное выходное значение (которое можно игнорировать или использовать для проверки переполнения).

```
0 1 0 0 0
  0 1 0 1
  0 1 1 0
  -------
  1 0 1 1
```

$c_4\ c_3\ c_2\ c_1\ c_0$ ← биты переноса
$x_3\ x_2\ x_1\ x_0$ } входные биты
$y_3\ y_2\ y_1\ y_0$
$z_3\ z_2\ z_1\ z_0$ ← выходные биты

4-разрядное сложение

Реализация в виде суммы произведений (гипотетическая)

N-разрядный сумматор может быть реализован на базе $n + 1$ комбинационной схемы (по одной для каждого выходного бита), каждая из которых имеет $2n + 1$ входов. Теоретически можно рассмотреть реализацию таких схем на базе суммы произведений, но так как таблицы истинности будут содержать 2^{2n+1} строки, этот подход придется отклонить. Сам факт существования такой схемы важен, но нам придется поискать вариант с разумным количеством логических элементов.

Каскадный сумматор

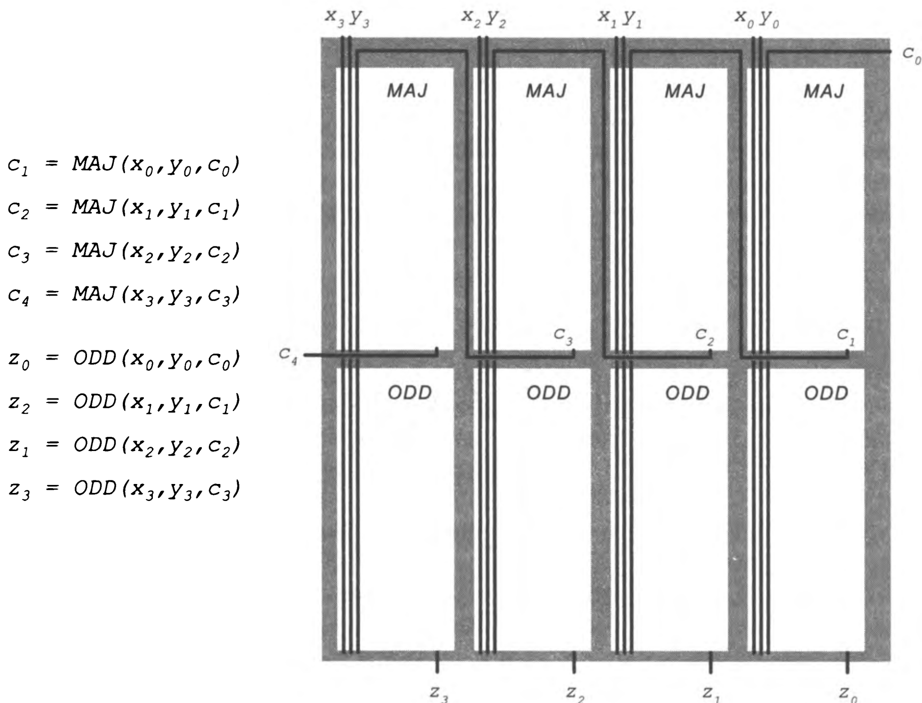
Вместо этого мы разработаем схему, которая действует так же, как человек, складывающий два числа. В приведенном выше примере крайнее правое значение 1 прибавляется к крайнему правому значению 0 (с неявным переносом 0); получаем

крайний правый бит результата 1 с переносом 0. Затем перенос 0 и входные биты 0 и 1 суммируются во второй позиции справа, что дает выходной бит 1 и еще один перенос 0. Этот перенос суммируется с двумя входными битами 1 в третьей позиции справа, давая выходной бит 1 и перенос 1. Наконец, перенос прибавляется к двум входным битам 0 в крайней левой позиции, мы получаем выходной бит 1 и перенос 0.

Для каждой позиции суммирование сводится к вычислению двух логических функций трех аргументов: выходного бита и бита переноса для следующей позиции. Как выглядят эти две логические функции?

- Бит переноса равен 0, если количество 1 среди трех входных битов равно 0 или 1; он равен 1, если количество 1 во входных битах равно 2 или 3 — это не что иное, как *мажоритарная функция*, применяемая к этим трем битам.
- Выходной бит равен 1, если количество 1 во входных битах равно 1 или 3; количество равно 0, если количество 1 во входных битах равно 0 или 2 — это не что иное, как *функция нечетности*, применяемая к этим трем битам.

Соответственно мы можем записать логические формулы (внизу слева), которые определяют каждое из необходимых выходных значений как функцию входных значений и переносов. Таблица фактически является спецификацией для построения 4-разрядного сумматора (справа от формул). Мы используем четыре схемы

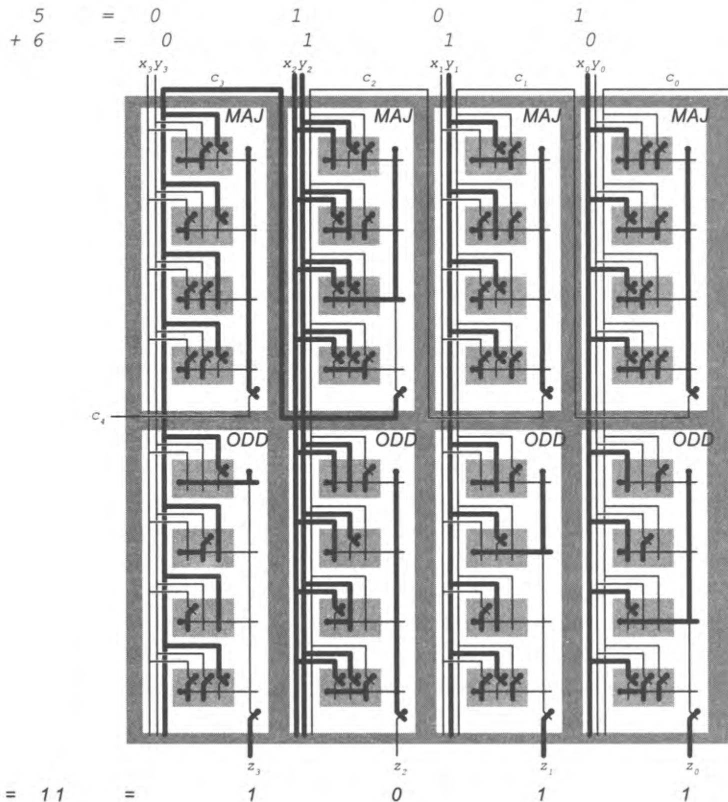


4-разрядный каскадный сумматор

MAJ и четыре схемы элемента ODD, соединенных вместе так, как следует из этих формул. Такая конструкция называется *каскадным сумматором*. Начиная справа, мы передаем входные значения c_0, x_0 и y_0 схемам MAJ и ODD, расположенным справа; схемы вычисляют перенос c_1 и бит результата z_0. Затем c_1, x_1 и y_1 передаются схемам MAJ и ODD для позиции 1, вычисляется перенос c_2 и результат z_1 и т. д. Перенос «каскадом» проходит по схемам справа налево.

В современных компьютерах используются более сложные сумматоры, в которых каскадное распространение переносов устраняется, поэтому они быстрее работают с данными большой разрядности, но наша схема хорошо демонстрирует идею выполнения вычислений. Обратите внимание: мы поднялись на еще один уровень абстракции, так как схема строится из схем MAJ и ODD, которые, в свою очередь, строятся из логических элементов.

Анализ на уровне переключателей показывает значения на всех проводниках схемы в том случае, когда на вход 4-разрядного сумматора подаются данные для вычисления 5 + 6 = 11. Этот пример заслуживает тщательного изучения.



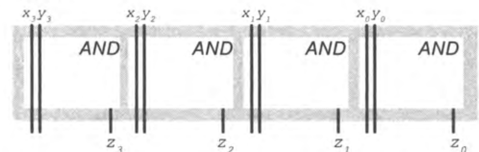
Анализ 4-разрядного сумматора, выполняющего сложение 5 + 6 = 11, на уровне переключателей

Эта конструкция напрямую масштабируется до n разрядов, давая схему с $2n + 1$ входами и $n + 1$ выходами; такая схема может суммировать два n-разрядных числа с использованием $8n$ обобщенных логических элементов AND и $2n$ многовходовых элементов OR. Иначе говоря, наш 4-разрядный сумматор содержит 40 логических элементов, а 32-разрядный — всего 320 логических элементов. Конечно, эти показатели значительно выигрывают по сравнению с тривиальным решением, построенным по таблице истинности с 2^{2n+1} строками (для чего потребуется более 9×10^{18} элементов).

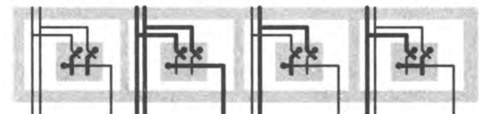
Арифметико-логическое устройство (АЛУ)

Чтобы построить АЛУ для такой машины, как TOY, необходимо устройство с возможностью выполнения следующих арифметических и логических операций: сложения, вычитания, конъюнкции (AND), исключающего OR (XOR), левого сдвига и правого сдвига. Для простоты и для того, чтобы подчеркнуть важнейшие аспекты конструкции, мы будем работать с TOY-8 — младшим братом семейства TOY, упоминавшимся в конце главы 6 и подробно описанным в разделе 7.5. Для АЛУ это означает, что реализовать нужно только сложение, AND и XOR. Все необходимые базовые схемы уже были рассмотрены выше; теперь мы увидим, как объединить их в одно устройство, которое имеет два входа для входных n-разрядных значений и три управляющих линии «выбора операции» и выдает на выходе n-разрядный результат (входной и выходной перенос игнорируются). Управляющие линии служат для выбора нужного выхода АЛУ.

реализация



анализ на уровне переключателей (0101 & 0110 = 0100)



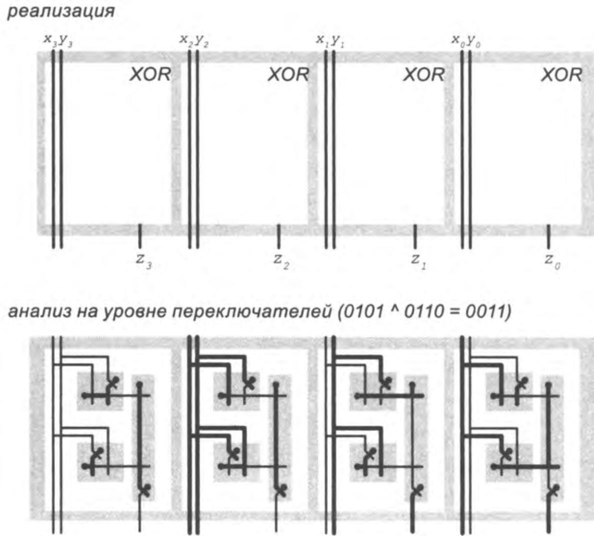
4-разрядная схема AND

Битовые операции

Схемы для поразрядных операций AND и XOR очень просты. Для поразрядной операции AND (сверху справа) мы просто используем по одному элементу AND для каждого разряда. Для поразрядной операции XOR (с. 996) используются схемы с суммой произведений XOR, которая рассматривалась ранее среди примеров, по одной для каждого разряда.

Входы

Все наши схемы — сумматор, поразрядная операция AND, поразрядная операция XOR — получают одинаковые входные данные. Как видно из интерфейса всех трех операций, они спроектированы так, что могут накладываться друг на друга: входные проводники проходят через схемы насквозь (в левой части каждой 1-разрядной



4-разрядная схема XOR

ячейки). Все схемы имеют одинаковую общую компоновку, чтобы нам было удобно подать один набор входных сигналов на каждую схему.

Выходы

Все три схемы также вычисляют *n* выходных значений (и перенос для сложения), но в любой конкретный момент времени нужно выбрать, какой набор значений должен находиться на выходах АЛУ. Для этого мы просто направляем для каждого разряда результата все три выхода на мультиплексор «3-в-1» (первая схема уровня логических элементов на с. 982) и проводим управляющие линии по горизонтали через все мультиплексоры. Тогда вычисленное значение для выбранной операции появляется на выходах этих мультиплексоров.

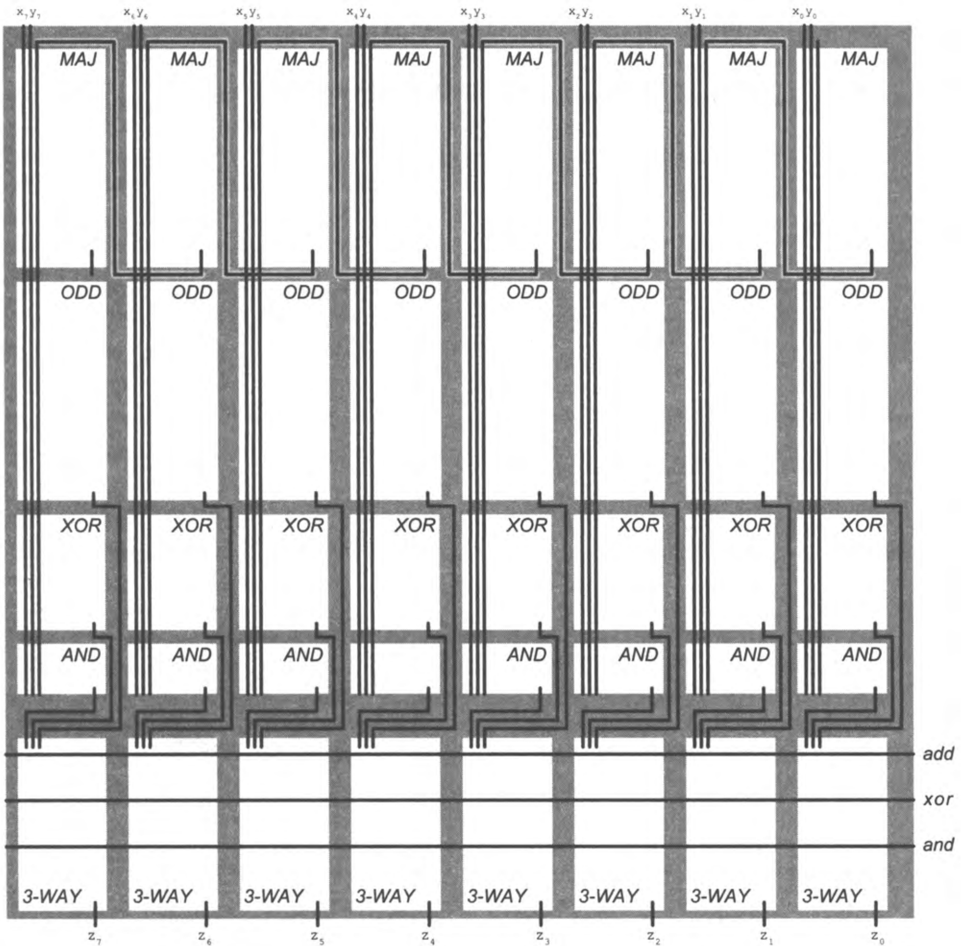
Объединяя воедино все сказанное, мы получаем полную схему 8-разрядного АЛУ, необходимого для TOY-8 (см. иллюстрацию на следующей странице). Она реализует сложение, поразрядные операции AND и XOR, получает 16 разрядов входных данных и выдает 8-разрядный результат в соответствии с состоянием 3 управляющих линий. Когда ровно одна из управляющих линий равна 1, на выходах появляются вычисленные выходные значения выбранной схемы. Интересно, что какая бы вычислительная схема ни была выбрана управляющей линией, все остальные результаты тоже вычисляются, но игнорируются АЛУ. Это типичная ситуация: ваш компьютер вычисляет множество результатов, которые затем полностью игнорируются!

Мы не станем раскрывать реализации на уровне логических элементов в этой схеме, потому что вы уже видели все подробности, а логика самого АЛУ лучше всего понятна на таком уровне абстракции. Но если бы мы привели эту реализацию, вы

смогли бы увидеть каждый переключатель и понять его роль в вычислениях. Впрочем, все переключатели будут показаны в конце главы для завершенного процессора.

		4	**8**	**64**
сложение	10n	40	80	640
XOR	3n	12	24	192
AND	n	4	8	64
мультиплексор «3-в-1»	4n	16	32	256
Итого	18n	72	144	1152

Количество логических элементов в



8-разрядное арифметико-логическое устройство

Эта схема принципиально не отличается от АЛУ вашего компьютера; таким образом, тщательно изучив ее, вы поймете, как ваш компьютер выполняет вычисления. Возможно, АЛУ вашего компьютера содержит больше компонентов и работает со словами большей разрядности, но вы наверняка понимаете, как масштабировать свое решение по обоим измерениям. В таблице на с. 997 указано количество логических элементов, необходимых для n-разрядного АЛУ с такой архитектурой.

Это устройство убедительно демонстрирует мощь абстракции и является достойным завершением нашего изучения комбинационных схем. Этот важный функциональный блок играет центральную роль в процессоре любого компьютера, и он занимает видное место в схеме TOY-8, которая будет рассмотрена в разделе 7.5.

Модули и шины

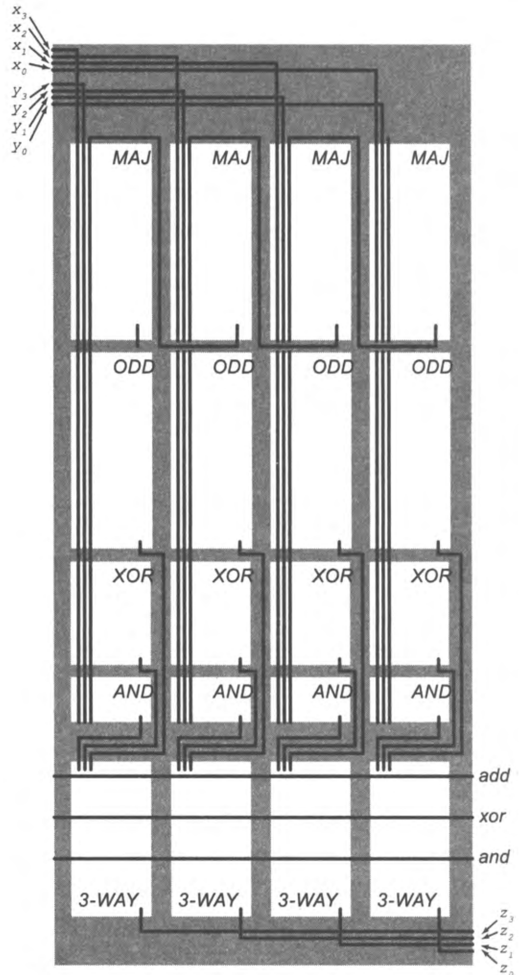
Ранее мы показали, что комбинационные схемы предоставляют нам возможность вычисления логических функций. По этой причине они играют важнейшую роль в вычислительных схемах «микро»-уровня. Далее мы рассмотрим их роль на «макро»-уровне — объединение основных компонентов схем. Но для этого необходимо представить некоторые новые термины.

Модули

В ходе построения компьютера необходимы схемы для реализации различных абстракций, которые и составляют компьютер. В частности, необходимо построить память, регистры и АЛУ. Такие схемы (реализующие основные компоненты компьютера) называются *модулями*, или *блоками*. Как ни странно, количество таких блоков в типичном компьютере невелико.

Шинные соединения

Для передачи данных между модулями используются *шины* (buses); в сущности, это просто группы проводников. Мы также иногда будем

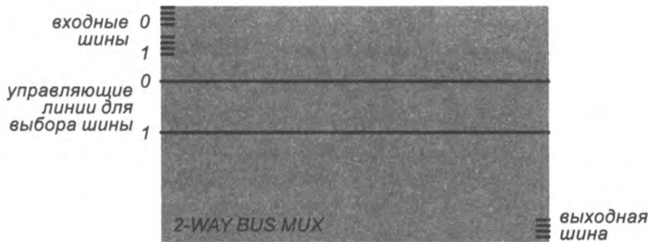


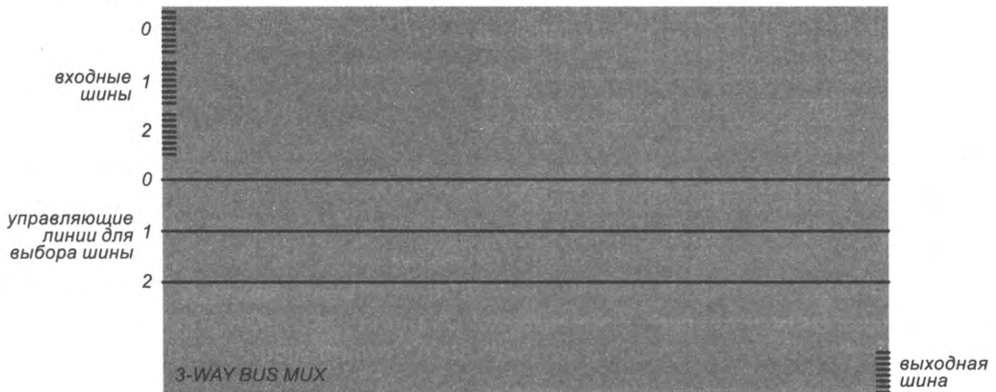
Шинные соединения в 4-битовом АЛУ

называть шины *магистралями*, чтобы четко обозначить их функцию: в процессе вычислений данные передаются по шине из памяти в регистр, из регистра в АЛУ, из регистра в память и т. д. Вместо разводки отдельных входов и выходов они группируются в шины. Например, справа изображены входные и выходные шинные соединения для нашей схемы АЛУ. Как обычно, входы размещены сверху и слева, а выходы — снизу и справа. Шинные соединения позволяют легко соединить АЛУ с другими блоками процессора.

Шинный мультиплексор

Для разминки мы рассмотрим очень важный пример: семейство комбинационных схем, получающих на входе m шин с разрядностью n (разрядность шины равна количеству проводников в шине) и реализующих логическое переключение для всей шины. Все выглядит так, словно на выход переключается одна из шин. Для выбора шины используется m управляющих линий; предполагается, что не более одного из них находится в состоянии 1, а выбранная шина переключается на выход. Например, верхнее изображение на иллюстрации на этой странице — интерфейс шинного мультиплексора «2-в-1» для 4-разрядных шин, а нижнее — «3-в-1» для 8-разрядных шин.

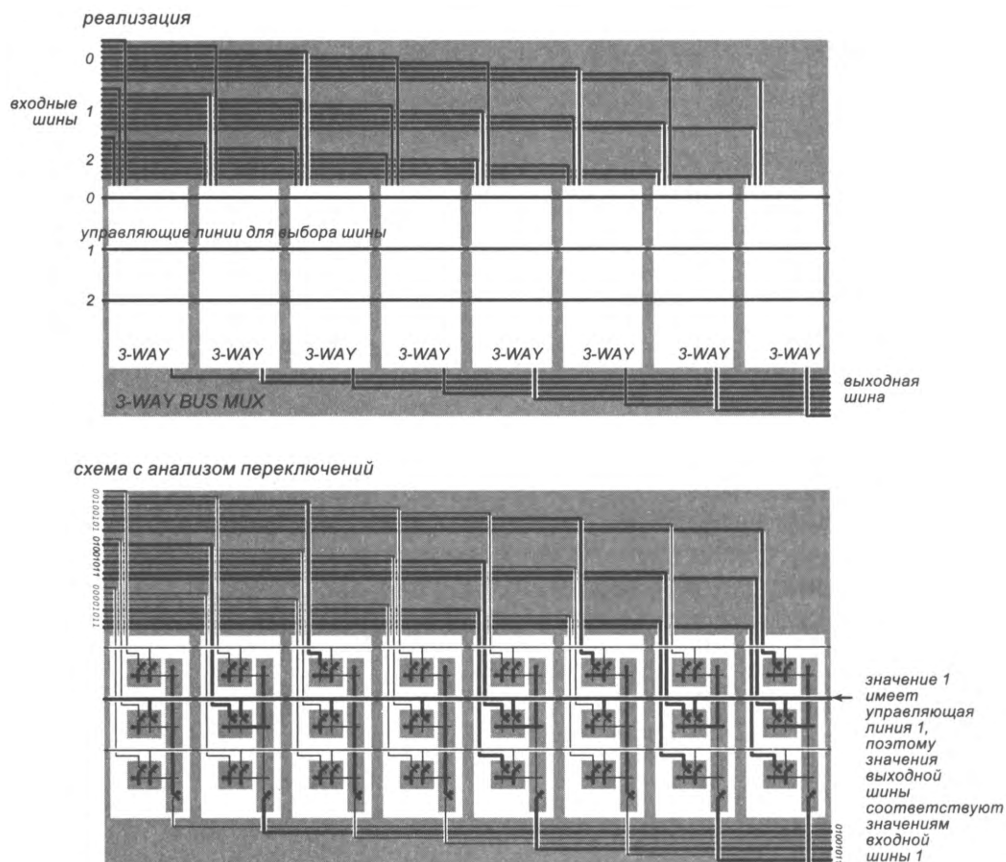




Интерфейсы шинных мультиплексоров

Эта реализация представляет собой очень простой пример применения мультиплексоров «3-в-1», по одному для каждого проводника (разряда) шины, как и в нижней части АЛУ. Выходы для каждого разряда шины подаются на мультиплексор (первая схема на уровне логических элементов, которая была рассмотрена на с. 982), а управляющие линии проходят через мультиплексоры по горизонтали. Вычисленное значение для выбранной операции появляется на выходах мультиплексоров. Реализация шинного мультиплексора «3-в-1» для 8-разрядных шин изображена вверху этой страницы, а ниже приведен пример анализа переключений. ***Убедитесь в том, что вы понимаете, как работает эта схема***: ее работа критична для работы схемы процессора, которая будет рассматриваться в разделе 7.5.

И снова, шинные мультиплексоры представляют собой *логические* переключатели — ни один входной проводник не соединяется ни с одним выходным. Но в любой момент времени, кроме периода переключения, схема работает так, словно такое соединение существует.



Шинный мультиплексор «3-в-1» для 8-разрядных шин (реализация, схема и пример анализа)

Уровни абстракции

Мы строили схемы из проводников и переключателей на трех уровнях абстракции.

- *Логический элемент* представляет собой схему, построенную из переключателей (AND, NOR).
- *Схема уровня логических элементов*, или *узел*, представляет собой схему, построенную из таких простейших элементов (MAJ, ODD, дешифратор).
- *Модуль*, или *блок*, представляет собой схему, построенную из логических элементов или более сложных узлов, которая имеет входные и выходные шины, а также управляющие линии для соединения с другими модулями (например, АЛУ или шинный мультиплексор).

После полного рассмотрения внутреннего строения схем мы теперь можем работать со схемами на уровне абстракции функциональных блоков, используя только интерфейс, определяемый шинами, управляющими линиями и размерами, как показано на следующей странице. Каждый блок сыграет критически важную роль в работе вычислительного устройства, и мы можем ожидать, что на более высоком уровне абстракции каждый из них будет работать так, как положено.

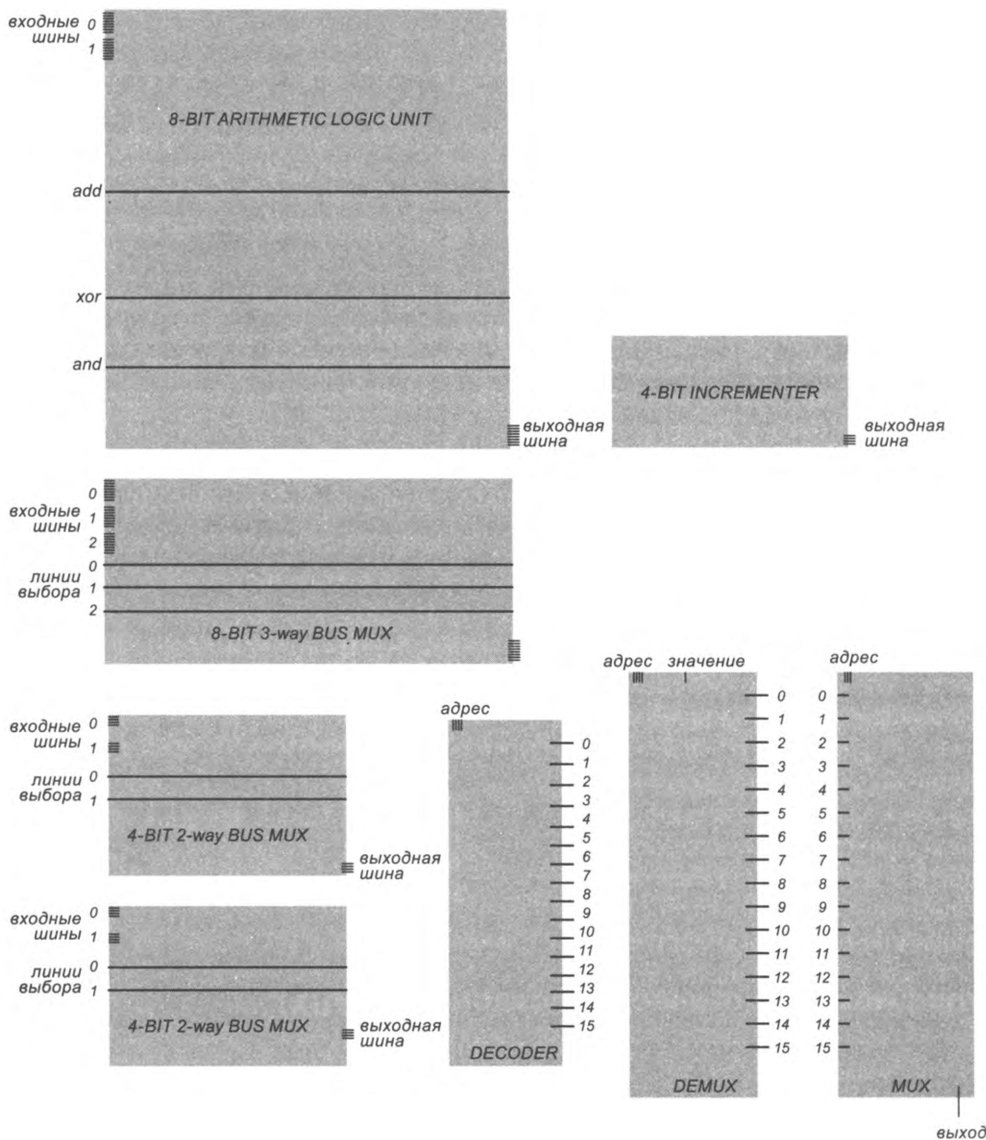
С этого момента нам придется думать не о том, как строятся эти блоки, а о том, что они делают. Не жалейте времени и еще раз убедитесь в том, что вы понимаете базовую функцию каждого из блоков, представленных на следующей странице.

Все рассмотренные нами логические элементы обладают общим свойством: их выходы зависят только от их входов. Этим же свойством обладают все узлы и модули, которые рассматривались до сих пор: все они являются комбинационными схемами. В следующем разделе будут рассмотрены последовательностные схемы, физическое состояние (и выход) которых зависит от того, как именно изменялись до этого их входные значения.

При построении схем уровни абстракции используются по той же причине, по какой они используются при построении программ: они помогают управлять сложностью и способствуют отделению клиента от реализации (то есть предоставляют те же преимущества, что и в программировании). Вы можете применять независимое тестирование и отладку блоков, подставлять улучшенные реализации и повторно использовать блоки — для этого достаточно, чтобы клиент и реализация использовали согласованный интерфейс (то есть набор входов и выходов).

Геометрия

Впрочем, схема все же ближе к реальному миру, чем программный продукт. Соответственно нам приходится учитывать в интерфейсах геометрическую информацию (топологию), указывая размер схемы и расположение входных и выходных проводников. Этот интуитивно понятный подход моделирует ситуацию с построением высокоуровневой схемы из готовых физических блоков. Вы можете отключить один блок и подключить на его место другой — при условии, что входы и выходы нового блока находятся в тех же позициях, как и у старого.



Интерфейсы функциональных блоков (для комбинационных схем)

Если полностью исключить геометрические ограничения, комбинационные схемы мало чем отличаются от логических функций. Например, логические формулы, приведенные для схем мажоритарной функции и функции четности в сумматоре, эквивалентны описаниям схем для вычисления этих функций с использованием логических элементов и более сложных узлов. Эти формальные описания важны, но они не имеют ничего общего с физической схемой. Включение геометрии в абстракцию является своего рода «золотой серединой». С одной стороны, мы можем

полностью игнорировать абстракцию и работать с переключателями и проводниками. С другой стороны, можно работать с полностью абстрактным представлением и рассматривать размещение и топологию как отдельную задачу. Наш метод рассматривает схемы и как логические функции, и как физические компоненты, которые вы видите внутри своего компьютера.

Вам было бы трудно разобраться, как работает схема АЛУ, не зная о разработанных нами уровнях абстракции, включая таблицы истинности, представление в виде суммы произведений, функции MAJ и ODD и их место в алгоритме сложения. В то же время важно мыслить понятиями физической схемы. Когда мы приводим примеры анализа на уровне переключателей, полезно проследить за тем, на каких проводниках при каждом виде вычислений передается 1, а на каких 0; это поможет вам лучше понять, как алгоритм может быть реализован в виде множества взаимосвязанных проводников и переключателей.

В нашей абстракции схемы представляются прямоугольниками с входами и выходами, расположенными на границах, но подробности конкретной абстракции не так важны, как сама идея использовать ее. Также можно попробовать разработать альтернативную абстракцию, при которой входы и выходы могут находиться в любой точке схемы, или абстракцию трехмерных схем в форме параллелепипеда, или любую другую.

Если вы хотите построить компьютер, то в какой-то момент вычислительные абстракции неизбежно столкнутся с реальным миром. Если бы в нашем реальном мире можно было легко справиться с экспоненциальным ростом количества логических элементов, то все комбинационные схемы можно было построить как прямую реализацию ее СДНФ по таблице истинности. Если бы в этом мире существовал простой способ размещения элементов и проводников, то нам не пришлось бы уделять столько внимания этому в своих разработках. Абстракции, которые мы описали, основаны на долгой истории изменений нашего понимания реального мира. В самом деле, современное проектирование и производство компьютеров базируются на схемах и чертежах, где полностью задается размер и расположение каждого проводника и каждого элемента, соединенного с проводником, причем эти чертежи не так уж сильно отличаются от тех, которые используем мы в этой книге. Такие чертежи становятся входными данными для процесса производства микросхем процессоров — наподобие тех, что установлены в вашем компьютере или мобильном устройстве.

Вот уже более полувека при появлении новой технологии (даже новой реализации простого физического переключателя) нам удавалось довольно быстро строить новые схемы на базе этой технологии и немедленно переходить к совершенствованию программных систем и приложений, уровень за уровнем. Благодаря этой способности компьютерные технологии получили повсеместное распространение и, пожалуй, стали наиболее убедительным свидетельством мощи абстракции.

Другое доказательство мощи абстракции — эта глава. В ней мы смогли разработать абстрактное представление, которое одновременно раскрывает фундаментальные свойства реальных схем и позволяет описать полную схему АЛУ всего на нескольких десятках страниц.

Вопросы и ответы

В. Реальные компьютеры действительно строятся именно так?

О. Мы игнорируем многие физические ограничения, но логическое проектирование абсолютного большинства компьютеров, которые строятся сейчас или строились в прошлом, выглядит примерно так.

В. Есть ли какие-то скрытые проблемы?

О. Наша основная цель — научить вас понимать схемы, а не проектировать их. Вспомните, что в начале изучения Java вы читали готовые программы перед тем, как заняться их самостоятельным созданием. Если начинать с «чистого листа», вряд ли ваши решения сразу будут элегантными и эффективными, подобно тому как вы вряд ли напишете эффективный и элегантный код Java с первых дней изучения языка. Все сводится к тщательному проектированию уровней абстракции и интерфейсов. После того как интерфейс будет зафиксирован, его недостатки проявятся только после завершения проектирования; следовательно, сравнение эффективности альтернативных решений потребует нескольких итераций с тщательным анализом. Как и в случае с программами, вам предоставляется возможность создать многие правила, которые затем нужно будет тщательно соблюдать.

В. Имя Шеннона мне уже попадалось — но в связи с чем?

О. Шеннон также является основоположником *теории информации* — науки о представлении информации, лежащей в основе механизмов передачи данных, сжатия данных и других областей, сформировавшихся в цифровую эпоху.

В. Является ли сумматор в моем компьютере каскадным?

О. Вероятно, нет. Современные компьютеры используют более сложную схему, при которой операция выполняется намного быстрее (за время, пропорциональное логарифму разрядности).

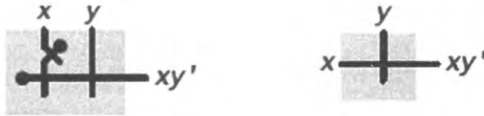
В. Почему бы не использовать «адресные» мультиплексоры для реализации шинных мультиплексоров?

О. Это возможно, но управляющие линии обычно не зависят друг от друга и не кодируются в виде двоичных чисел, поэтому схема, представленная в этом разделе, более удобна. См. упражнение 7.3.16.

Упражнения

7.3.1. На вход переключателя «вход/выкл» подается значение x, а на управляющую линию – значение y. Приведите логическое выражение для значения на выходе.

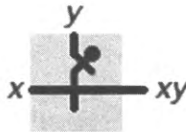
Решение. xy', функция AND NOT. Таким образом, по нашим правилам следующие две схемы эквивалентны (выход равен 1 в том и только в том случае, если значение x равно 1, а значение y равно 0).



Левый вариант схемы используется для обеспечения симметрии входов и предотвращения неоднозначности относительно того, какой проводник управляет переключателем. В реальных схемах мы избегаем передачи значений по проводникам с входа на выход, чтобы все выходы гарантированно управлялись близлежащим источником питания. Наш обобщенный логический элемент NOR можно представить в виде цепочки таких схем, соединенных с элементом NOT.

7.3.2. В духе предыдущего упражнения приведите реализацию логического элемента AND, которая использует только два переключателя.

Решение. Инвертируйте вход `y` в приведенной выше схеме. Тогда в соответствии с нашими правилами на выходе схемы выводится 1 в том и только в том случае, если и `x`, и `y` равны 1.



Наш обобщенный элемент NOR можно представить в виде цепочки таких схем, соединенных с элементом NOT.

7.3.3. Покажите, как построить NOT, NOR, OR, AND и обобщенный элемент AND с использованием только элементов AND NOT, изображенных справа в упражнении 7.3.1.

7.3.4. Постройте 3-канальную схему переключения, на выход которой подается 1 в том и только в том случае, если ровно одна линия выбора находится в состоянии 1 и соответствующий вход данных тоже равен 1. Например, в отличие от схемы на с. 983, если два входа и две управляющие линии находятся в состоянии 1, выход должен находиться в состоянии 0.

7.3.5. Опишите, к какому эффекту приведет соединение выходов адресного демультиплексора со входами адресного мультиплексора (той же разрядности) и подключение адресных входов обоих к одним и тем же адресным линиям.

7.3.6. Докажите, что `MAJ(x, y, z) = xy + xz + yz`. Изобразите схему, которая вычисляет MAJ, используя только три 2-входовых логических элемента AND и два 2-входовых элемента OR.

7.3.7. Постройте схему для 3-входовой функции ODD, использующую менее 5 логических элементов.

7.3.8. Спроектируйте две схемы, реализующие функцию XOR от двух аргументов с использованием только 2-входовых логических элементов NAND. В первой схеме вы можете соединить любой элемент NAND с 1 (подключение к источнику питания) или 0 (оставив его без такого подключения). Во второй схеме следует полагать, что все входы элементов NAND должны быть соединены со входами схемы или выходом другого элемента NAND.

7.3.9. Приведите доказательство следующего равенства через таблицу истинности:

`MAJ(x, y, z) = (x + y + z)(x' + y + z)(x + y' + z)(x + y + z')`

7.3.10. Покажите, что каждая логическая функция может быть выражена в виде СДНФ, то есть как произведение сумм, в которых задействованы каждый вход или его инверсия (как в примере из предыдущего упражнения).

7.3.11. Опишите метод построения схем на основе ДНФ (см. предыдущее упражнение) и разработайте схему ODD на базе этого метода.

7.3.12. Спроектируйте схему на основе ДНФ для 3-аргументной функции мультиплексирования из упражнения 7.1.12. Приведите 3-канальную схему для той же функции.

7.3.13. Используя свою схему мультиплексирования «3-в-1» из предыдущего упражнения для каждого выходного бита, спроектируйте схему, которая передает на выход свои n входов, переставленные в обратном порядке. *Примечание*: по тому же принципу можно построить схему для реализации любых перестановок входных данных.

7.3.14. Спроектируйте схему уровня логических элементов для вычисления функции с 4 аргументами, которая интерпретирует первые два аргумента и вторые два аргумента как 2-разрядные двоичные числа и возвращает 1 в том и только в том случае, если `wx > yz`.

7.3.15. Покажите, как изменить схему сумматора, описанную в тексте, чтобы заменить выход переноса выходом переполнения, который равен 0, если входной перенос в позицию крайнего левого бита равен выходному переносу из этой позиции, и равен 1, если они различны. Докажите, что с таким изменением сумматор правильно работает для n-разрядных чисел в дополнительном коде.

7.3.16. Приведите схему шинного мультиплексора «2^m-в-1» с двоичным кодированием входов выбора на m проводниках.

7.3.17. Допустим, переключение занимает время t (а проводники мгновенно принимают нужное значение по всей длине). Вычислите максимальное время, необходимое в каждой из следующих схем для полного отклика на изменение состояния входов.

a. NOR

b. NAND

c. 3-WAY

d. DECODE

e. MUX

f. MAJ

g. ODD

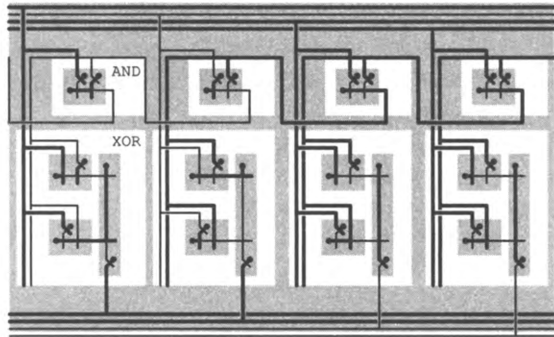
h. n-разрядный сумматор

7.3.18. Спроектируйте схему для инкрементирования (увеличения на 1) 4-разрядного числа.

Решение. Измените описание для сумматора в тексте, используйте входной перенос 1: пусть $x_3x_2x_1x_0$ — число, $c_4c_3c_2c_1 1$ — переносы, а $z_3z_2z_1z_0$ — выходы. Выходной перенос в каждой позиции равен 1 в том и только в том случае, если входной перенос и бит x равны 1 (функция AND), а выход в каждой позиции равен 0, если оба бита (входной перенос и бит x) равны 1 или оба равны 0, и равен 1, если один бит равен 0, а другой равен 1 (функция XOR). Итак, мы можем использовать схему, сходную с сумматором, но построенную на базе элементов AND для битов переноса и элементов XOR для битов суммы.

c_1 = AND(x_0, 1)
c_2 = AND(x_1, c_1)
c_3 = AND(x_2, c_2)
c_4 = AND(x_3, c_3)

z_0 = XOR(x_0, 1)
z_2 = XOR(x_1, c_1)
z_1 = XOR(x_2, c_2)
z_3 = XOR(x_3, c_3)



4-разрядный инкрементор с каскадным переносом

Упражнения повышенной сложности

7.3.19. *Построение схемы логического элемента.* Разработайте программу Java для построения схемы n-входового обобщенного элемента AND в соответствии с иллюстрациями в тексте. Программа должна получать из командной строки неотрицательное целое число, меньшее 2^n, и использовать позиции единиц в двоичном представлении числа для определения позиций инверторов.

7.3.20. *Построение схемы дешифратора.* Разработайте программу Java для построения схемы n-разрядного дешифратора. Используйте решение из предыдущего упражнения.

7.3.21. *Построение схемы для вычисления функции в ДНФ.* Разработайте программу Java для графического представления n-разрядной схемы для вычисления логической функции в СДНФ (в виде суммы произведений). Программа получает из командной строки целое число в диапазоне от 0 до 2^{2^n} и использует двоичное представление числа для определения столбца таблицы истинности, который задает значение функции.

7.3.22. *Построение схемы сумматора.* Разработайте программу Java для построения схемы n-разрядного сумматора в соответствии с иллюстрациями в тексте. Используйте решение из предыдущего упражнения.

7.3.23. *Компаратор.* Спроектируйте схему с двумя n-разрядными входами и одним выходом, который равен 1, если при интерпретации входных значений в виде двоичных целых чисел первое число меньше второго. Приведите формулу для количества логических элементов, необходимых для вашего решения, как функции от n; изобразите свою схему для $n = 4$.

7.3.24. *Функционально полный набор логических элементов.* Набор логических элементов называется *функционально полным*, если любая логическая функция может быть реализована схемой, использующей только проводники и элементы из этого набора. Наша конструкция схемы с логической суммой произведений показывает, что обобщенные многовходовые логические элементы универсальны (что не удивительно, если учесть, сколько существует разных типов логических элементов). Считая, что NOT имеет только один вход, а все остальные элементы имеют только двухвходовые версии, покажите, что все наборы из следующего списка, кроме одного, являются универсальными; докажите, что набор-исключение не универсален.

a. NOT и AND

b. NOR

c. NAND

d. AND и OR

e. NOT и OR

f. AND и XOR

7.3.25. *Универсальность переключателей.* Докажите, что переключатели и логические элементы являются эквивалентными моделями для комбинационных схем в том смысле, что для любой заданной сети проводников, переключателей «вкл/выкл» и переключателей «вход/выход» возможно построить связную сеть логических элементов для получения тех же выходных значений (то есть вычисления той же логической функции), и наоборот. *Подсказка*: это доказательство равносильно добавлению AND NOT в список из предыдущего упражнения.

7.3.26. *Сдвиг вправо.* Постройте 4-разрядную схему для выполнения сдвига вправо. Используйте входы $x_0x_1x_2x_3$, выходы $z_0z_1z_2z_3$, и управляющие линии $s_0s_1s_2s_3$ (ровно

одна управляющая линия находится в состоянии 1, определяя величину сдвига). Ваше решение должно базироваться на следующих четырех логических формулах:

$z_0 = x_0s_0$

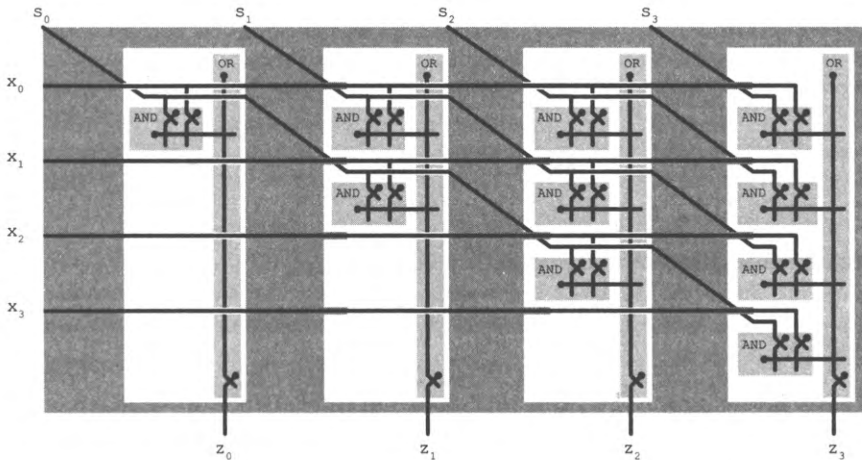
$z_1 = x_0s_1 + x_1s_0$

$z_2 = x_0s_2 + x_1s_1 + x_2s_0$

$z_3 = x_0s_3 + x_1s_2 + x_2s_1 + x_3s_0$

Из этих формул следует, что $z_0z_1z_2z_3$ содержит $x_0x_1x_2x_3$, если выбрана управляющая линия s_0; $0x_0x_1x_2$, если выбрана управляющая линия s_1; $00x_0x_1$, если выбрана управляющая линия s_2; $000x_0$, если выбрана управляющая линия s_3.

Решение



4-разрядная схема для сдвига вправо

7.3.27. *Анализ схемы для сдвига вправо.* Создайте копию схемы для сдвига вправо из упражнения 7.3.26. Обведите линии, находящиеся в состоянии 1, когда схема выполняет вычисление 1001 >> 2 = 0010.

7.3.28. *Сдвиг влево.* В стиле описания схемы сдвига вправо из упражнения 7.3.26 приведите логические формулы для выходов $z_0z_1z_2z_3$ в схеме сдвига влево с использованием $x_0x_1x_2x_3$ (входные значения) и $t_0t_1t_{2t}t_3$ (выбор величины сдвига). Затем покажите, как добавить логические элементы «под диагональю» в схеме сдвига вправо, чтобы сдвиг мог выполняться в любую сторону. Предполагается, что не более чем одна из восьми управляющих линий $s_0s_1s_2s_3$ и $t_0t_1t_2t_3$ находится в состоянии 1.

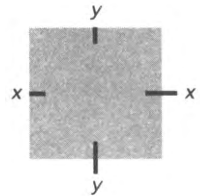
7.3.29. *Арифметический сдвиг.* Для положительных целых чисел сдвиг вправо эквивалентен делению на степень 2, а сдвиг влево — умножению на степень 2.

Спроектируйте схему сдвига, которая обеспечивает правильное умножение и деление для двоичных целых чисел, которые могут быть как положительными, так и отрицательными (и представляются в дополнительном коде). Включите выход для переполнения. Изобразите свою схему на уровне абстракции устройства для $n = 4$.

7.3.30. *Подсчет единиц.* Спроектируйте схему, которая получает на входе n битов и выдает на выходе двоичное представление количества 1 среди входных данных. Приведите формулу для количества логических элементов, необходимых для вашего решения, и изобразите схему на уровне абстракции логических элементов для $n = 4$.

7.3.31. *Умножение.* Спроектируйте схему для умножения двух n-разрядных двоичных чисел, основанного на чередовании операций сдвига и сложения. Приведите формулу для количества элементов, необходимых для вашего решения, и изобразите схему на уровне абстракции логических элементов для $n = 4$.

7.3.32. *Компьютер на плоскости.* Покажите, что любую схему можно изобразить в виде множества взаимосвязанных логических элементов на плоскости без пересечения проводников. Используйте *кроссоверные* схемы, реализующие пересечения с помощью логических элементов. У таких схем вход x находится на левой стороне, а вход y на верхней стороне; выход x располагается на правой, а выход y — на нижней стороне.



7.4. Последовательностные схемы

В этом разделе рассматриваются схемы с *обратной связью* (то есть схемы, в которых соединения образуют *циклы* — от выходов некоторых компонентов к их же входам). Вообще говоря, мы рассмотрим очень небольшое подмножество возможностей, так как обратная связь создает гораздо больше сложностей, чем добавление циклов и ветвлений в программу на Java. В наших схемах для обратной связи будут устанавливаться серьезные ограничения.

Сначала мы рассмотрим крошечные схемы (похожие на логические элементы) с элементарными обратными связями, в которых задействованы всего два переключателя. Примечательно, что это наделяет схемы *памятью*, что существенно расширяет наши возможности при построении схем. Мы сосредоточимся на добавлении в такие схемы *управляющих линий*, которые позволяют точно управлять значениями, хранимыми в памяти. Все схемы, которые рассматриваются в этом разделе, ограничиваются одной элементарной обратной связью. Разобравшись в том, как это работает, вы поймете, как реализована память в вашем компьютере. Память состоит из слов, которые состоят из битов, а в основе каждого бита лежит обратная связь.

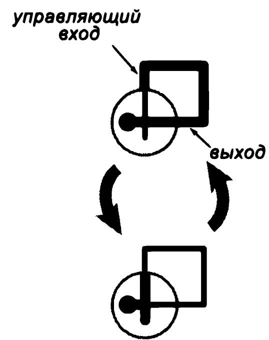
В следующем разделе мы рассмотрим *макро*циклы обратной связи между крупными компонентами схем. Количество таких связей в вычислительном устройстве невелико, и мы можем точно управлять их поведением.

Элементарные схемы с обратной связью

Для начала рассмотрим две простейшие схемы с обратной связью. Первая схема состоит из одного переключателя «вкл/выкл»; во второй схеме два переключателя.

Автогенератор (зуммер)

Рассмотрим схему, изображенную справа: выход переключателя «вкл/выкл» подается на управляющий вход. Если выход находится в состоянии 1 (вверху), то управляющий вход тоже находится в состоянии 1, в результате чего выход переключателя переходит в состояние 0. Но как только переключатель выполнит свою функцию (внизу), управляющий вход тоже перейдет в состояние 0, а переключатель снова перейдет в состояние 1 (вверху). Возникает нестабильность: переключатель попеременно меняет значение на выходе (и управляющем входе) с максимально возможной частотой. Такая схема называется *автогенератором*. Старые дверные звонки на электромагнитных реле и так называемые зуммеры были устроены именно таким образом: когда выход реле соединяется с управляющим входом, из-за переключения контакта возникает звуковой сигнал.

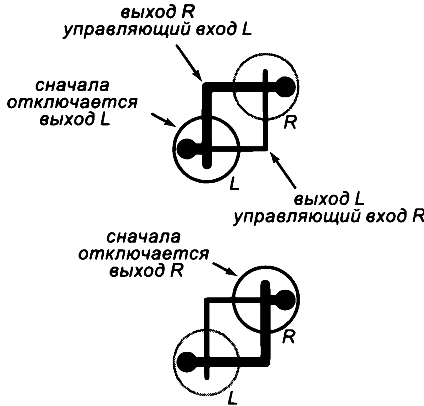


Автогенератор (зуммер)

Бистабильная схема с обратной связью

А теперь рассмотрим обратную связь на схеме слева. Она состоит из двух связанных переключателей «вкл/вкл», причем выход каждого переключателя является управляющей линией другого. Такой способ соединения называется *перекрестной обратной связью*. Как видно из рисунка, схема с такой связью может находиться в одном из двух состояний в зависимости от очередности срабатывания переключателей. Допустим, питание отключено (все проводники находятся в состоянии 0), а затем оно включается. Верхнее изображение на с. 1012 показывает, что произойдет, если питание переключателя L включится раньше переключателя R (даже на ничтожно малый промежуток времени): по команде с управляющей линии переключатель L переводит свой выход в состояние 0; это означает, что управление переключателя R переходит в состояние 0, а значит, выход переключателя R переходит в состояние 1, что поддерживает переключатель L закрытым (его выход остается в состоянии 0) — состояние устойчиво. Нижнее изображение на с. 1012 показывает, что произойдет, если первым включится питание переключателя R (по той же логике). В любом случае схема сохраняет свое состояние: после подачи

питания на первый переключатель ничего не изменяется. Вообще говоря, все источники питания должны включаться одновременно, но мы никак не управляем тем, в каком состоянии окажется схема. Первое, что нужно сделать, — добавить управляющие линии, чтобы исправить ситуацию.



Перекрестная обратная связь

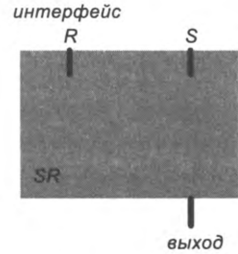
Эти два примера демонстрируют два принципиально разных варианта поведения, возможных при введении обратной связи в схему с переключателями. Это позволяет без особых усилий строить схемы с исключительно сложным поведением, поэтому мы будем следить за тем, чтобы строить компоненты нарастающей сложности из меньших, хорошо понятных компонентов, как это делалось с комбинационными схемами. К счастью, как будет показано в этом разделе, добавление проводников, управляющих простейшей бистабильной схемой — двумя переключателями с перекрестной обратной связью, — предоставляет ровно столько гибкости, сколько необходимо для построения микросхем компьютерной памяти.

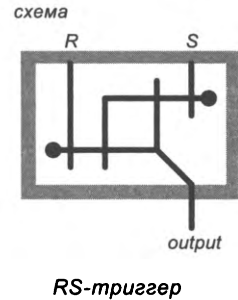
Триггеры

Добавляя в нашу схему с перекрестной обратной связью два управляющих входа, можно создать *триггер* — базовую реализацию запоминающей ячейки на один бит. Управляющие линии делают именно то, что следует из их названия: они позволяют управлять тем, в каком из двух состояний находится схема. Когда мы хотим перевести схему в одно из состояний, мы активизируем одну из управляющих линий; чтобы схема находилась в другом состоянии, нужно активизировать другую управляющую линию. Конечно, одно из этих состояний интерпретируется как 0, а другое — как 1; таким образом, добавление двух управляющих линий реализует один бит памяти, который при помощи управляющих линий можно перевести в состояние 0 или 1.

Такое поведение реализуется на удивление просто, как видно из схемы справа на с. 1013. Мы добавляем управляющую линию перед каждым из источников питания

и размещаем выход на нижней границе (в произвольной точке). По традиции управляющие линии помечаются S (Set — установка) и R (Reset — сброс). Традиционно это подразумевает элементы NOR с перекрестной обратной связью (см. упражнение 7.3.1), а сама схема называется *RS-триггером*. Теперь рассмотрим это поведение на уровне абстракции переключателей.





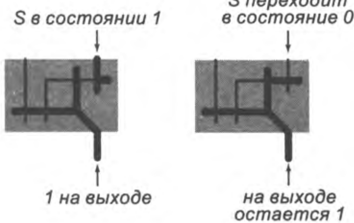
RS-триггер

Установка

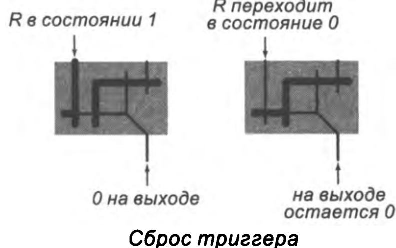
Чтобы установить триггер (то есть перевести его в устойчивое состояние с выходным значением 1), мы просто устанавливаем значение управляющей линии S равным 1 в течение времени, достаточного для включения переключателя, блокирующего соединение с источником питания справа. На выходной проводник подается значение 1, где оно и остается даже в том случае, если управляющая линия S перейдет в состояние 0. Эта ситуация изображена вверху на иллюстрации слева. *Убедитесь в том, что вы понимаете ее,* — она играет ключевую роль для понимания того, как ваш компьютер запоминает информацию.

Сброс

Чтобы сбросить триггер (то есть перевести его в устойчивое состояние с выходным значением 0), мы подаем на управляющую линию R значение 1 в течение времени, достаточного для включения переключателя, блокирующего соединение с источником питания слева. На выходном проводнике появляется значение 0, и оно остается, даже когда управляющая линия R перейдет в состояние 0. Эта ситуация на иллюстрации слева изображена внизу.



Установка триггера



Сброс триггера

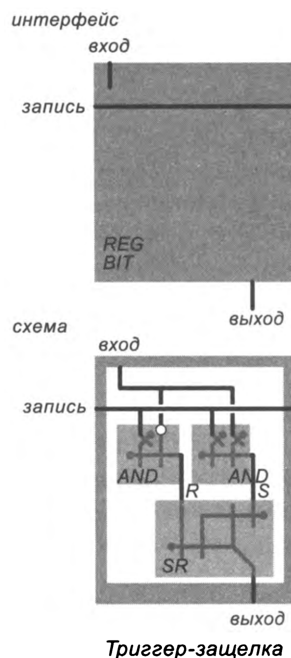
И хотя управляющие сигналы R и S часто могут одновременно быть равны 0, мы никогда одновременно не переводим их в состояние 1. Это приведет к состоянию, в котором мы не управляем значением, которое окажется на выходе. Такое состояние называется *состоянием гонки* (два переключателя соревнуются, кто первым успеет переключиться), оно нежелательно, и его следует избегать.

Истинная ценность триггеров заключается не в их простоте, а в том, что каждый триггер является *запоминающей ячейкой*, хранящей *один бит* информации — фундаментальная

абстракция для вычислений[1]. Такая реализация битов памяти невероятно проста, и именно триггеры десятилетиями оставались основой для построения памяти компьютеров. А теперь мы добавим дополнительные управляющие линии, которые позволяют использовать отдельные биты памяти для реализации регистров и основной памяти нашего процессора.

Регистры

На следующем уровне абстракции мы добавим еще две управляющие линии, чтобы построить на базе RS-триггера *триггер-защелку*, также называемый *D-триггером*. Несколько таких триггеров, объединенных в цепочку, образуют *регистр*. Регистры являются важнейшими компонентами любого компьютера; в частности, они используются и в машине TOY — вспомните регистры PC и R[0]–R[F].



Триггер-защелка

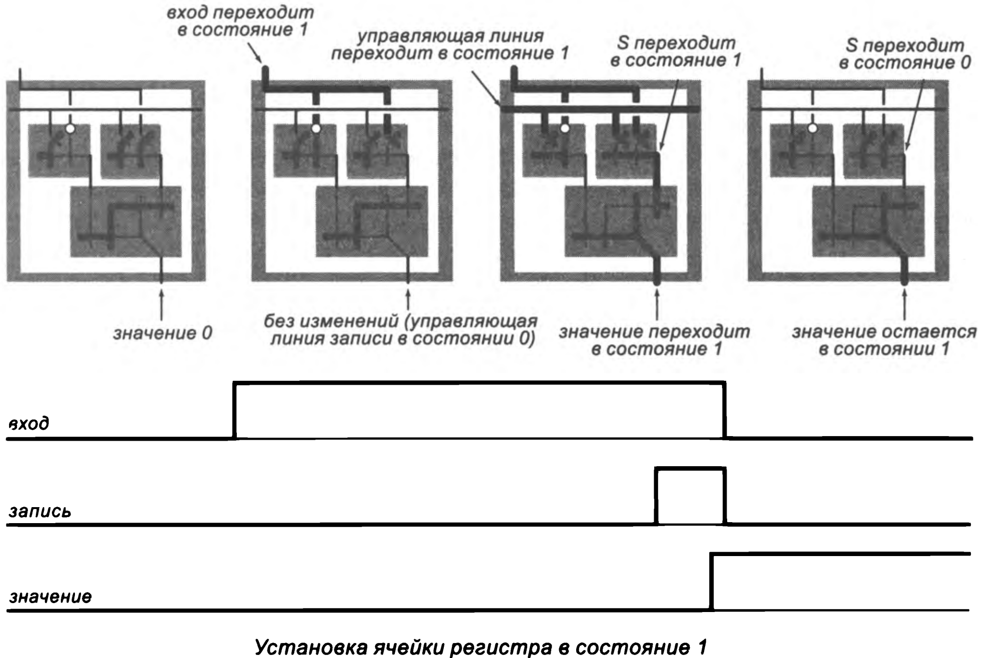
Триггер-защелка

Мы хотим реализовать следующую функциональность: сначала мы передаем сохраняемое значение на единственный выход. Затем управляющая линия записи используется для точного управления временем изменения значения триггера. Справа изображена схема, реализующая это поведение. Как видите, эта схема управляет сигналами S и R следующим образом.

- S находится в состоянии 1 в том и только в том случае, если управляющая линия записи находится в состоянии 1 и входной проводник тоже находится в состоянии 1.
- R находится в состоянии 1 в том и только в том случае, если управляющая линия записи находится в состоянии 1, а входной проводник находится в состоянии 0.

Значения в регистрах всегда задаются по одному принципу; как видно из следующих иллюстраций и временных диаграмм сигналов, последовательность событий очень важна. Сначала мы убеждаемся, что на входе присутствует нужное входное значение. Затем управляющая линия записи переводится в состояние 1, но только на очень короткое время — лишь для того, чтобы переключатели сработали для установки или сброса триггера («защелкивания» в нем входного значения).

[1] Далее будем называть это просто *битом памяти*. — *Примеч. науч. ред.*



Установка ячейки регистра в состояние 1

Почему же нам приходится уделять такое внимание временной последовательности событий? Потому что приходится учитывать вероятность того, что изменение выходного значения триггера может повлиять на его вход — возможно, через длинную цепь обратной связи с множеством проводников и переключателей. В самом деле, такая обратная связь становится значимой частью наших вычислительных устройств. К счастью, для управления ею в нашей схеме достаточно такого механизма записи[1].

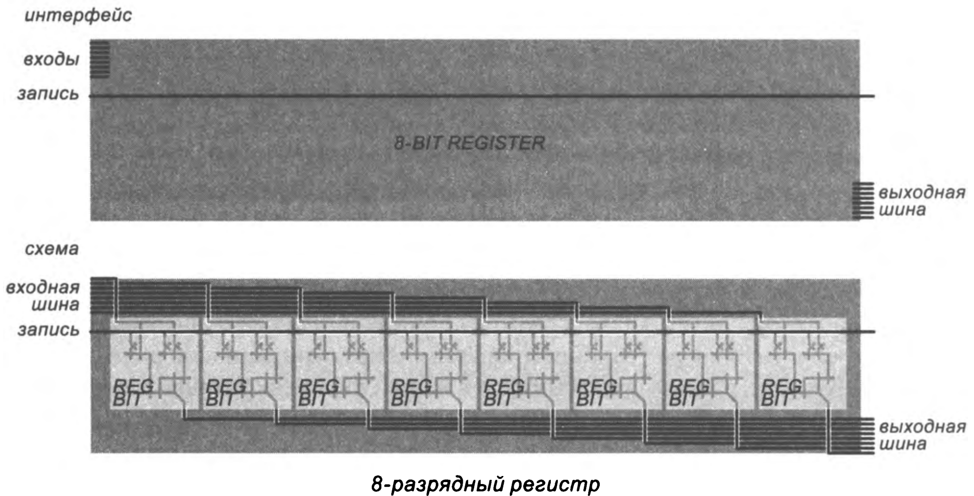
Регистры

Триггеры-защелки используются для реализации *регистров*. Пример 8-разрядного регистра изображен на с. 1016. Интерфейс состоит из 8 входных линий (слева наверху), 8 выходных линий (внизу справа) и управляющей линии записи, проходящей через весь регистр. Чтобы реализовать этот интерфейс, мы просто сцепляем *n* триггеров-защелок, соединяем их входы с верхней шиной, а выходы с нижней шиной и соединяем управляющие линии записи, проходящие через каждый триггер.

[1] Проблема неконтролируемых обратных связей и *самовозбуждения* схемы решается также использованием триггеров (и других узлов), управляемых не уровнем, а *фронтом* сигнала, но схемотехника таких узлов сложнее. — *Примеч. науч. ред.*

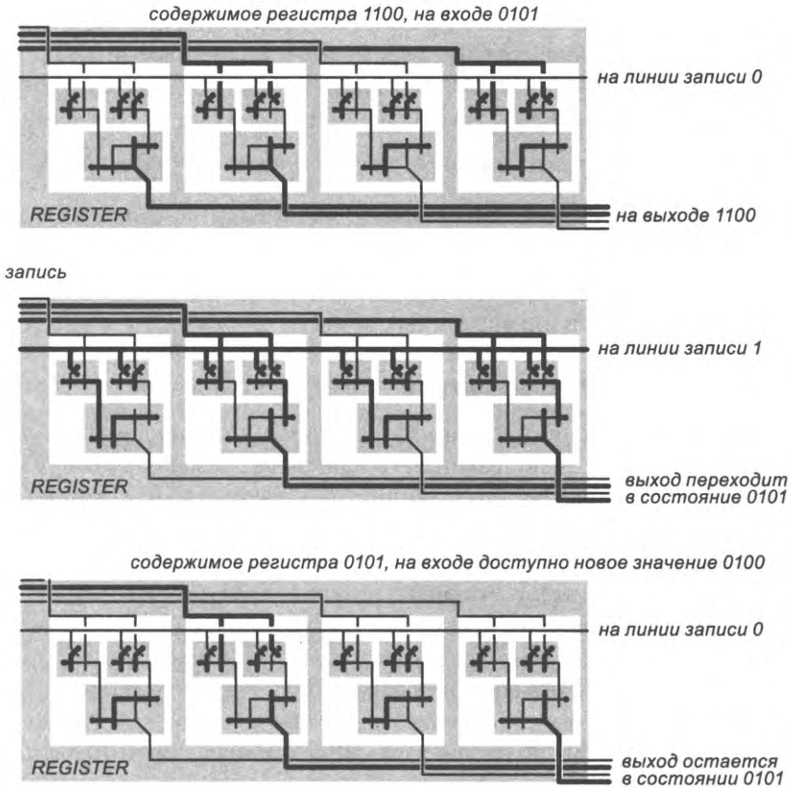
Запись в регистр

Используя сигнал записи так, как описано выше для битов памяти, мы управляем временной последовательностью изменений в значении регистра — см. изображение 4-разрядного регистра на следующей странице. В исходном состоянии регистр содержит значения 1100, а на входных проводниках находятся значения 0101. Так как управляющая линия записи находится в состоянии 0, выходные линии сохраняют состояние 1100 (содержимое регистра). Когда линия записи (на короткое время) переходит в состояние 1, как показано на среднем изображении (с. 1017), новые значения 0101 записываются в регистр и немедленно становятся доступными на выходных линиях. Когда сигнал записи возвращается в состояние 0, значение в регистре и на выходных линиях остается равным 0101, даже если на входных линиях появятся другие значения (нижнее изображение на с. 1017). Новые значения не будут записываться до тех пор, пока управляющая линия записи снова не перейдет в состояние 1.



8-разрядный регистр

Регистры играют важную роль в большинстве процессоров. Они используются для хранения внутренней информации процессора (как регистры PC и IR в нашем компьютере TOY), а также образуют память для хранения данных, используемых в арифметических операциях (например, R[0]–R[F] в TOY). Каждый регистр хранит свое значение в триггерах. Хранимые значения всегда доступны на выходных линиях. Когда по управляющей линии передается сигнал записи, значение на входных линиях записывается в триггеры регистра. Как вы вскоре увидите, этот простой интерфейс позволяет реализовать всю функциональность типичного процессора.



4-разрядный регистр с анализом переключений для сигнала записи

Память

Рассмотрим следующую последовательностную схему — *память*. С точки зрения аппаратной реализации память представляет собой просто набор последовательно адресуемых регистров. Если машина использует размер слова n, а в машинных инструкциях используются адреса разрядностью m, то память состоит из 2^m слов, каждое из которых состоит из n битов. Каждое слово очень похоже на регистр с добавлением механизма адресации/выбора[1].

[1] Здесь описывается только *статическая* память. Большинство реальных компьютеров использует в качестве основной память другого типа — *динамическую*, более емкую и дешевую. — *Примеч. науч. ред.*

Интерфейс

Память имеет $m + n$ входов (m адресных битов и n входных битов) и n выходов. Управляющий сигнал памяти управляет временной последовательностью записи, как и для регистров. Схема должна обеспечивать следующее поведение.

- Содержимое адресуемого слова всегда доступно на выходных проводниках.
- Когда по управляющей линии записи проходит сигнал, значения на входных проводниках записываются в адресуемое слово.

Память с 2^m словами обычно имеет достаточно большой объем и занимает значительную часть полезной площади микросхемы.

Разрядно-модульная архитектура

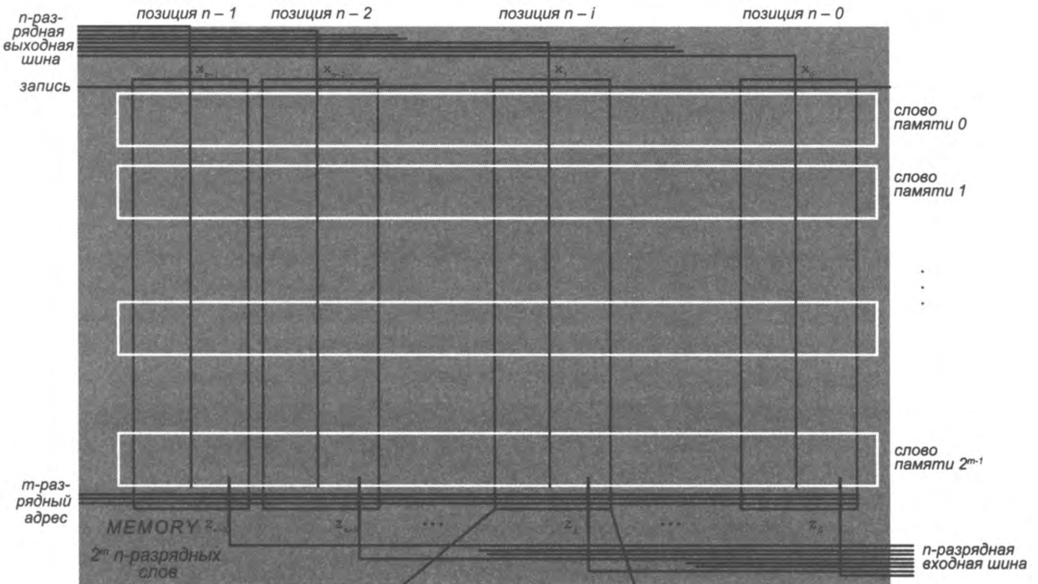
Наше решение представляет собой классическую *разрядно-модульную память*: для всех разрядов слова используются одинаковые схемы с «адресным» демультиплексором для выбора бита для записи и «адресным» мультиплексором для выбора бита для чтения (см. схему на следующей странице). Возможно, сейчас вам стоит перечитать описание «адресных» демультиплексора на с. 985 и мультиплексора на с. 986. Для каждого разряда эти биты образуют вертикальный столбец, с демультиплексором слева и мультиплексором справа. Одни и те же адресные входы управляют как демультиплексором, так и мультиплексором; таким образом, в любой момент времени активен только один бит — относящийся к адресуемому слову (активен в том смысле, что на него влияет вход или он сам влияет на выход). Соединяя n таких схем, мы получаем n-разрядные слова, при этом активно только одно адресуемое слово.

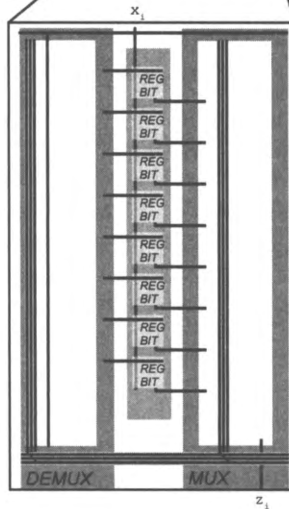
Память имеет одну управляющую линию записи, по которой передается входное значение для мультиплексора; переключение этой линии используется для выбора внутренней управляющей линии записи для каждого бита адресуемого слова. Когда в память поступает сигнал записи, он передается на каждый бит адресуемого слова (и только на эти биты), что приводит к установке или сбросу триггера каждого бита в соответствии с входным значением. Значения из памяти могут *читаться* в любой момент времени, потому что мультиплексор для каждого разряда слова всегда предоставляет текущее значение триггера адресуемого бита как выходное.

Реализация схемы

Память в такой реализации занимает слишком много места, поэтому мы построим эквивалентную схему, которая получается гораздо более компактной.

- Мы используем только один «адресный» демультиплексор (а не по одному для каждого разряда слова), а каждый из выходов проходит горизонтально по всем битам слова.





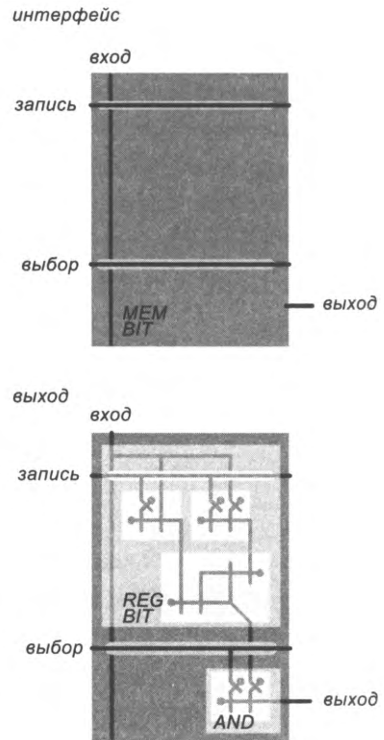
Разрядно-модульная память

- Логические элементы AND мультиплексоров интегрируются в триггеры-защелки регистров, чтобы получить схему *бита памяти* с одной управляющей линией выбора.

В результате этих изменений получается «секция» шириной приблизительно в 5 раз меньше той, которая потребовалась бы при прямой реализации (при этом, как вы увидите, в нашем процессоре она все равно заполняет целую страницу). А с учетом того, что наше решение реализует тот же интерфейс памяти, понимать все подробности не обязательно. Если тема вас заинтересует, вы найдете куда более подробную информацию в разделе «Вопросы и ответы» в конце этого раздела.

Запоминающие ячейки

Как уже упоминалось ранее, память строится из *запоминающих ячеек* — триггеров-защелок с добавлением управляющей линии выбора (для чтения). Если на линию выбора подается 0, то выход тоже находится в состоянии 0; если на линию выбора подается 1, то состояние выхода соответствует значению, содержащемуся в триггере. Чтобы реализовать это поведение, мы просто объединяем выход триггера с линией выбора операцией AND, как показано справа. Эти выходы для всех позиций подаются в вертикальный элемент OR; он завершает реализацию мультиплексора, передающего значение заданного бита на выход. Линия выбора гарантирует, что все выходные значения битов будут равны 0 (возможно, кроме выбранного).



Запоминающая ячейка на 1 бит

Схема памяти

Каждое слово памяти строится по тому же принципу, что и регистры: мы выстраиваем ячейки (триггеры) по горизонтали и соединяем управляющие линии, чтобы они проходили по горизонтали через все разряды слова. Далее память наращивается вертикально добавлением слов. Реализация памяти с четырьмя 6-разрядными словами ($m = 2$ и $n = 6$) показана в нижней части с. 1022. Мы будем обозначать слова 2-разрядными адресами: 00, 01, 10 и 11. Обратите внимание на следующие особенности реализации.

- В каждой позиции только одна вертикальная линия получает входные данные с верхней шины.
- Одна схема реализует функции дешифратора и «адресного» демультиплексора (активна только одна адресуемая пара выходов).
- Управляющие линии (записи и адреса) управляют работой дешифратора/демультиплексора, выходами которого являются управляющие линии записи и выбора, проходящие по горизонтали через каждое слово.
- Вертикальный элемент OR собирает выходы всех позиций.
- Шина в нижней части несет выходные значения, по одной линии для каждого разряда.

На схеме на с. 1022 выделено слово по адресу 10 (с раскрытием его битов) для более внимательного анализа операции записи в память. В этом примере на входной шине передается значение 001111, биты адреса содержат 10, а управляющая линия записи находится в состоянии 1; линии записи и выбора для слова памяти по адресу 10 равны 1, в результате чего триггеры устанавливаются в состояние 001111 (слева направо). Эти значения отправляются на вертикальные элементы OR и передаются на выходную шину.

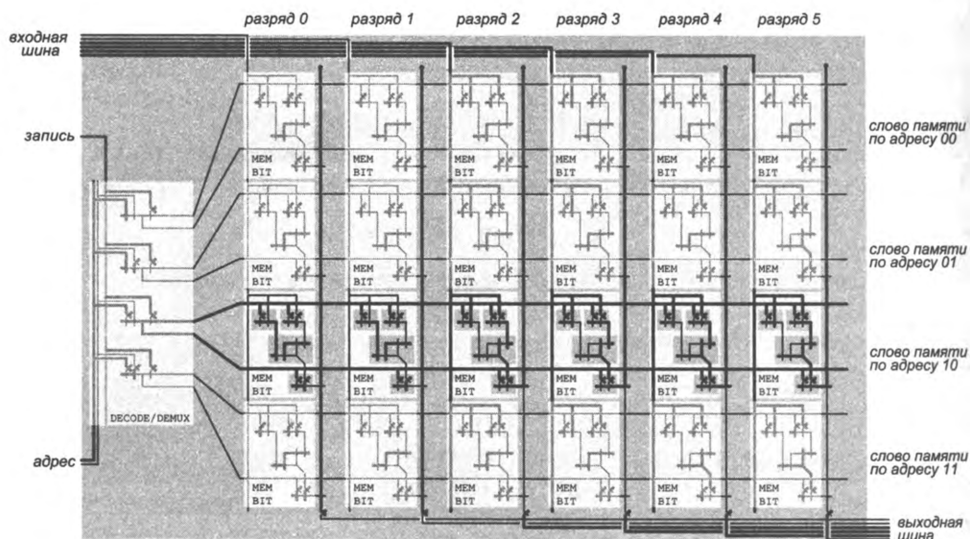
Подведем итог: на базе этого решения строятся схемы с m адресными входами, n-разрядными входной и выходной шиной и управляющей линией разрешения записи, которые обладают следующей функциональностью.

- Содержимое адресуемого слова всегда доступно на выходной шине.
- При получении сигнала записи значения с входной шины записываются в адресуемое слово.

В таблице ниже указано количество логических элементов, необходимое для реализации памяти с различными значениями этих параметров. Таблица показывает, насколько важно разработать компактную реализацию запоминающих ячеек — это быстро становится очевидно каждому, кто проектирует компьютерные схемы.

количество слов	**разрядность слов**	**количество триггеров**	**общее количество других вентилей**
4	6	24	82
8	8	64	208
256	16	4096	12 562
2^{30}	64	2^{36}	около 70 миллиардов

Количество логических элементов для памяти различного объема



Память (четыре 6-разрядных слова) с анализом переключений при получении сигнала записи

Тактовый генератор

Последним примером последовательностной схемы станет критический компонент компьютера: *тактовый генератор*. Эти компьютерные «часы» являются фундаментальной абстракцией для синхронизации всего, что происходит в вычислительных схемах. Наша схема представляет собой физическое воплощение этой абстракции. Ее главная цель — управление циклом выборки/исполнения, упоминавшимся при первом знакомстве с TOY. На самом деле, все события, происходящие в наших компьютерных схемах, синхронизируются по сигналу тактового генератора.

Такты и импульсы

Наша методология измерения синхронизации основана на генерировании периодических импульсов следующего вида.

Такие сигналы называются *тактовыми сигналами*. Обычно линия находится в состоянии 0, но она переключается в состояние 1 с заданной периодичностью. Обычно отрезок времени, когда сигнал находится в состоянии 1, называется *импульсом*, или «*тиком*». В реальном компьютере точность этого сигнала чрезвычайно важна.

Разработчики компьютеров прикладывают все усилия для создания надежных, быстрых тактовых генераторов. В наши дни компьютерные тактовые генераторы могут выдавать импульсы миллиарды раз в секунду.

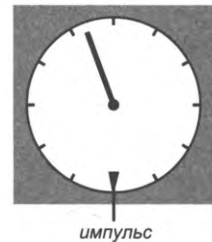
Кроме базового предположения о наличии внешнего источника, способного генерировать периодический сигнал вроде изображенного выше, мы делаем два дополнительных предположения по поводу сигнала.

- Импульс имеет достаточную продолжительность для активизации любого переключателя.
- Время от начала одного импульса до начала другого больше продолжительности самой длинной цепочки срабатываний переключателей в схеме.

Второй параметр — *тактовая частота* — безусловно критичен, потому что он, например, определяет количество операций, выполняемых за секунду. Точное значение не всегда вычисляется легко (см. упражнение 7.4.14). В реальных компьютерах значение этого параметра обычно определяется эмпирически: вы первоначально выбираете период достаточно большим, с запасом, и увеличиваете тактовую частоту, пока схема не перестает работать, потому что некоторые из ее переключателей не успевают активизироваться; после этого тактирование немного замедляется.

Как правило, тактовые генераторы компьютеров строятся на базе специализированной технологии, которая отличается от технологии, используемой для построения переключателей, — это делается для того, чтобы поднять тактовую частоту до максимума. Обычно используются колебания, обусловленные воздействием электричества, в физическом материале — *пьезоэлектрике*. Как и в случае с переключателями, конкретная связь с физическим миром выходит за рамки книги и не имеет значения для логических аспектов проектирования компьютерных схем.

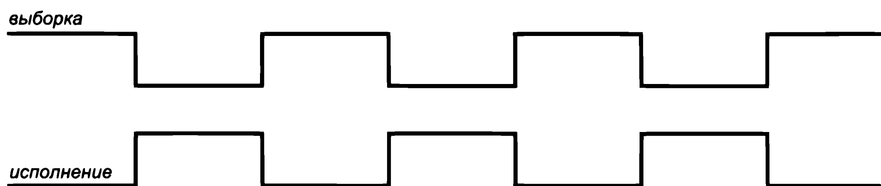
Для наглядности можете представить электрический контакт на циферблате часов, посредством которого секундная стрелка выдает сигнал продолжительностью в 1 секунду с минутными интервалами (см. справа). Реальный тактовый генератор работает в миллиарды раз быстрее, но концептуально он устроен так же. Генерируемый тактовый сигнал выглядит так же, хотя и в совершенно ином масштабе времени.

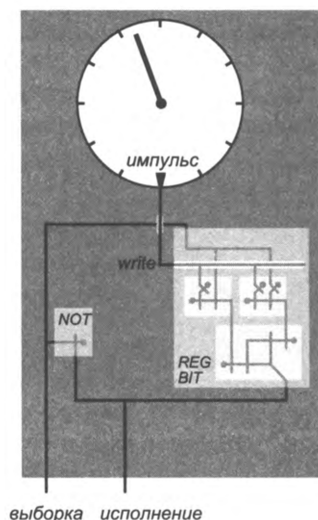


Модель тактового генератора

Выборка и исполнение

Самое важное применение тактового генератора — выдача периодических управляющих сигналов, по которым можно различить фазы выборки и исполнения при выполнении каждой машинной инструкции. Выражаясь конкретнее, сигналы должны выглядеть так (рис. 1024, вверху).





Эти два сигнала являются *дополняющими*, то есть *инверсными* по отношению друг к другу: если подать сигнал, сгенерированный для выборки, на элемент NOT, вы получите сигнал для исполнения, и наоборот. Существует много способов генерирования сигналов. Возможно, самый простой из них — связать тактовый генератор с *триггером-защелкой*, как показано справа. Управляющая линия тактового генератора соединяется с управляющей линией записи триггера, а входное значение триггера является инверсным по отношению к хранимому значению. С такими соединениями каждый импульс заставляет триггер переключиться на инверсное значение, которое удерживается до следующего импульса (полученную схему называют *T-триггером*). Выходное значение триггера используется как сигнал исполнения, а его инверсия — как сигнал выборки. Триггер «защелкивает» состояние до следующего импульса, когда его значение переключается с 0 на 1 или с 1 на 0.

Управляющие выходы записи

Как в фазе выборки, так и в фазе исполнения вычислительного цикла значения должны записываться в регистры. Более подробное описание будет приведено ниже, но этот очевидный факт требует добавления двух логических элементов в тактовый генератор. А конкретнее, необходимо добавить два управляющих выхода: записи в фазе выборки и записи в фазе исполнения, причем значения этих сигналов должны быть равны 0 все время, кроме импульсов в конце соответствующих фаз. Объединение импульса с сигналом выборки операцией AND дает импульс записи в фазе выборки, а объединение импульса с сигналом исполнения операцией AND дает импульс записи в фазе исполнения.

Входы запуска и остановки

Нашей схеме тактового генератора нужны еще две входные управляющие линии: для остановки и запуска. Вход запуска пускает тактовый генератор — представьте, что он соединен с кнопкой ПУСК на консоли. Вход остановки останавливает генератор; этот сигнал передается при выполнении программой инструкции останова.

На следующей странице изображена схема тактового генератора со всеми дополнениями. Все управляющие линии наших вычислительных схем в конечном итоге управляются одним из четырех выходов тактового генератора. На управляющих выходах схемы тактового генератора бесконечно повторяется следующий цикл.

- Линия выборки переходит в состояние 1, а линия исполнения переходит в состояние 0.
- Линия записи в фазе выборки на короткое время переходит в состояние 1.
- Линия исполнения переходит в состояние 1, а линия выборки переходит в состояние 0.
- Линия записи в фазе исполнения на короткое время переходит в состояние 1.

Как будет показано в следующем разделе, эта последовательность позволяет реализовать цикл выборки-исполнения в схеме; при этом состояние регистров и памяти изменяется в точном соответствии с требованиями архитектуры компьютера.

Тактовый генератор достойно завершает тему последовательностных схем. Он показывает, какое сложное поведение можно реализовать при помощи одного триггера и нескольких логических элементов. В то же время тактовый генератор демонстрирует необходимость жесткого контроля за обратными связями в схемах, потому что любая обратная связь может привести к непредсказуемому поведению. А ведь полная предсказуемость — важнейшее свойство вычислений!

Выводы

Как и в случае с комбинационными схемами, с этого момента мы будем работать на более высоком уровне абстракции с теми узлами и блоками, которые были определены, ограничиваясь их размерами, расположением входных и выходных шин и управляющих линий. Эта возможность в очередной раз свидетельствует о мощи абстракции: после перехода на этот уровень вам уже не нужно знать подробности внутренней реализации.

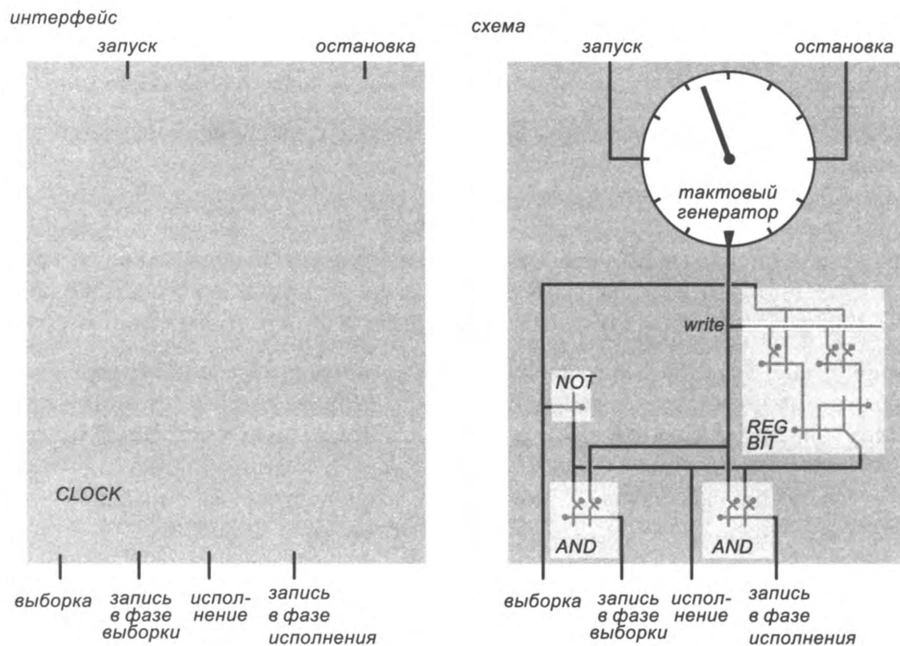
Что касается *регистров*, достаточно знать, что содержимое регистра всегда доступно на выходной шине, а импульс разрешения записи инициирует процесс записи значений с входной шины в регистр.

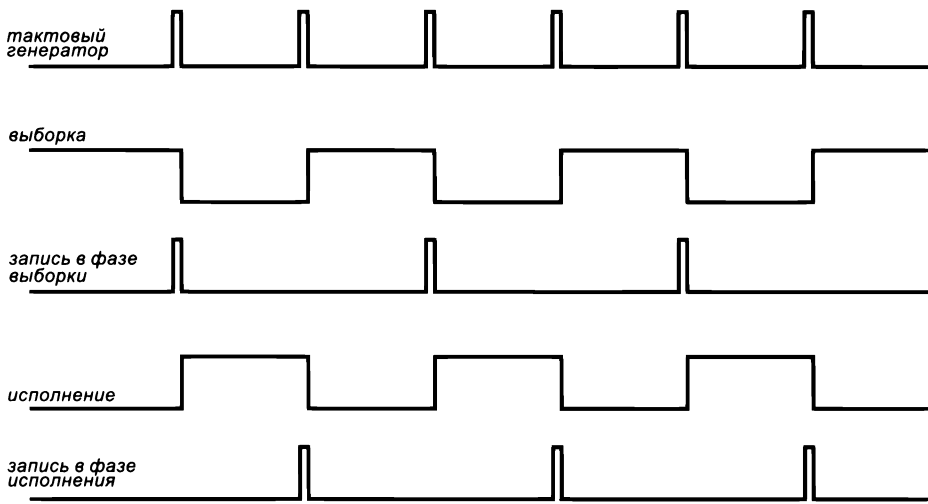
Аналогичным образом для *памяти* достаточно знать, что содержимое адресуемого слова всегда доступно на выходной шине, а импульс разрешения записи инициирует процесс записи значений с входной шины в адресуемое слово.

Для *тактового генератора* достаточно знать, что он генерирует бесконечную последовательность рассмотренных выше четырех сигналов, которые управляют циклом выборки-исполнения в компьютере.

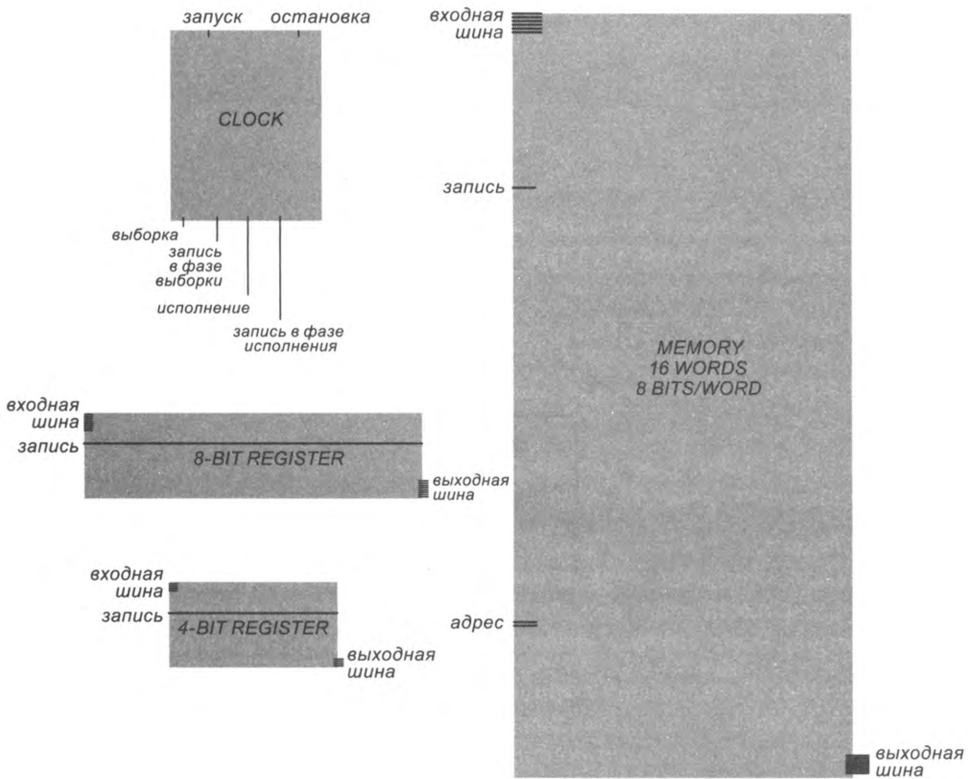
Узлы и блоки последовательностных схем, которые мы будем использовать в своем вычислительном устройстве, изображены справа: один 4-разрядный регистр, два

8-разрядных регистра, память с 16 8-разрядными словами и тактовый генератор. В следующем разделе будет рассказано, как объединить их с комбинационными схемами из предыдущего раздела (и несколькими дополнительными логическими элементами) для создания вычислительного устройства.





Цикл выборки-исполнения с импульсами записи



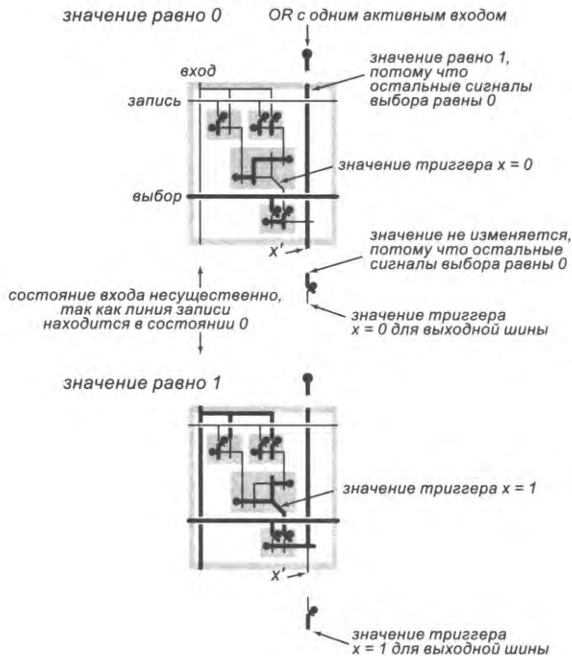
Интерфейсы функциональных блоков (для последовательностных схем)

Вопросы и ответы

В. Я все еще плохо понимаю механизм чтения значений битов памяти. Можете объяснить более подробно?

О. Подробный анализ схемы представлен на иллюстрации на с. 1028. Тем не менее лучше мыслить понятиями мультиплексирования. Если рассмотреть различия между дешифратором и адресным мультиплексором, вы увидите, что дешифратор в схеме памяти, в сочетании с логическими элементами AND триггера и вертикальным элементом OR для каждого разряда, образует мультиплексор.

В. Временные диаграммы выглядят слишком сложно. Они действительно необходимы?



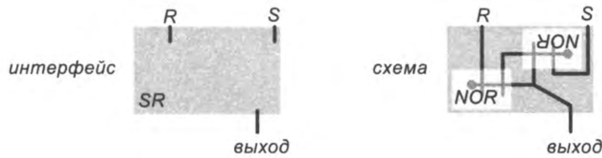
Чтение бита памяти

О. С одной стороны, слишком тщательно изучать эти диаграммы — значит слишком буквально воспринимать нашу абстрактную модель; синхронизация — это сложная задача, которая на практике решается по-разному. С другой стороны, вам будет, безусловно, полезно перечитать описание процесса установки значения триггера запоминающей ячейки (чтобы закрепить понимание работы управляющих линий записи) и сводное описание тактового генератора (чтобы закрепить понимание временной последовательности управляющих линий). К счастью, ничего сложнее этих концепций нам не понадобится — в дальнейшем мы будем работать исключительно на уровне поведения узлов и блоков, описанного выше.

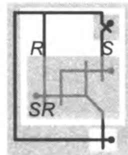
Упражнения

7.4.1. Изобразите триггер в виде пары элементов NOR с перекрестной обратной связью.

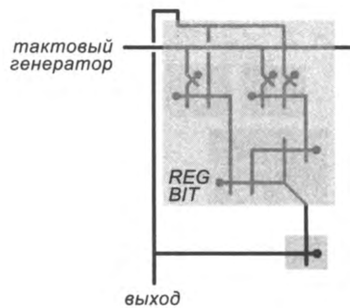
Решение. (Занимает чуть больше места по сравнению с нашей прежней реализацией.)



7.4.2. Опишите поведение этой схемы. Являются ли ее состояния устойчивыми?



7.4.3. Опишите поведение этой схемы.



Решение. Выход переключается между 0 и 1, изменяя свое состояние при каждом импульсе от тактового генератора.

7.4.4. Постройте автогенератор (схему с обратной связью и нестабильным состоянием) с тремя переключателями.

7.4.5. Используя восемь триггеров для битов памяти и дешифратор/демультиплексор, постройте схему памяти для реализации четырех 2-разрядных слов.

7.4.6. Используя восемь триггеров для битов памяти и дешифратор/демультиплексор, постройте схему памяти для реализации двух 4-разрядных слов.

7.4.7. Выведите формулу для количества логических элементов в реализации памяти с 2^m n-разрядных слов. Используйте свою формулу для проверки данных в таблице на с. 1021.

Упражнения повышенной сложности

7.4.8. *Анализ схем.* Приведите количество переключателей для реализации схем из следующего списка.

a. Триггер-защелка (в регистре).

b. Триггер запоминающей ячейки в памяти.

c. 8-разрядный регистр.

d. Память с 16 8-разрядными словами.

e. Тактовый генератор.

При подсчете следует учитывать каждый переключатель (например, 2-входовый логический элемент AND содержит четыре переключателя).

7.4.9. *Схема регистра.* Напишите программу Java, которая получает целое число n из командной строки и строит схему n-разрядного регистра.

7.4.10. *Схема регистра (продолжение).* Доработайте свое решение предыдущего упражнения, чтобы программа получала второй аргумент командной строки x и строила схему регистра, в котором хранится двоичное представление x (с сигналом записи 1). Если программа получает третий аргумент командной строки y, она строит схему регистра с сигналом записи 0 и подает двоичное представление y на входные проводники.

7.4.11. *Схема памяти.* Напишите программу Java, которая получает целые числа m и n в аргументе командной строки и строит схему памяти на 2^m n-разрядных слова.

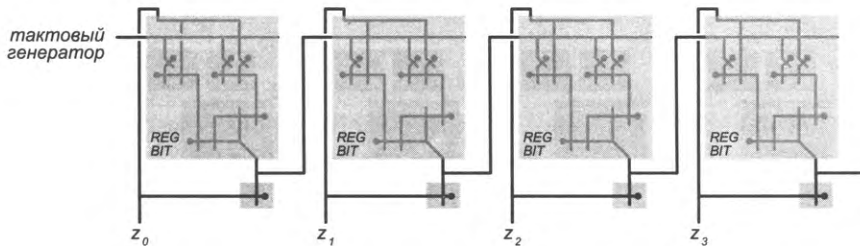
7.4.12. *Схема памяти (продолжение).* Доработайте свое решение предыдущего упражнения, чтобы программа получала 2^m целых числа из стандартного ввода и строила схему памяти, в которой хранятся эти целые числа. Затем добавьте функцию визуального представления записи нового значения в память.

7.4.13. *Двухпортовая память.* Спроектируйте память с двумя выходными шинами, двумя наборами адресных проводников и двумя управляющими линиями выбора для чтения; память должна представлять содержимое двух слов памяти на выходных шинах.

7.4.14. *Задержка.* Приведите длину самой длинной цепочки переключателей, активизируемой при подаче импульса записи на память с 2^m n-разрядными словами.

7.4.15. *Тактовые генераторы.* Опишите, как построить схему тактового генератора (m, n), которая определяется следующим образом. Предполагается, что все переключатели срабатывают за фиксированное время x, а проводники переходят в состояние 1 по всей длине при соединении с источником питания. Тактовый генератор (m, n) генерирует сигнал 0 в течение mx секунд и сигнал 1 в течение nx секунд с циклическим повторением каждые $(m + n)x$ секунд.

7.4.16. *Двоичный счетчик.* Опишите поведение следующей схемы при поступлении на вход серии тактовых импульсов.



Решение. Это *счетчик.* Значения $z_3z_2z_1z_0$ представляют двоичное значение — каждый импульс увеличивает это двоичное значение. См. упражнение 7.4.3.

7.4.17. *Циклический счетчик*[1]. Спроектируйте схему с управляющими входами сброса и записи и шестью регистровыми битами с выходами z_0–z_5 со следующим поведением: сигнал сброса переводит z_0 в состояние 1, а все остальные триггеры в состояние 0, после чего последовательные импульсы записи переводят $z_0z_1z_2z_3z_4z_5$ в состояния 100000, 010000, 001000, 000100, 000010, 000001, 100000, ...

7.4.18. *LFSR.* Спроектируйте схему, которая реализует регистр сдвига с линейной обратной связью в соответствии с описанием из упражнения 7.1.15. А именно постройте схему с 12 регистровыми битами, пронумерованными от 0 до 11 справа налево, которая выдает данные из бита 0 и получает данные с входной шины. На управляющий вход подается тактовый сигнал. По каждому тактовому импульсу бит 0 заполняется результатом операции XOR с битами 11 и 9, а остальные значения сдвигаются влево на один разряд; значение, выходящее из бита 11, игнорируется. Такое устройство может инициализироваться 11-разрядным исходным значением, а затем генерирует «случайный» бит при каждом импульсе от генератора.

7.5. Цифровые вычислительные устройства

Остается сделать последний шаг — связать воедино все блоки, рассмотренные в двух предыдущих разделах, и организовать управление ими. Этой теме посвящен данный раздел.

Первый аспект проблемы — вычисления, второй — память... но есть и третий: управление. На долю АЛУ, шинных мультиплексоров, регистров и памяти приходится более 99% логических элементов в процессоре, поэтому в каком-то смысле работа почти выполнена. Но в другом смысле мы только сейчас подходим к ответу на во-

---

[1] Сохранен буквальный перевод (ring counter). Описанный порядок функционирования соответствует *циклическому регистру сдвига. — Примеч. науч. ред.*

прос «Как работает компьютер?», потому что эти оставшиеся элементы и управляют тем, как и когда информация проходит через компьютер. Именно они реализуют абстракцию «исполнения программы», которая лежит в основе вычислений.

TOY-8

Для начала определимся с целью: мы хотим спроектировать процессор для младшего представителя семейства воображаемых компьютеров TOY. Вся основная информация о TOY-8 уже была представлена ранее; остается привести некоторые подробности.

Базовые параметры

Наша настоящая цель — учебная; мы хотим представить схему, которая реализует целый законченный компьютер, на двух страницах книги в масштабе, при котором виден каждый переключатель. Соответственно TOY-8 — очень маленький компьютер, оперирующий 8-разрядными словами, с 16 словами памяти и одним регистром. Несмотря на столь жесткие ограничения, на TOY-8 можно решать достаточно сложные вычислительные задачи (хотя мы не будем отвлекаться на это).

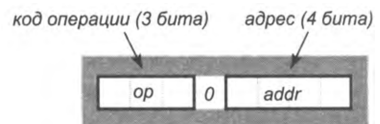
Набор инструкций

Так как регистр только один, формат инструкций TOY-8 заметно отличается от форматов TOY. А именно...

- Каждая инструкция неявно подразумевает обращение к единственному регистру, поэтому явные обращения к регистру не нужны.
- Для адресации слова памяти необходимо 4 бита.
- Для кода операции необходимо 3 бита.
- Один бит не используется (и всегда равен 0).

Формат всех инструкций TOY-8 изображен справа.

В этих решениях воплощены факторы, которые необходимо учитывать в архитектуре набора инструкций. Это был важный аспект проектирования ранних компьютеров. Допустим, вы еще не решили, будет ли следующая версия компьютера иметь 16 различных инструкций или 32 слова памяти. Можно представить, что проектировщик на начальном этапе работы оставляет неиспользуемый бит для того, чтобы принять это решение позже. Мы в данном случае пытаемся просто разработать по возможности более простой компьютер, демонстрирующий основные характеристики реальных компьютеров, поэтому ограничимся 8 инструкциями и 16 словами памяти.



Структура инструкций TOY-8

код операции	шестнадцатеричная запись	описание	псевдокод
0000	0	останов	
0010	2	сложение	R = R + M[addr]
0100	4	поразрядная операция AND	R = R & M[addr]
0110	6	поразрядная операция XOR	R = R ^ M[addr]
1000	8	загрузка адреса	R = addr
1010	A	загрузка данных	R = M[addr]
1100	C	сохранение данных	M[addr] = R
1110	E	переход по нулю	if (R == 0) PC = addr

Набор инструкций TOY-8

Полный набор инструкций TOY-8 приведен в таблице вверху. В нем не хватает инструкций вычитания, сдвига, косвенных обращений и трех альтернативных инструкций перехода (передачи управления), присутствующих в наборе TOY, но он содержит все основные инструкции, необходимые для реализации арифметических операций, условных переходов и циклов, которые лежат в основе вычислений.

Так как неиспользуемый бит всегда равен 0, коды операций всегда четны даже в том случае, если для кодирования инструкции используется шестнадцатеричное число из двух цифр. Например, инструкция `2E` означает «прибавить к `R` содержимое ячейки памяти `E`», а `CE` — «сохранить `R` в ячейке памяти `E`».

Предполагается, что ячейка памяти `F` соединена со стандартным вводом/выводом, как и в случае TOY. Также предполагается, что ячейка памяти `0` всегда содержит `0`.

Программа для TOY-8

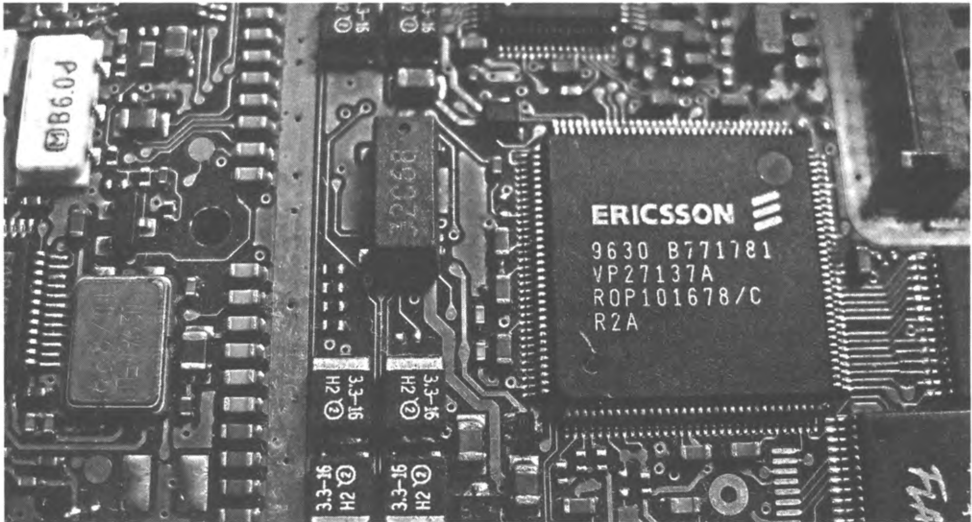
Для примера ниже приведена версия программы для вычисления суммы чисел в стандартном вводе (листинг 6.3.3) для машины TOY-8:

```
1  A0  R = 0
2  CE  M[E] = R             int sum = 0
3  AF  R = stdin            while (!StdIn.isEmpty())
4  E9  if (R == 0) PC = 9   {
5  2E  R = R + M[E]            c = StdIn.readInt()
6  CE  M[E] = R                if (c == 0) break
7  A0  R = 0                   sum = sum + c
8  E3  if (R == 0) PC = 3   }
9  AE  R = M[E]
A  CF  stdout = R           StdOut.println(sum)
B  00  halt
```

Обратите внимание: регистр (R) явно перегружен работой. Например, чтобы выполнить безусловный переход, необходимо сначала загрузить 0 в R, а потом выполнить переход по 0. Именно необходимость писать такой код заставила уже разработчиков первых компьютеров предусмотреть несколько регистров. Также интересно рассмотреть вопрос о том, что предпочтительнее — 32 слова памяти или еще 16 инструкций? Конечно, с точки зрения проектирования расширить память несложно, но в мире, в котором с каждым битом были связаны немалые затраты, возникали серьезные проблемы. С другой стороны, добавление новых инструкций требует разработки дополнительных схем, что тоже непросто.

Все эти подробности сейчас не так важны, потому что нас интересует не программирование для TOY-8, а разработка схемы, реализующей TOY-8. Главный вывод, который следует сделать из этого обсуждения: машина TOY-8 похожа на TOY и на настоящие машины, отличаясь от них только масштабом. Как при расширении памяти, так и при расширении набора инструкций мы сможем реализовать все программы из предыдущей главы на такой машине, как TOY-8; таким образом, мир вычислений TOY-8 по многим фундаментальным характеристикам не отличается от мира современных компьютеров.

Здесь не рассматриваются физические характеристики компьютеров: клавиатуры, мониторы, аккумуляторы, источники питания и т. д. Нас интересует архитектура вычислительной схемы — процессора. Если вы заглянете внутрь своего компьютера, там где-то найдется большая квадратная микросхема — это и есть процессор. А теперь разберемся, что же находится внутри него.



Центральный процессор

Разминка

Для разминки мы начнем с цифрового устройства, находящегося в основе любого компьютера: счетчика адреса программы (PC). Вспомните, о чем говорилось в главе 6: счетчик предназначен для хранения адреса текущей выполняемой инструкции. Его значение может изменяться одним из двух способов: либо в результате инкремента (чаще всего), либо с явной записью нового значения (при переходе).

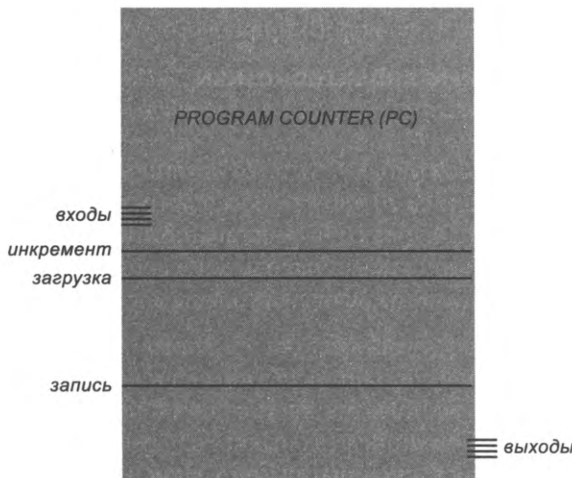
Счетчик адреса программы наглядно демонстрирует важнейшие факторы, которые должны учитываться при реализации любого цифрового устройства. Вы должны ответить на следующие вопросы:

- Какие блоки вам потребуются?
- Как информация передается из одного блока в другой?
- Какие понадобятся управляющие линии, какими временными характеристиками должны обладать управляющие сигналы?

Все эти вопросы кратко уже рассматривались выше, когда мы занимались построением блоков регистров и основной памяти. Теперь мы рассмотрим каждый из них для более сложного цифрового устройства — PC. На с. 1037 показан конечный результат.

Интерфейс

Главная функция PC — хранение адреса текущей инструкции, поэтому можно ожидать, что этот адрес всегда должен быть доступен на выходной шине. Значе-



Интерфейс блока PC

ние PC может изменяться двумя способами: либо в результате инкремента, либо записью нового значения (для инструкции перехода). Следовательно, понадобятся два управляющих входа (инкремент и загрузка), а также шинный вход для адреса перехода. Еще нужна линия записи для управления изменением значения PC. Этот интерфейс изображен на с. 1035.

Модули

Потребуются как минимум два узла:

- регистр для хранения адреса;
- инкрементор для увеличения адреса на 1.

Впрочем, это только начало. Необходимость еще одного узла будет рассмотрена при обсуждении механизмов взаимодействия между ними.

Шинные соединения

Чтобы информация могла передаваться между узлами, их необходимо связать проводниками. Большинство таких проводников является частью *шинных соединений* — обычно выходная шина одного узла соединяется со входной шиной другого, по одному проводнику на бит. А поскольку в нашей схеме имеется регистр и инкрементор, очевидно, потребуются как минимум четыре шинных соединения[1]:

- от регистра PC к инкрементору (увеличиваемое число);
- от инкрементора к регистру PC (результат);
- от входов PC к регистру PC (новое значение в случае перехода);
- от регистра PC к выходам PC (значение, используемое для выборки инструкции).

Описания получаются краткими, потому что шинные соединения всегда соединяют выходную шину одного узла со входной шиной другого. Например, «От регистра PC к инкрементору» означает «От выходной шины регистра PC к входной шине инкрементора».

Из этого списка видно, что регистр PC имеет два входа и два выхода. Однако между входом и выходом существует асимметрия. Его выход можно разделить между двумя разными входами при помощи простого T-обратного соединения — после разделения на обеих шинах передаются одни и те же значения. С другой стороны, соединить выходы двух разных узлов на входе третьего не удастся; по проводникам передаются разные значения, поэтому придется использовать переключатель (шинный мультиплексор). Соответственно мы добавляем шинный мультиплексор

[1] Широко применяется концепция *общей шины*: подключение многих узлов к одной и той же шине параллельно. При этом необходим *арбитраж* шины — управление моментами доступа каждого из узлов к ней. — *Примеч. науч. ред.*

«2-в-1» и заменяем второе и третье шинное соединение в приведенном ранее списке следующими:

- от инкрементора к шинному мультиплексору (вход 0);
- от входа блока PC к шинному мультиплексору (вход 1);
- от шинного мультиплексора к регистру PC.

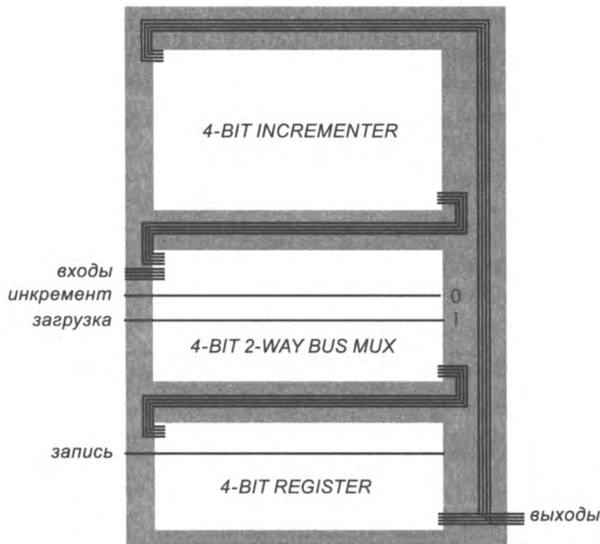


Схема счетчика адреса программы

Соответствующие магистрали изображены справа. Для наглядности мы придерживаемся соглашения о том, что шинные выходы отходят от правого нижнего угла узла, а шинные входы присоединяются к левому верхнему углу. Мы могли бы сделать эти магистрали короче, соединяя оба узла с одной стороны, но тогда вам было бы труднее различать входы и выходы, а сами шины пришлось бы переворачивать для сохранения порядка битов.

Управляющие линии

Управляющие линии связываются с используемыми узлами и в точности соответствуют управляющим линиям в описанном интерфейсе.

- Две линии (инкремент и загрузка) управляют шинным мультиплексором, позволяя выбрать между увеличением и загрузкой PC.
- Линия записи инициирует процесс загрузки регистра.

Как и для любого шинного мультиплексора, линии инкремента и загрузки никогда не устанавливаются в состояние 1 одновременно. Как и для любой схемы памяти,

ничего не происходит до поступления импульса записи. Если в этот момент сигнал инкремента находится в состоянии 1 (а сигнал загрузки находится в состоянии 0), содержимое регистра увеличивается на 1; если же при поступлении импульса сигнал загрузки находится в состоянии 1 (а сигнал инкремента в состоянии 0), то значение на входной шине PC загружается в регистр.

Обратите внимание на наличие длинной цепи обратной связи в шинных соединениях этой схемы. Если сигнал инкремента находится в состоянии 1 без управляющей линии записи, схема будет работать в бесконечном цикле, инкрементируя PC. С управляющей линией записи мы можем обеспечить синхронизацию изменений в PC с изменениями в других блоках. Этот же механизм применяется в нашей вычислительной схеме.

Соединения и временные характеристики

Поведение PC (а на самом деле любого цифрового устройства) полностью определяется последовательностью получаемых им управляющих сигналов и входных данных, то есть его связями с другими блоками. В нашей реализации PC четыре таких соединения.

- Значения входной шины задаются в соответствии с адресными битами в IR (в случае, если выполняется инструкция перехода).
- Управляющее значение инкремента или загрузки (ровно одно из двух) задается во время исполнительной фазы тактового генератора в зависимости от того, является текущая инструкция инструкцией перехода по нулю при R, равном 0 (загрузка), или нет (инкремент).
- На управляющую линию подается импульс записи в фазе исполнения от тактового генератора, по которому новое значение сохраняется в регистре PC.

С такими соединениями поведение схемы определяется сигналами цикла тактирования, поступающими от тактового генератора: либо увеличенное значение, либо адрес из инструкции перехода по нулю загружается в регистр PC в конце фазы исполнения. Содержимое PC всегда доступно на выходной шине (но используется только в фазе выборки для получения адреса следующей инструкции).

Подведем итог: наша реализация PC состоит из трех узлов, трех внутренних шинных соединений, входной и выходной шины и трех управляющих входов. Эта схема станет хорошей разминкой перед построением процессора, потому что она помогает понять сигналы выборки-исполнения и роль сигналов записи при управлении и синхронизации изменений состояния содержимого памяти. Если вы понимаете, как работает PC, вам будет намного проще понять, как работает процессор.

Как и с любой схемой, мы теперь можем работать с PC на более высоком уровне абстракции, используя интерфейс вместо реализации, и рассчитывать на то, что

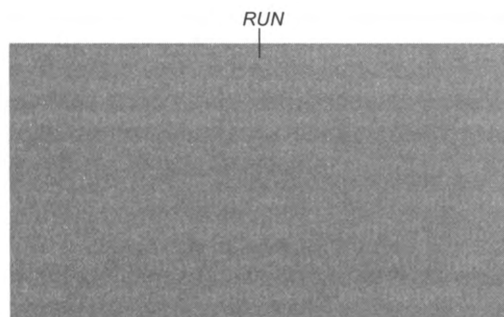
значения в PC будут изменяться под воздействием управляющих сигналов в соответствии с приведенным описанием.

Архитектура и соединения процессора TOY-8

Наконец, все готово для главной темы этой главы — проектирования процессора целиком. Мы будем точно следовать методологии, рассмотренной ранее для PC, но с большим количеством узлов и модулей, шинных соединений и управляющих линий. Интересно, что увеличение не столь значительно: PC содержит 2 узла, 1 шинный мультиплексор, 5 шинных соединений и 3 управляющие линии, а весь процессор содержит 7 модулей разного уровня сложности, 2 шинных мультиплексора, 10 шинных соединений и 16 управляющих линий[1].

Интерфейс

Наша схема процессора соединяется с внешним миром, а не с другой схемой. Например, кнопка RUN на передней панели TOY-8 запускает тактовый генератор. Остальные подробности соединений — с переключателями, кнопками и индикаторами на передней панели, а также подключение устройства ввода/вывода — мы опускаем. В контексте построения процессора можно считать, что память и PC содержат исходные значения (программа и ее начальный адрес), которые предоставляются внешним оборудованием. В реальном мире эта базовая функция прошла путь от ввода программистами двоичного кода с помощью переключателей до специализированного оборудования, инициализирующего всю память перед запуском тактового генератора. Кроме того, с точки зрения процессора совершенно не важно, является ли стандартное устройство ввода/вывода перфоратором для лент/карт или подключением к Интернету, так что рассматривать этот интерфейс во всех подробностях тоже не обязательно.



Интерфейс процессора

[1] Как следует из дальнейшего текста, под «процессором» здесь понимается вполне завершенная машина, включающая также и память. — *Примеч. науч. ред.*

Функциональные блоки процессора

Наша схема процессора TOY-8 состоит из семи блоков. Все пункты списка, кроме последнего, вам уже знакомы; они уже неоднократно упоминались в тексте с момента первого знакомства с машиной TOY.

- АЛУ.
- Регистр процессора (R).
- Регистр инструкции (IR).
- Счетчик адреса программы (PC).
- Основная память.
- Тактовый генератор (с сигналами записи).
- Комбинационная схема CONTROL для организации работы управляющих линий.

Полные реализации всех пунктов, кроме CONTROL, уже приводились ранее в этой главе. Проектирование и реализация схемы CONTROL являются главной темой этого раздела.

Шинные соединения

По большинству шин передается содержимое машинного слова, поэтому в TOY-8 они состоят из 8 проводников; некоторые передают адреса, на машине TOY-8 они состоят из 3 проводников. Разрядность шины входит в число главных факторов проектирования — на 64-разрядной машине шины от машинных слов должны состоять из 64 проводников. Даже в машине TOY-8 шины занимают значительное место на схеме.

Шинные соединения определяются потребностями набора инструкций. Например, для выполнения инструкции сохранения на машине TOY-8 адресные биты IR должны быть соединены с адресными линиями памяти, а регистр R должен быть соединен с входной шиной памяти. Аналогичный элементарный анализ дает все шинные соединения в машине TOY-8, перечисленные в таблице на с. 1041 вверху.

Как было показано на примере PC, две разные выходные шины не могут быть соединены с одной входной шиной, а значит, наш список соединений подразумевает необходимость в двух шинных мультиплексорах: «3-в-1» для переключения между входами R (АЛУ, биты поля адреса IR и память) и «2-в-1» для переключения между входами адресной входной шины (PC и биты поля адреса IR). В дальнейшем мы будем называть их «R мульт» и «MA мульт» соответственно. В остальном все шинные соединения идут непосредственно с выходной шины одного блока на входную шину другого.

инструкция	шинные соединения
выборка (все инструкции)	От PC к адресным линиям памяти От памяти к IR
останов	–
сложение, XOR, AND	От битов поля адреса IR к адресным линиям памяти От памяти к АЛУ 0 От R к АЛУ 1 От АЛУ к R
загрузка адреса	От битов поля адреса IR к R
загрузка	От битов поля адреса IR к адресным линиям памяти От памяти к R
сохранение	От битов поля адреса IR к адресным линиям памяти От R к памяти
переход по нулю	От битов поля адреса IR к PC

Шинные соединения в машине TOY-8

Функциональные блоки, шинные мультиплексоры и шинные соединения нашего процессора TOY-8 изображены на следующей странице (вместе с управляющими линиями, которые будут описаны ниже). Чтобы лучше понять таблицу шинных соединений, проанализируйте каждую из ее строк, сверяясь со схемой.

CLOCK запуск
CLOCK остановка

АЛУ сложение
АЛУ xor
АЛУ and

R мульт АЛУ
R мульт MEMORY
R мульт IR

R запись
IR запись
PC загрузка
PC инкремент
PC запись

MA мульт PC
MA мульт IR
MEMORY запись

Управляющие линии для TOY-8

Управляющие линии

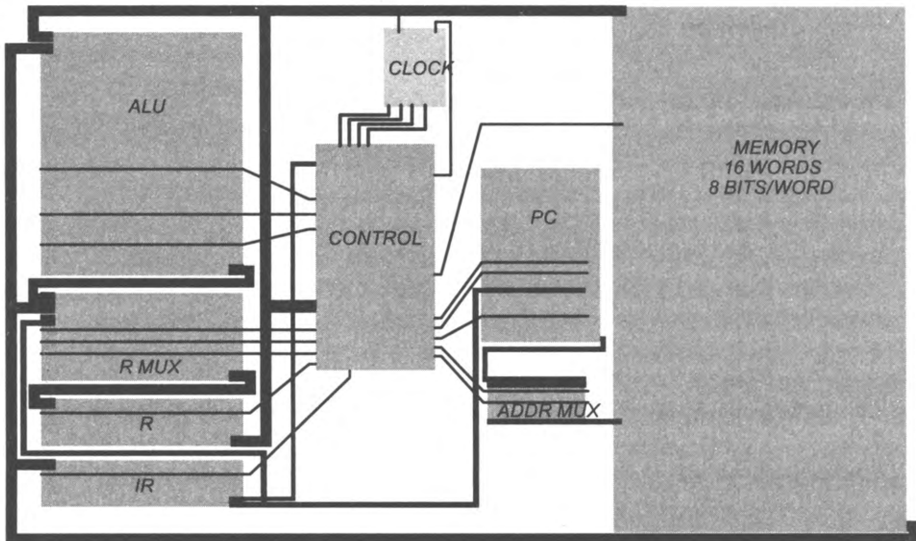
Теперь необходимо разобраться с линиями, управляющими работой функциональных блоков и мультиплексоров. Каждая управляющая линия является входной для некоторого блока и выбирает некоторое действие. Линиям присваиваются соответствующие названия (см. таблицу справа). Все управляющие линии работают под управлением сигналов тактового генератора, поэтому каждая линия в какой-то момент времени должна быть соединена с тактовым сигналом. Управлением этими линиями занимается схема CONTROL, которая будет представлена позднее в этом разделе. Но прежде чем рассматривать ее, мы переберем необходимые управляющие линии для каждого из четырех сигналов тактового генератора.

Выборка

Для начала рассмотрим фазу выборки процессора. Эта фаза проста, потому что управляющие сигналы одинаковы для всех инструкций. Цель фазы выборки — загрузка инструкции, адрес которой хранится в PC, в регистр IR. Это действие выполняется в два этапа.

- Сигнал выборки устанавливает управляющую линию *MA мульт PC* в состояние 1.
- Импульс записи в фазе выборки подается непосредственно на линию *IR запись*.

Так как шинный вход IR соединяется только с выходом памяти, в результате этих двух шагов адресуемое слово (следующая инструкция) будет загружено в IR.



Строение, шинные соединения и управляющие линии в процессоре машины TOY-8

Исполнение

В фазе исполнения последовательность управляющих сигналов зависит от текущей инструкции. Собственно, *последовательность управляющих сигналов и выполняет инструкцию*. Результатом большинства инструкций является запись в R нового значения, поэтому импульс записи в фазе исполнения управляет линией *MEMORY запись* для инструкции сохранения или *R запись* для инструкций арифметических операций, загрузки адреса и загрузки данных. В остальных случаях управляющие сигналы обеспечивают переключение шинных мультиплексоров и выбор соответствующей операции для АЛУ.

Кроме того, в фазе исполнения значение PC всегда изменяется, поэтому импульс записи в фазе исполнения подается на управляющую линию *PC запись*. Для всех инструкций, кроме успешной инструкции перехода по нулю, тактовый сигнал ис-

полнения управляет линией *PC инкремент*; для успешной инструкции перехода по нулю он управляет линией *PC загрузка*.

Чтобы лучше понять эту таблицу, стоит проверить по ней каждую инструкцию, задавая себе следующие вопросы: что эта инструкция должна делать? Какие шинные соединения участвуют в выполнении ее предназначения? Какая последовательность управляющих сигналов обеспечит это выполнение? Мы рассмотрим все инструкции по порядку.

Останов

Инструкция останова устанавливает управляющую линию *CLOCK остановка*, что приводит к остановке тактового генератора.

Арифметические операции

При наличии шинных подключений и установке управляющего сигнала *MA мульт IR* АЛУ всегда вычисляет сумму, поразрядную операцию XOR или поразрядную операцию AND слова памяти, адрес которого хранится в IR (первая входная шина)

Инструкция	*Выборка*
Все	адрес мульт PC
	Исполнение
Сложение	MA мульт IR АЛУ сложение R мульт АЛУ
Остановка	Остановка
XOR	MA мульт IR АЛУ xor R мульт (АЛУ)
AND	MA мульт IR АЛУ and R мульт (АЛУ)
Загрузка адреса	R мульт (IR)
Загрузка данных	адрес мульт (IR) R мульт (память)
Сохранение данных	адрес мульт (IR)
Все, кроме успешной инструкции перехода по нулю	PC инкремент
Успешная инструкция перехода по нулю	PC загрузка

Управляющие линии для инструкций TOY-8

и R (вторая входная шина), но в фазе исполнения на выходную шину АЛУ подается только один результат, для которого соответствующий управляющий сигнал находится в состоянии 1. Эта шина соединена с мультиплексором R, поэтому установка управляющей линии *R мульт АЛУ* в фазе исполнения приводит к тому, что значения на ее проводниках будут записаны в R по импульсу *R запись*.

Загрузка адреса

Установка управляющей линии *R мульт АЛУ* в фазе исполнения приводит к тому, что биты поля адреса IR сохраняются в R по сигналу записи в фазе исполнения на управляющей линии *R запись*.

Загрузка

Установка управляющей линии *R мульт IR* и управляющей линии *MA мульт IR* в фазе исполнения приводит к тому, что адресуемое слово памяти сохраняется в R по сигналу записи в фазе исполнения на линии *R запись* (так как выходная шина памяти соединена с мультиплексором R).

Запись

Установка управляющей линии *MA мульт IR* в фазе исполнения приводит к тому, что адресуемое слово памяти запишется в R по сигналу записи в фазе исполнения на линии *R запись* (так как выходная шина R соединена с входной шиной памяти).

Инкремент PC

Установка управляющей линии *PC инкремент* в фазе исполнения приводит к тому, что значение, вычисленное инкрементором, будет сохранено в PC по сигналу записи в фазе исполнения на линии *PC запись*. Это происходит всегда, кроме одного случая: текущей инструкцией является переход по нулю, и R содержит нуль.

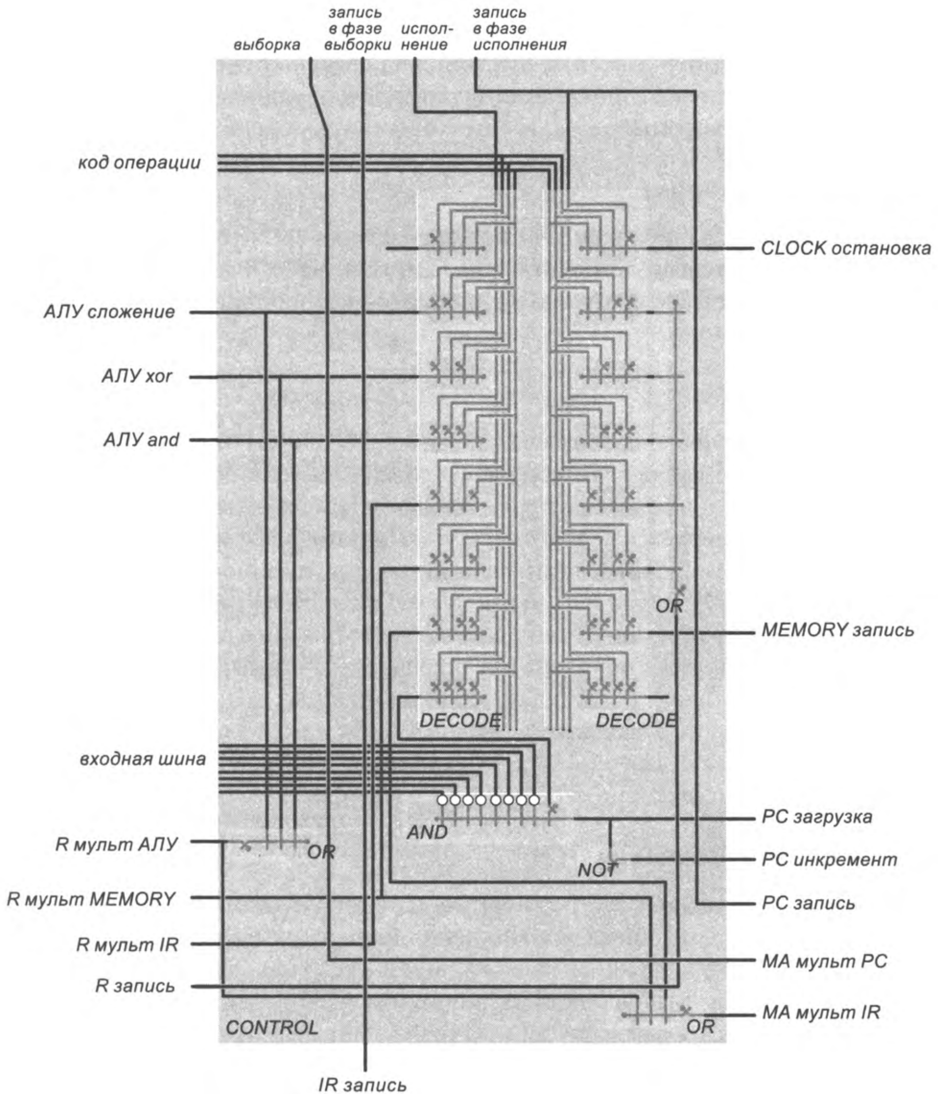
Загрузка PC

Установка управляющей линии *PC загрузка* в фазе исполнения приводит к тому, что значение на адресных проводниках IR будет сохранено в PC по сигналу записи в фазе исполнения на линии *PC запись*. Это происходит тогда, когда текущей инструкцией является переход по нулю и R содержит нуль.

Итак, каждая инструкция реализуется определенной последовательностью управляющих сигналов. Каждой последовательностью управляет тактовый генератор по указаниям еще одной комбинационной схемы CONTROL, которая будет рассмотрена ниже.

CONTROL

Полная реализация схемы устройства управления CONTROL, передающей тактовые сигналы на управляющие линии, приведена на следующей странице. Интересно, что схема состоит всего из двух демультиплексоров и пяти логических элементов. Затем мы подробно рассмотрим реакцию на последовательность такто-



Комбинационная схема для управления последовательностью действий на управляющих линиях процессора TOY-8

вых сигналов, представленную наверху, в бесконечном цикле, создаваемом схемой тактового генератора: выборка, запись в фазе выборки, исполнение, запись в фазе исполнения. Конечно, ключом к пониманию реакции является то, что он очень сильно зависит от текущего состояния машины, особенно значения битов кода операции в IR. Эта схема является небольшим исключением в наших соглашениях о входах/выходах блоков: входы располагаются сверху, но выходы располагаются вдоль всех трех других сторон.

Выборка

Реакция на сигнал выборки проста: эта линия напрямую соединяется с управляющей линией *MA мульт PC*, приказывая мультиплексору переключить адрес из PC на адресные входы памяти.

Запись в фазе выборке

Реакция на импульс записи в фазе выборки тоже проста: эта линия напрямую соединяется с управляющей линией *IR запись*. В сочетании с выборкой (см. выше) импульс приводит к тому, что слово памяти, адрес которого хранится в PC, загружается в IR.

Исполнение

Демультиплексор в левой средней части схемы CONTROL выполняет функции переключателя, который передает значение сигнала исполнения на одну из своих выходных линий в зависимости от кода операции. Таким образом, поведение схемы полностью определяется кодом операции. Например, для кода операции XOR демультиплексор активизирует третью линию сверху, что приводит к активизации управляющей линии XOR для АЛУ и выбору АЛУ для *R мульт*. Возможно, вы начнете лучше понимать эту схему, если проверите, что она активизирует правильные управляющие линии; если понадобится, обращайтесь к таблице из предыдущего раздела. Особый интерес представляет переход по нулю, который устанавливает управляющую линию загрузки для PC, если все биты R равны нулю. Единственный элемент NOT устанавливает линию *PC инкремент*, если инструкция не является переходом по нулю или какой-либо бит R не равен нулю.

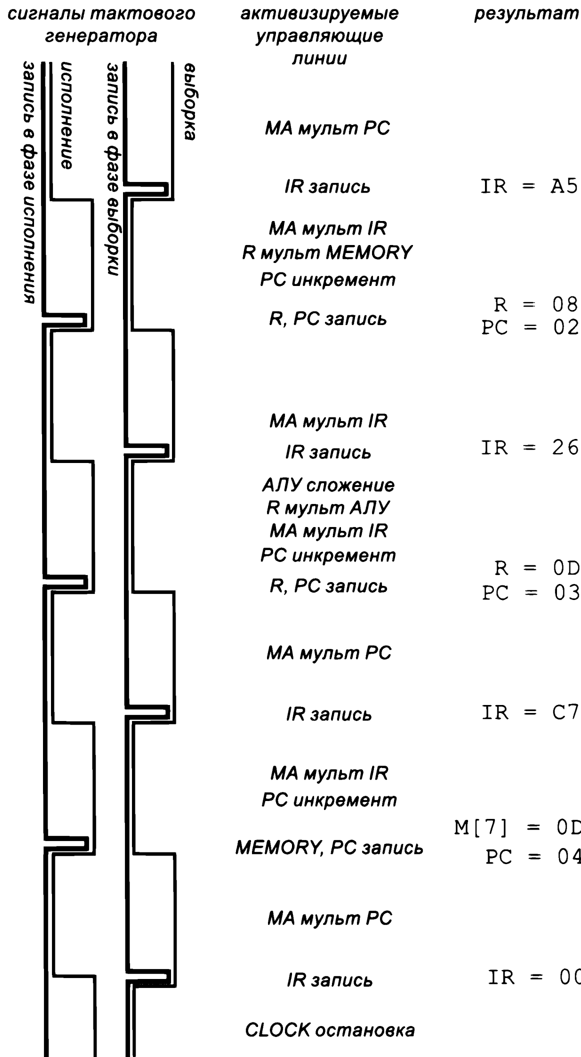
Запись в фазе исполнения

Демультиплексор в правой части схемы выполняет функции переключателя, который передает значение сигнала исполнения на одну из своих выходных линий в зависимости от кода операции. И снова поведение схемы полностью определяется кодом операции. Выходы демультиплексора для операций сложения, XOR, AND, загрузки адреса и загрузки данных поступают на логический элемент OR, который активизирует линию *R запись*, чтобы импульс приводил к сохранению соответствующего значения в R, а выход мультиплексора для инструкции сохранения напрямую активизирует линию *MEMORY запись*, так что импульс приводит к сохранению значения R в памяти.

Пример: программа TOY-8

В последнем примере мы рассмотрим полную последовательность управляющих сигналов для аналога нашей «первой TOY-программы» для машины TOY-8. Напомним, что эта программа складывает два числа:

```
1  A5  R = M[5]
2  26  R = R + M[6]
3  C7  M[7] = R
4  00  halt
5  08
6  05
7  00
```



Последовательность активации управляющей линии для программы TOY-8

Предполагается, что ячейки памяти с 1 по 7 загружены числами, а в PC хранится 01. Неважно, как была заполнена память: программистом с помощью переключате-

лей и кнопок, как описано в разделе 6.2, или специализированным оборудованием в вашем компьютере. Важно то, что конечный результат представляет собой не что иное, как исходное состояние памяти и PC. Нас сейчас интересует то, что происходит при запуске тактового генератора.

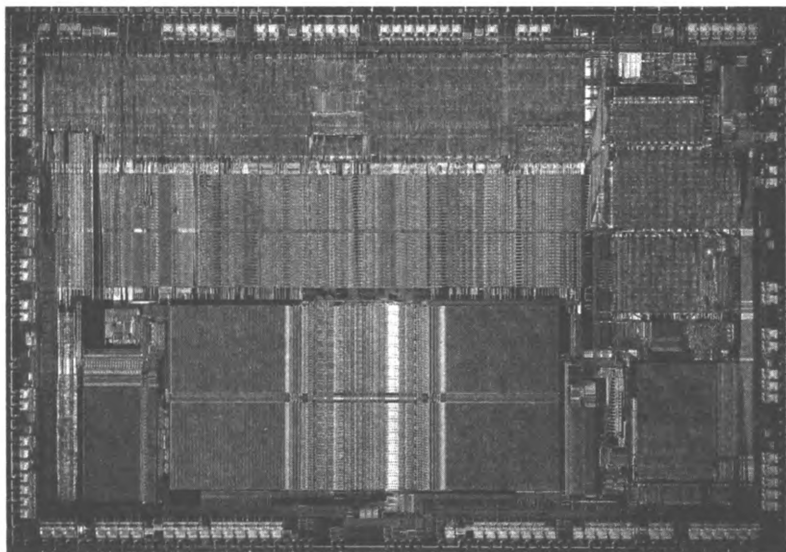
На диаграмме на с. 1048 показан цикл из четырех сигналов, вырабатываемых тактовым генератором, — выборка, запись в фазе выборки, исполнение и запись в фазе исполнения — бесконечно повторяющийся, пока работает генератор. В среднем столбце изображена последовательность управляющих линий, активизируемых сигналами генератора, а в правом столбце — последствия для состояния памяти, IR и PC. Тщательный анализ диаграммы подкрепит центральную идею этого раздела: любая программа TOY-8 реализуется последовательностью активизаций управляющих линий, обусловленных циклом сигналов тактирования. Примечательно, что простая схема CONTROL может реализовать таким образом полный набор инструкций TOY-8.

И теперь... вы знаете, так работает ваш компьютер. Маленькая схема преобразует периодические тактовые импульсы в бесконечно повторяющийся цикл сигналов тактирования, которые запускают последовательность управляющих сигналов, изменяющих состояние машины, в соответствии с ее текущим состоянием, прежде всего PC (адрес выполняемой инструкции в памяти) и IR (сама инструкция, особенно код операции). Так что если вас кто-нибудь спросит, вы сможете объяснить, как работает компьютер.

Перспективы

На с. 1050–1051 вы видите полную схему процессора для TOY-8 на уровне детализации, на котором показан каждый переключатель. Тем самым будет выполнена одна из основных целей этой книги — снять покров тайны с того, что происходит внутри вашего компьютера. И сейчас стоит задуматься над тем, что же из этого следует.

Как мы уже подчеркнули, главное различие между TOY-8 и вашим компьютером — масштаб, а львиная доля задач масштабирования решается достаточно просто. Например, мы можем легко масштабировать схему, чтобы получить 32-разрядный компьютер с 29-разрядными адресами и 2^{29} (более 500 миллионов) 32-разрядными словами памяти — правда, такая схема будет содержать более 16 миллиардов триггеров вместо 150 триггеров в TOY-8. В то же время удвоение числа инструкций в наборе может потребовать более сложной логики, но при этом увеличит схему всего вдвое, то есть относительно ненамного. Может, заявление слишком сильное, потому что АЛУ TOY-8 реализует всего три операции, тогда как значительная часть усилий по проектированию современных процессоров направлена на поддержку всевозможных операций, включая операции с плавающей точкой, и на управление памятью. Тем не менее мы лишь указываем на то, что эти затраты зависят от разрядности машинного слова линейно, тогда как затраты на реализацию памяти масштабируются по экспоненте.



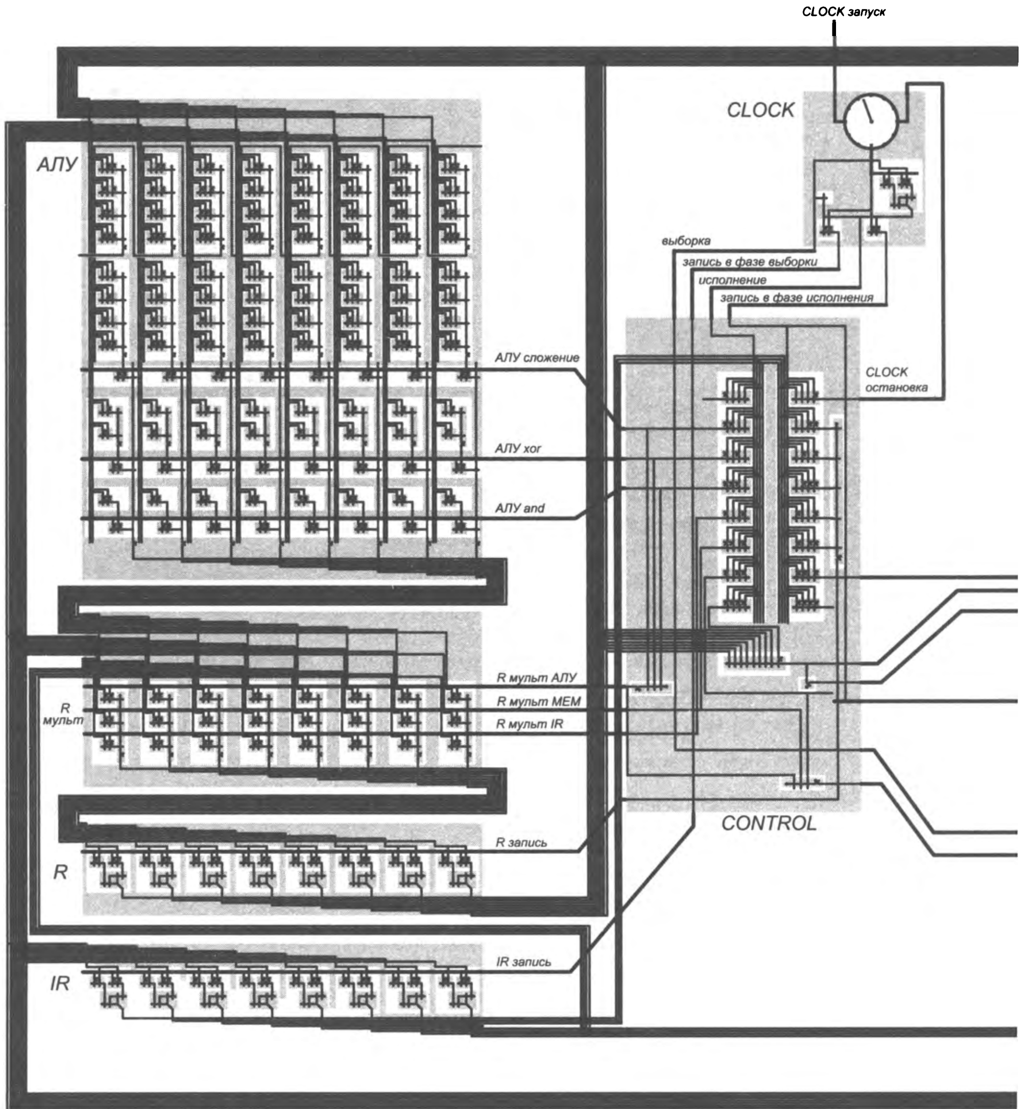
Микропроцессор Intel 80486 SX

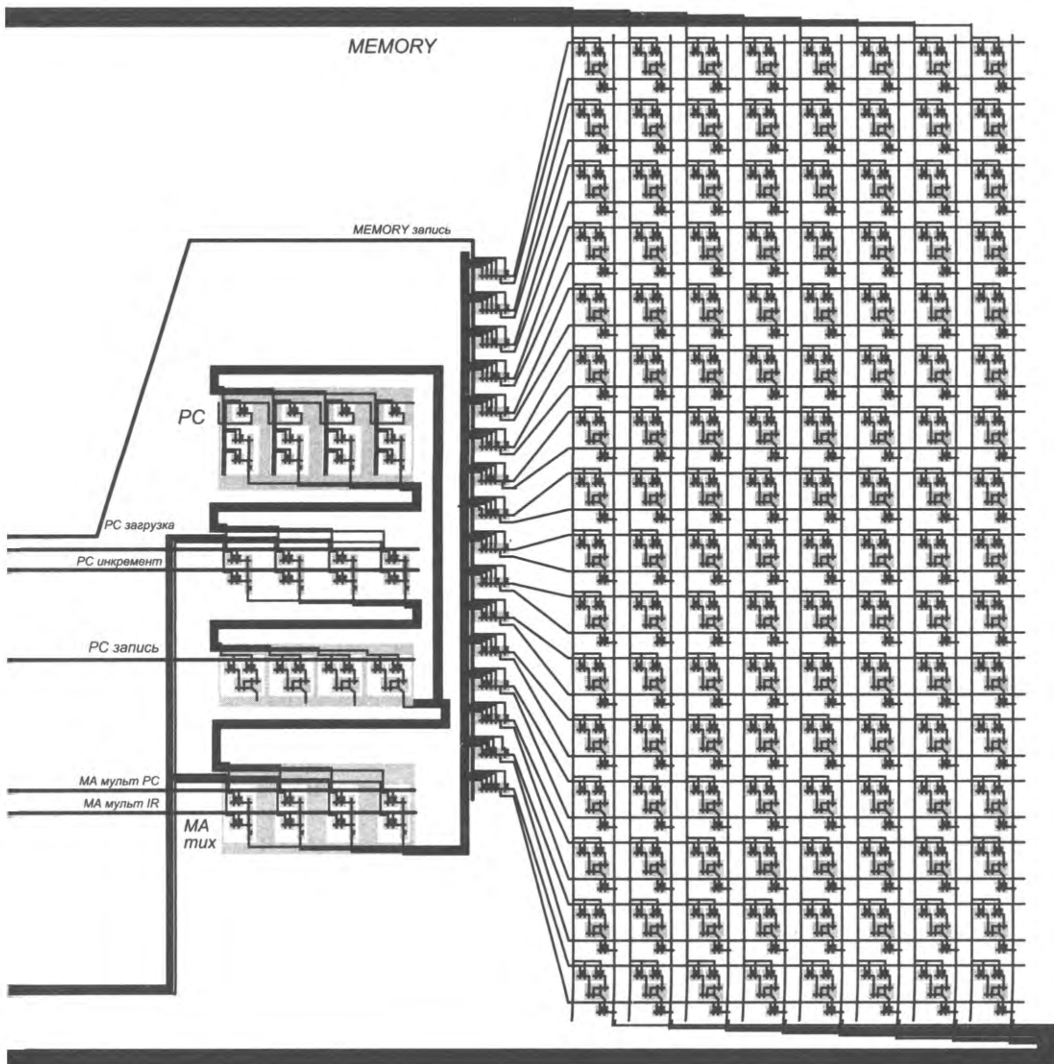
Как упоминалось ранее, и TOY-8, и ваш компьютер имеют фон-неймановскую архитектуру — в них нет принципиальных различий между инструкциями программ и словами данных. При виде схемы это становится предельно ясно. Можно также представить себе и построение схемы с двумя разными видами памяти: для программ и для данных. В самом деле, в наши дни такие схемы создаются по соображениям безопасности[1].

Геометрия и топология схем занимают центральное место в проектировании. Мы уже говорили о том, что между абстрактным миром цифровых схем и физическим миром материалов, выполняющих функции переключателей, существует прямая связь. В наши дни разработчики строят свои вычислительные схемы (с помощью компьютерных программ!), а затем передают их компьютеризированным системам, изготавливающим микросхемы.

Предупреждаем: в этой главе мы старались помочь вам понять, что происходит внутри компьютера, а не научить вас строить современные высокопроизводительные схемы. Все принимавшиеся решения были направлены на то, чтобы сделать схемы простыми и понятными. Например, повороты позволяют сэкономить довольно значительную полезную площадь. Кроме того, учтите, что никто на практике не пытается изображать на схеме каждый переключатель: разработчики реальных компьютеров работают на уровне логических элементов (на один уровень абстрак-

[1] Это так называемая *гарвардская* архитектура (см. примечание в предыдущей главе). Кроме безопасности, она может выигрывать в производительности и в некоторых технических аспектах реализации. — *Примеч. науч. ред.*


CLOCK запуск
CLOCK
АЛУ
выборка
запись в фазе выборки
исполнение
запись в фазе исполнения
АЛУ сложение
CLOCK
остановка
АЛУ xor
АЛУ and
R мульт АЛУ
R мульт MEM
R мульт IR
R
мульт
CONTROL
R запись
R
IR запись
IR


MEMORY
MEMORY запись
PC
PC загрузка
PC инкремент
PC запись
MA мульт PC
MA мульт IR
MA
mux

ции выше), а топологией схем занимаются, как правило, отдельно. Тем не менее мы хотим убедить вас, что вы можете спроектировать собственный компьютер, что вы можете написать программу, чтобы масштабировать его, и можете представить, как построить такой компьютер.

Если вы найдете микроскоп, вскроете процессор своего компьютера и изучите его с увеличением в 100 000 раз или около того (что вообще-то сделать не так просто), вы увидите, что он не так уж сильно отличается от наших абстрактных представлений. Конечно, в настоящем компьютере действуют разного рода физические ограничения, он имеет гораздо более сложную архитектуру и содержит больше различных типов блоков, чем мы рассматривали, поэтому для понимания всех подробностей необходимо что-то знать об архитектуре. Но даже в этом случае вы сможете опознать шины, регистры, память и другие компоненты, и вы будете видеть каждый переключатель. И наоборот, после некоторых уступок реальному миру та же технология может использоваться для изготовления физического процессора TOY-8 по схеме, приведенной выше.

Базовая идея о том, что все вычисления основаны на двух простых абстракциях (переключатель и тактовый генератор), обладает огромной мощью. Чтобы создать более быстрый компьютер с бóльшим объемом памяти и регистров, достаточно иметь более компактный и быстрый переключатель и более быстрый тактовый генератор. Простота использования незначительных технологических усовершенствований для построения улучшенных компьютеров по сравнению с построением новых схем «с нуля» объясняет, почему компьютеры развиваются без смены базовой архитектуры. При этом никто не отрицает, что усовершенствования архитектуры могут стать новым передовым рубежом компьютерных технологий.

Вопросы и ответы

В. Серьезно? И это все? Не могу поверить, что все настолько просто.

О. Добро пожаловать в клуб! Конечно, при разработке современных компьютерных микросхем приходится учитывать множество подробностей, в которых воплощены результаты непрерывной поэтапной разработки и совершенствования на протяжении нескольких десятилетий. Одно дело — нарисовать тысячи отрезков и точек на печатной странице, и совсем другое — упаковать миллиарды физических элементов на нескольких квадратных сантиметрах и заставить их срабатывать миллиарды раз в секунду.

Конечно, мы позволили себе некоторые художественные вольности. В нашей схеме присутствует множество аспектов, к которым еще можно придраться, прежде чем переходить к разработке реального устройства; перечислить их все попросту невозможно. Однако компоненты, рассмотренные нами, просты и функциональны, они выдержали проверку временем на бесчисленных технологиях и физических реализациях, и они находят множество возможных применений.

Впрочем, это всего лишь отправная точка. Интересно то, что она не так уж сильно удалена от отправной точки первых компьютеров, построенных в середине XX века. Если отвлечься от подробностей, многие фундаментальные идеи, лежащие в основе любого современного процессора, безусловно, представлены и в машине TOY-8.

Упражнения

7.5.1. Напишите программу для TOY-8, которая вычитает одно 8-разрядное число в дополнительном коде из другого.

7.5.2. Напишите программу для TOY-8, которая выводит в стандартное устройство вывода числа Фибоначчи, меньшие 256.

7.5.3. Напишите программу для TOY-8, которая перемножает два 8-разрядных целых числа без знака. При необходимости считайте, что доступны 32 слова памяти. *Примечание*: чтобы избежать путаницы, считайте, что память разбита на два банка M0[] и M1[], так что инструкция `36` означает «прибавить к R содержимое M1[6]».

7.5.4. Опишите, как обеспечить, чтобы в ячейке памяти `0` всегда хранился 0.

7.5.5. Опишите, какие шинные соединения потребуются для реализации инструкции передачи управления с сохранением точки возврата (branch and link) для TOY-8.

7.5.6. Сколько переключателей содержит схема процессора TOY-8? Постройте таблицу с информацией об их количестве в каждом узле (с процентами от общего количества) и в устройстве в целом.

7.5.7. Приведите последовательность управляющих линий, активизируемых для программы TOY-8, представленной в начале этого раздела (программа для вычисления суммы чисел в стандартном вводе).

Упражнения повышенной сложности

7.5.8. *PC на базе счетчика.* Постройте 4-разрядный регистр адреса программы PC на базе счетчика (см. упражнение 7.3.15) вместо отдельных регистра и инкрементора.

7.5.9. *Построение схемы компьютера.* Расширьте свои решения упражнений 7.2.15–7.2.18 и упражнений 7.3.8–7.3.11 так, чтобы они строили полную схему компьютера (по аналогии со схемой TOY-8 в конце этой главы).

Заключение

В завершение мы кратко подведем итоги того, что вы узнали о программировании и об информатике, а затем опишем ряд аспектов мира вычислительных систем, с которыми вы можете столкнуться на следующем этапе своей работы. Надеемся, эта информация подтолкнет вас к практическому применению знаний, полученных из книги, чтобы вы смогли больше узнать о роли компьютеров и информационных технологий в окружающем вас мире.

После чтения глав 1–4 вы научились программировать. Если вы уже умеете водить машину в городе, научиться управлять внедорожником несложно; точно так же вы без особых затруднений освоите другой язык программирования. Многие специалисты регулярно работают на разных языках для разных целей. Примитивные типы данных, ветвления, циклы, массивы и подпрограммы из глав 1 и 2 (хорошо послужившие программистам первые лет двадцать компьютерной эпохи), объектно-ориентированное программирование из главы 3 (используемое современными программистами) — базовые модели, встречающиеся во многих языках программирования. Ваше умение пользоваться ими, а также фундаментальные типы данных из главы 4 подготовят вас к работе с библиотеками, средами разработки и специализированными приложениями самых разных видов. Также ваша подготовка позволит вам оценить мощь абстракции при проектировании сложных систем и понять, как они работают.

Как вы узнали из глав 5–7, изучение информатики — это намного больше, чем просто изучение программирования. Ваше умение программировать и способность быть с компьютером на «ты» позволят вам соприкоснуться с рядом выдающихся интеллектуальных достижений прошлого века, некоторыми важнейшими нерешенными задачами нашего времени и их ролью в эволюции окружающей нас вычислительной инфраструктуры. Пожалуй, еще важнее другое: как мы неоднократно указывали в книге, вычислительные технологии играют все более важную роль в нашем понимании природы, от геномики и молекулярной динамики до астрофизики. Поэтому дальнейшее изучение базовых принципов компьютерных технологий наверняка принесет свои плоды.

Библиотеки Java

Система Java предоставляет в распоряжение программиста многочисленные ресурсы. Мы часто используем в книге некоторые библиотеки Java (такие, как `Math` и `String`), но не затрагиваем другие. Одна из уникальных особенностей Java заключается в том, что в Интернете можно найти много информации о библиотеках.

Если вы еще не ознакомились с библиотеками Java, самое время заняться этим. Большинство библиотек предназначено для профессиональных разработчиков, но есть немало других, которые будут полезны и для вас. Изучая новую библиотеку, думайте не о том, что ее *нужно* использовать, а о том, что ее *можно* использовать. А когда вы найдете API, который покажется вам удобным, — пользуйтесь!

Среды программирования

Несомненно, в будущем вы столкнетесь и с другими средами программирования, кроме Java. Многие программисты (и даже опытные профессионалы) оказываются зажаты между прошлым (из-за гигантских объемов унаследованного кода на старых языках, таких как C, C++ и Fortran) и будущим (из-за появления таких современных инструментов, как Ruby, Python и Scala). Если вы захотите изучить Python, возможно, вам понравится наша книга *An Introduction to Programming in Python* — близкий родственник этой книги. Как и прежде, самое важное, о чем следует помнить при работе с любым языком программирования, — что пользоваться им *не обязательно*. Если другой язык в большей степени отвечает вашим потребностям, ничто не мешает вам пользоваться им. Люди, которые упорно держатся в границах одной среды программирования (независимо от причины), упускают хорошие возможности.

Научные вычисления

Обработка числовых данных нередко требует сложных и хитроумных решений (из-за точности представления), поэтому использование библиотек математических функций безусловно оправданно. Многие научные работники используют Fortran, старый язык научных расчетов; другие работают в среде Matlab на языке, созданном специально для вычислений с применением матриц. Благодаря сочетанию хороших библиотек и встроенной поддержки матричных операций Matlab становится привлекательным решением для многих задач. Но в Matlab нет поддержки изменяемых типов и других современных средств, поэтому для многих других задач Java подходит лучше. Но вы можете использовать сразу два языка! Ряд математических библиотек, используемых программистами Matlab и Fortran, доступны также и из Java (и из современных интерпретируемых языков.)

Распределенные и облачные вычисления

Сегодня значительная часть наших контактов с вычислениями связана с построением и использованием программ, запускаемых в браузере или на мобильном устройстве — возможно, на виртуальном компьютере в облаке. Ситуация в высшей степени примечательная, потому что она кардинально расширила круг людей, жизнь которых благодаря компьютерным технологиям изменилась в лучшую сторону. Если вы принадлежит к их числу, то, скорее всего, вы отметите эффективность основных методов, описанных в книге. Вы можете писать программы для обработки данных, хранящихся в других местах; программы, взаимодействующие с программами, работающими в других местах; и пользоваться многими другими

возможностями обширной и непрерывно развивающейся вычислительной инфраструктуры. В частности, наш научный подход к анализу быстродействия подготовит вас к организации вычислений в гигантском масштабе.

Прогрессивные компьютерные системы

Свойства конкретных компьютерных систем когда-то полностью определяли природу и масштаб решаемых ими задач, но в наши дни ситуация изменилась. Конечно, вы по-прежнему можете рассчитывать на то, что через год в вашем распоряжении появится более быстрый компьютер с большим объемом памяти. Стремитесь к тому, чтобы ваш код не зависел от машины, — но также приготовьтесь изучать и исследовать новые технологии и архитектуры, от графических процессоров до вычислительных систем с массовым параллелизмом и компьютерных сетей.

Теория вычислений

В отличие от тех возможностей, которые дает рост вычислительной мощности, некоторые фундаментальные ограничения на реализуемость вычислений были очевидны с самого начала. Они до сих пор играют важную роль в определении задач, которые вообще могут решаться на компьютерах. Как вы теперь знаете, существуют задачи, которые не могут быть решены ни одной компьютерной программой, а многие другие задачи, часто встречающиеся на практике, остаются слишком сложными (трудоемкими) для решения на любом современном компьютере. Каждый специалист, у которого процесс решения задач, творческой работы и исследований зависит от вычислений, должен учитывать эти факты.

Машинное обучение

Область искусственного интеллекта давно привлекала внимание специалистов по компьютерным технологиям. Огромные масштабы современных вычислительных технологий означают, что мечты ранних исследователей воплотились в жизнь до такой степени, что мы начинаем зависеть от способности компьютера извлекать информацию из его окружения: учиться водить автомобиль в автономном режиме, привести нас к продуктам, которые мы захотим купить, или предоставить нам ту информацию, которая нас интересует. Применение вычислительных технологий на этом уровне – задача безусловно более фундаментальная, чем изучение очередного набора API, но вам наверняка захочется заняться ею в будущем.

С того момента, когда вы для пробы создали, откомпилировали и запустили программу `HelloWorld`, вы прошли очень длинный путь. Тем не менее вам еще очень многое предстоит узнать. Программируйте, изучайте новые среды программирования, среды для научных вычислений, сетевые и облачные технологии, архитектуры компьютерных систем, теорию вычислений, теорию машинного обучения — и перед вами будут открываться такие возможности, о которых даже не подозревают люди, которые программировать не умеют.

Глоссарий

.class, файл — файл с расширением .class; содержит байт-кодJava, который выполняется виртуальной машиной Java.

.java, файл — файл, содержащий программу, написанную на языке программирования Java.

API (Application Programming Interface, программный интерфейс) типа данных — определение множества операций с этим типом, которые доступны для клиентского программного кода.

ASCII (American Standard Code for Information Interchange) — распространенный стандарт представления символов текста на английском языке, включенный в Юникод.

grep (generalized regular expression pattern match) — классическая утилита для поиска по шаблону с применением регулярных выражений.

IEEE 754 — международный стандарт представления данных и вычислений в формате с плавающей точкой, поддерживаемый современным компьютерным оборудованием (см. *плавающая точка*).

IR (регистр инструкции) — функциональный блок процессора, хранящий выполняемую инструкцию.

Java — объектно-ориентированный язык программирования общего назначения.

JIT-компилятор — компилятор, который непрерывно переводит программу на высокоуровневом языке на низкоуровневый язык во время выполнения программы. JIT-компилятор Java переводит байт-код Java на машинный язык.

JVM (виртуальная машина Java) — программа, выполняющая байт-код Java на микропроцессоре с использованием как интерпретатора, так и JIT-компилятора.

NP — множество всех задач поиска (например, задачи выполнимости логических уравнений, разложения на простые множители, и все задачи из **P**).

NP-полнота — характеристика самых «сложных» задач из **NP** (таких, как задача выполнимости логических уравнений, задача о вершинном покрытии и задачи выполнимости целочисленных линейных неравенств).

null — специальный литерал, представляющий отсутствие ссылки на какой бы то ни было объект.

P ≠ NP — известная недоказанная гипотеза о том, что некоторые задачи поиска не могут быть решены за полиномиальное время.

P — множество всех задач поиска, для которых существуют алгоритмы с полиномиальным временем (например, задачи сортировки, кратчайшего пути и выполнимости линейных неравенств).

PC (счетчик адреса программы) — функциональный блок в процессоре, хранящий адрес следующей выполняемой инструкции.

PDP-8 — реальный компьютер, использовавшийся в 1970-е годы.

this — ключевое слово, используемое в методах экземпляров или конструкторах для обозначения текущего объекта, для которого вызывается метод.

TOY — воображаемый компьютер, спроектированный для этой книги (похожий на PDP-8).

абстрактная машина — математическая модель процесса выполнения программы.

автогенератор — неустойчивая (самовозбуждающаяся) схема с обратной связью.

алгоритм с полиномиальным временем — алгоритм, который гарантированно выполняется за время, ограниченное сверху некоторой полиномиальной функцией от размера входных данных.

алгоритм с экспоненциальным временем — алгоритм, время выполнения которого ограничено снизу экспоненциальной (показательной) функцией от размера входных данных.

алгоритм — пошаговое описание решения задачи (например, алгоритм Евклида, сортировка слиянием или произвольная машина Тьюринга).

АЛУ (арифметико-логическое устройство) — устройство в компьютере, непосредственно выполняющее вычисления.

алфавит — конечный набор символов (например, двоичный алфавит `{a, b}`).

аргумент командной строки — значение, передаваемое программе (команде) из командной строки.

аргумент — выражение, результат которого программа Java вычисляет и передает по значению в метод.

архитектура фон Неймана — архитектура вычислительных устройств, в которой программы и данные хранятся в общей памяти.

ассоциативность операторов — правила, определяющие порядок применения операторов с одинаковым приоритетом (например, `1-2-3`).

байт-код Java — низкоуровневый машинно-независимый язык, используемый виртуальной машиной Java.

библиотека с сайта книги — библиотека, разработанная авторами для использования в рамках книги (`StdIn`, `StdOut`, `StdDraw` и `StdAudio`).

библиотека — файл .java, организованный так, что его функциональность может повторно использоваться другими программами Java.

бит — двоичная цифра (0 или 1).

блок управления (в процессоре), CONTROL — комбинационная схема в составе процессора, вырабатывающая сигналы управления для остальных схем.

булева алгебра — формальная система для работы с логическими выражениями.

булева логика — научная дисциплина, изучающая логические функции.

ветвление, условный переход — конструкция, обеспечивающая выполнение разных вычислений в зависимости от значения одного или нескольких логических выражений (например, `if`, `if-else` или `switch`).

виртуальная машина — определение или реализация компьютера в виде программы для другого компьютера.

возвращаемое значение — значение, передаваемое в вызывающий код как результат выполнения метода.

встроенный тип — тип данных, встроенный в язык Java (например, `int`, `double`, `boolean`, `char` или `String`).

выборка-инкремент-исполнение, цикл — процесс, на котором основана работа компьютера.

выдача исключения — генерирование ошибки компиляции или времени выполнения.

вызов метода — выражение, которое выполняет метод и возвращает полученное значение.

выполнимость — класс задач, в которых требуется определить, существует ли набор значений переменных, с которыми все заданные равенства (или неравенства) будут истинными.

выражение — комбинация литералов, переменных, операторов и вызовов методов, которая вычисляется программой Java для получения результата.

вычисление выражения — процесс преобразования выражения в значение (результат) применением операторов к операндам выражения. Порядок применения операторов к операндам определяется приоритетом операторов, их ассоциативностью и модификаторами порядка вычисления.

вычислимость — теоретическая возможность решения задачи на компьютере. Некоторые задачи (например, проблема остановки) не являются вычислимыми.

вычислительная сложность — невозможность решения задачи на компьютере за полиномиальное время.

глобальная переменная — переменная, областью видимости которой является вся программа или файл. См. также *статическая переменная*.

демультиплексор — комбинационная схема для передачи значений с входов на выбранные выходы.

детерминированность — свойство вычисления, каждый шаг которого полностью определяется его текущим состоянием.

детерминированный конечный автомат (ДКА) — детерминированная абстрактная машина, которая распознает регулярный язык.

дешифратор — комбинационная схема для выбора одного из своих выходов как активного.

дизъюнктивная нормальная форма (ДНФ) — стандартное алгебраическое представление логических функций в виде логической суммы произведений.

директива импорта — конструкция языка Java, позволяющая обращаться к коду из другого пакета без использования полного имени.

дополнительный код — способ машинного представления отрицательных чисел.

задача поиска — задача, для которой существует алгоритм с полиномиальным временем для проверки правильности предлагаемого решения.

закон Мура — эмпирическое наблюдение, сформулированное Гордоном Муром: с момента появления интегральных схем в 1960-е годы производительность процессоров и емкость памяти удваиваются каждые два года.

закрытый (метод) — часть кода реализации типа данных, не предназначенная для использования клиентами.

идентификатор — имя, используемое для обозначения переменной, метода, класса или другой сущности.

изменяемый объект — объект типа данных, значения которого могут изменяться.

изменяемый тип данных — тип данных, значения экземпляров которого могут изменяться (например, `Counter`, `Picture` или массивы).

инициализация переменной — первое присваивание значения переменной во время выполнения программы.

интерпретатор — программа для построчного выполнения программы, написанной на высокоуровневом языке программирования. Виртуальная машина Java интерпретирует байт-код Java и выполняет его на реальном компьютере.

интерфейс — контракт на реализацию классом некоторого набора методов.

исключение — аномальное состояние или ошибка во время выполнения программы.

исходный код — программа или фрагмент программы на языке высокого уровня (например, на Java).

итератор — тип данных, реализующий интерфейс `Iterator`. Используется для реализации типов данных с поддержкой перебора (`Iterable`).

класс — конструкция языка Java для реализации типа данных, определенного пользователем. Класс предоставляет «шаблон» для создания объектов, содержащих значения этого типа, и выполнения операций над ними в соответствии с API.

клиент — программа, работающая с реализацией типа данных по правилам его API.

командная строка — интерфейс для выполнения системных команд и запуска программ в терминальном окне: команда и ее аргументы вводятся в виде текстовой строки.

комбинационная схема — схема, не содержащая обратных связей (схема без памяти).

комментарий — пояснительный текст, который помогает читателю понять назначение кода. При компиляции (выполнении) программы комментарии игнорируются.

компилятор — программа, переводящая программу с языка высокого уровня на язык низкого уровня. Компилятор Java преобразует файл .java (содержащий исходный текст на языке Java) в файл .class (содержащий байт-код Java).

константа — переменная, значение которой известно во время компиляции и не изменяется в ходе выполнения программы (или между запусками программы).

конструктор — специальный метод типа данных, предназначенный для создания и инициализации нового объекта.

литерал — представление в исходном тексте программы значения типа данных для встроенных типов (например, `123`, `"Hello"` или `true`).

логическая функция — функция, отображающая логические значения в другое логическое значение.

логический элемент (вентиль) — небольшая комбинационная схема, реализующая логическую функцию (например, элементы AND, OR или NOT).

логическое выражение — выражение, результатом которого является значение типа `boolean`.

логическое значение — 0 или 1; `true` или `false`.

локальная переменная — переменная, определяемая внутри метода; область видимости такой переменной ограничивается этим методом.

лямбда-выражение — анонимная функция, которую можно передать для последующего выполнения.

лямбда-исчисление — универсальная модель вычислений, разработанная Алонзо Чёрчем.

магистраль данных — группа проводников (шина), связывающих один модуль с другим (также называется «шинным соединением»).

маскирование — выделение группы битов (разрядов) в машинном слове.

массив с изменяемым размером — структура данных, поддерживающая определенный процент заполнения ее элементами. Резервируемый для такого массива объем памяти меняется в зависимости от количества реально находящихся в нем элементов.

массив — структура данных, содержащая последовательность значений одного типа, с поддержкой создания, чтения и записи по индексу, а также перебора элементов.

машина Тьюринга — универсальная модель абстрактной вычислительной машины, предложенная Аланом Тьюрингом.

машинная инструкция — операция, реализованная на аппаратном уровне (например, сложение, загрузка или переход по нулю в машине TOY).

машинное слово — последовательность битов (разрядов) фиксированной длины, рассматриваемая как единое целое в архитектуре компьютера.

метод экземпляра — реализация операции с экземпляром типа данных (метод, вызываемый для конкретного объекта).

метод — именованная последовательность строк кода, которая может вызываться из другого кода для выполнения вычислений.

модуль (аппаратный), блок — компонент компьютера, обычно имеющий входные и выходные шины.

модуль (программный) — независимая программа, реализующая API (например, класс Java).

модульное программирование — стиль программирования, ориентированный на создание раздельных независимых модулей для решения конкретных задач.

модульное тестирование — практика включения в каждый модуль кода тестирования этого модуля.

мультиплексор — комбинационная схема для передачи значений с выбранных входов на выходы.

недетерминированность — свойство вычисления, в котором из одного или нескольких состояний возможно одновременно более одного перехода к следующим состояниям.

недетерминированный конечный автомат (НКА) — недетерминированная абстрактная машина, которая распознает регулярный язык.

неизменяемый объект — объект типа данных, значения которого не могут изменяться.

неизменяемый тип данных — тип данных, значения экземпляров которого не могут изменяться (например, `Integer`, `String` или `Complex`).

обертка — ссылочный тип, соответствующий одному из примитивных типов (например, `Integer`, `Double`, `Boolean` или `Character`).

область видимости — часть программы, где для обращения к переменной достаточно только ее имени.

обобщенный класс — класс, параметризованный по одному или нескольким параметрам-типам (например, `Queue`, `Stack`, `ST` или `SET`).

объект — представление значения типа данных в памяти компьютера; характеризуется состоянием (значение типа данных), поведением (операциями типа данных) и содержимым (местонахождением в памяти).

объектно-ориентированное программирование — стиль программирования, ориентированный на моделирование сущностей реального мира или абстрактных сущностей с использованием типов данных и объектов.

объявление переменной — указание имени и типа переменной.

операнд — значение, с которым работает оператор.

оператор — специальный символ (или последовательность символов), представляющий операцию со встроенным типом данных, например `+`, `-`, `*` и `[]`.

операционная система — программа, которая управляет ресурсами компьютера и предоставляет общие сервисные функции другим программам и приложениям.

операция — действие, которое может быть выполнено в Java (например, присваивание, условный переход `if`, цикл `while` или возврат `return`).

ошибка времени выполнения — ошибка, происходящая в процессе выполнения программы.

ошибка компиляции — синтаксическая ошибка, которая выявляется компилятором.

пакет — набор взаимосвязанных классов и интерфейсов, объединенных в общем пространстве имен. Пакет `java.lang` содержит фундаментальные классы и интерфейсы и импортируется автоматически; пакет `java.util` содержит Java Collections Framework.

память — компонент вычислительной машины, предназначенный для хранения данных и программ.

параметр — переменная, указанная в определении метода. Параметр инициализируется соответствующим аргументом при вызове метода.

параметр-тип — условное обозначение некоторого конкретного типа в обобщенном классе, которое клиент при вызове заменяет конкретным типом.

перегрузка метода — определение двух и более методов с одинаковыми именами (но разными списками параметров).

перегрузка оператора — определение поведения оператора, такого как `+`, `*`, `<=` или `[]`, для типа данных. В языке Java перегрузка операторов не поддерживается.

передача по значению — в Java используются два способа передачи аргументов методам: либо непосредственно в виде значения данных (для примитивных типов), либо в виде ссылки на объект в памяти (для ссылочных типов).

переменная класса — см. *статическая переменная*.

переменная экземпляра — переменная, определяемая в классе (но вне какого-либо метода) и представляющая значение некоторого типа данных, — то есть данные, связываемые с каждым экземпляром класса.

переменная — сущность для хранения значения. Каждая переменная языка Java обладает именем, типом и областью видимости.

переопределение метода — переопределение унаследованного метода (такого, как `equals()` или `hashCode()`).

переполнение — ситуация, при которой результат вычисления арифметической операции превышает максимально возможное значение.

перфолента — простейший носитель информации на заре компьютерной эпохи.

плавающая точка — общее описание формата «научной записи» для представления вещественных чисел на компьютере (см. IEEE 754).

побочный эффект — изменение внутреннего состояния программы: вывод данных, ввод данных, выдача исключения, изменение значения некоторого объекта (переменной экземпляра, параметра или глобальной переменной).

полиморфизм — использование одного API (или части API) для разных типов данных.

порядок вычисления — порядок обработки подвыражений в ходе вычислений (например, `f1()+f2()* f5(f3(),f4())`. В Java подвыражения обрабатываются слева направо независимо от ассоциативности или приоритета операторов. Аргументы методов в Java обрабатываются слева направо до вызова метода.

последовательностная схема — схема, содержащая обратную связь (схема с памятью).

примитивный тип данных — один из восьми типов данных, определяемых непосредственно в языке Java; к числу таких типов относятся `boolean`, `char`, `double` и `int`. В переменной примитивного типа хранится само значение данных (а не ссылка на содержимое объекта).

приоритет операторов — правила, определяющие порядок применения операторов в выражениях (например, `1+2*3`).

присваивание — операция Java, состоящая из имени переменной, за которым следует знак равенства (`=`) и выражение. Выполняя присваивание, Java вычисляет результат выражения и присваивает полученное значение переменной.

проблема остановки — задача, неразрешимость которой была доказана Тьюрингом.

проводник — элемент схемы, передающий логическое значение.

программа на машинном языке — последовательность машинных инструкций, выполняемая компьютером.

программа — последовательность операций (инструкций), выполняемых на компьютере.

разбор — преобразование строкового представления во внутреннее.

реализация — программа, реализующая набор методов, определяемых в API, для использования их клиентом.

регистр — узел в процессоре для хранения обрабатываемого слова.

регулярное выражение — выражение, использующее объединение, конкатенацию, замыкание и круглые скобки для определения регулярного языка.

регулярный язык — язык, который может распознаваться ДКА или определяться регулярным выражением.

сведение к полиномиальному времени — использование подпрограммы (с полиномиальным временем выполнения), предназначенной для решения одной задачи, для эффективного решения другой задачи.

сведение — использование подпрограммы, предназначенной для решения одной задачи, для решения другой задачи.

связный список — структура данных, состоящая из последовательности узлов, каждый из которых содержит ссылку на следующий узел последовательности.

совмещение имен — наличие двух (и более) ссылок на один и тот же объект.

ссылка на объект — механизм обращения к объекту путем указания на его содержимое в памяти (как правило, адрес памяти, где хранится объект).

ссылочный тип — тип, соответствующий классу или интерфейсу, или массив (например, `String`, `Charge`, `Comparable` или `int[]`). В переменной ссылочного типа хранится ссылка на объект, а не само значение типа данных.

стандартный ввод, вывод, графика и аудио — наши модули ввода/вывода для Java.

статическая переменная — переменная, связанная с классом в целом (а не с конкретным объектом).

статический метод — реализация функции в классе Java, без связи с конкретным объектом (например, `Math.abs()`, `Euclid.gcd()`, `StdIn.readInt()`).

строка — конечная последовательность алфавитных символов.

структура данных — способ организации данных в компьютере (обычно для экономии времени или места): массив, массив с изменяемым размером, связный список, бинарное дерево поиска и т. д.

сумматор — компонент вычислительного устройства, реализующий сложение (вычисление суммы) двух чисел.

схема — множество взаимосвязанных проводников, подключений к источнику питания и переключателей.

тактовый генератор — последовательностная схема, обеспечивающая синхронизацию управляющих линий выборки и исполнения в процессоре.

тезис Чёрча — Тьюринга — гипотеза о том, что машина Тьюринга может выполнить любое вычисление (принять решение о принадлежности к языку или вычислить функцию), которое может быть выполнено любым физически реализуемым вычислительным устройством.

теорема Клини — гипотеза о том, что регулярные выражения, детерминированные конечные автоматы (ДКА) и недетерминированные конечные автоматы (НКА) определяют регулярные языки.

терминальное окно — приложение операционной системы для ввода команд.

тип данных — множество значений и множество операций, определенных для этих значений.

типы данных Comparable (типы, поддерживающие сравнение) — в Java типы данных, реализующие интерфейс `Comparable`, который обеспечивает возможность их сравнения и упорядочивания

типы данных Iterable (итерируемые типы) — в Java типы данных, реализующие интерфейс `Iterable`, который обеспечивает возможность перебора их содержимого в циклах *foreach*, например `Stack`, `Queue` или `Set`.

трассировка — пошаговое протоколирование работы программы.

триггер — последовательностная схема, реализующая запоминающую ячейку на один бит.

уборка мусора — процесс автоматического выявления и освобождения неиспользуемой памяти.

универсальная машина Тьюринга — машина Тьюринга, которая может моделировать произвольную машину Тьюринга с произвольным вводом.

универсальность — гипотеза о том, что все достаточно мощные вычислительные устройства могут разрешать единое множество формальных языков и вычислять единое множество математических функций.

управляемый переключатель — элемент схемы, который может размыкать соединение.

управляющая линия — проводник для передачи сигнала управления (в отличие от данных).

формальный язык — множество строк над заданным алфавитом.

функциональный интерфейс — интерфейс, состоящий ровно из одного метода.

функция — см. *статический метод.*

хеширование — преобразование значения типа данных («ключа») в целое число из заданного диапазона так, чтобы разные ключи с большой вероятностью соответствовали разным целым числам.

хеш-таблица — реализация таблицы символов на основе хеширования.

центральный процессор — устройство, реализующее вычислительные функции компьютера.

цикл — конструкция в языке программирования, многократно выполняющая вычисления в зависимости от значения некоторого логического выражения (например, циклы `for` или `while`).

чистая функция — функция, которая для одного и того же набора аргументов всегда возвращает одно и то же значение без каких-либо побочных эффектов.

шестнадцатеричная система счисления — система счисления с основанием 16; подразумевается представление целых чисел.

шинное соединение — см. *магистраль данных.*

шинный мультиплексор — мультиплексор для переключения шинных соединений.

экземпляр — объект конкретного класса.

элемент (коллекции) — один из объектов в коллекции.

элемент (массива) — один из объектов или значений примитивного типа в массиве.

Юникод — международный стандарт представления символов текста.

API

public class Math

`double abs(double a)`	абсолютное значение (модуль) a
`double max(double a, double b)`	максимальное из значений a и b
`double min(double a, double b)`	минимальное из значений a и b

Примечание 1: abs(), max() и min() также определяются для типов int, long и float.

`double sin(double theta)`	синус theta
`double cos(double theta)`	косинус theta
`double tan(double theta)`	тангенс theta

Примечание 2: Углы задаются в радианах. Для преобразования угловых единиц используйте методы toDegrees() и toRadians().

Примечание 3: Обратные тригонометрические функции вычисляются методами asin(), acos() и atan()

`double exp(double a)`	экспонента (е в степени a)
`double log(double a)`	натуральный логарифм (logea, или ln a)
`double pow(double a, double b)`	возведение a в степень b (ab)
`long round(double a)`	округление a до ближайшего целого
`double random()`	случайное число в диапазоне [0, 1)
`double sqrt(double a)`	квадратный корень из a
`double E`	число e (константа)
`double PI`	число π (константа

public class String (тип данных Java)

`String(String s)`	Создает строку с таким же значением, как у s
`int length()`	Длина строки
`char charAt(int i)`	Символ в позиции с индексом i
`String substring(int i, int j)`	Символы с индексами от i до (j-1)
`boolean contains(String sub)`	Содержит ли строка подстроку sub?
`boolean startsWith(String pre)`	Начинается ли строка с префикса pre?
`boolean endsWith(String post)`	Заканчивается ли строка суффиксом post?
`int indexOf(String p)`	Индекс первого вхождения p
`int indexOf(String p, int i)`	Индекс первого вхождения p после i
`String concat(String t)`	Результат присоединения t к строке
`int compareTo(String t)`	Сравнение строк
`String replaceAll(String a, String b)`	Строка, в которой все вхождения a заменяются на b
`String[] split(String delim)`	Строка между вхождениями delim
`boolean equals(String t)`	Совпадает ли значение строки со значением t?

public class System.out/StdOut/Out

`Out(String name)`	Создает выходной поток для name
`void print(String s)`	Выводит s
`void println(String s)`	Выводит s и символ новой строки в поток
`void println()`	Выводит символ новой строки в поток
`void printf(String format, ...)`	Выводит аргументы в стандартный вывод в формате, заданном форматной строкой format

Примечание: Для System.out/StdOut методы являются статическими, а конструкторы не действуют.

public class StdIn/In

`In(String name)`	Создает входной поток для name
Методы экземпляров для чтения лексем из потока ввода	
`boolean isEmpty()`	Входной поток пуст (или содержит только пропуски)?
`int readInt()`	Читает лексему, преобразует ее в int и возвращает
`double readDouble()`	Читает лексему, преобразует ее в double и возвращает
`boolean readBoolean()`	Читает лексему, преобразует ее в boolean и возвращает
`String readString()`	Читает лексему и преобразует ее в String
Методы экземпляров для чтения символов из потока ввода	
`boolean hasNextChar()`	В потоке еще остались символы?
`char readChar()`	Читает символ из потока и возвращает его
Методы экземпляров для чтения текстовых строк из потока ввода	
`boolean hasNextLine()`	В потоке еще остались текстовые строки?
`String readLine()`	Читает остаток текстовой строки и возвращает его в формате String
Методы экземпляров для чтения оставшихся данных из потока ввода	
`int[] readAllInts()`	Читает все оставшиеся лексемы; возвращает результат в виде массива int
`double[] readAllDoubles()`	Читает все оставшиеся лексемы; возвращает результат в виде массива double
`double[] readAllBooleans()`	Читает все оставшиеся лексемы; возвращает результат в виде массива boolean
`double[] readAllStrings()`	Читает все оставшиеся лексемы; возвращает результат в виде массива String
`double[] readAllLines()`	Читает все оставшиеся лексемы; возвращает результат в виде массива String
`double[] readAll()`	Читает все оставшиеся лексемы; возвращает результат в виде String

Примечание 1: Для StdIn методы являются статическими, а конструкторы не действуют.

Примечание 2: Лексема (token) представляет собой максимальную последовательность символов, не являющихся пропусками (whitespace).

Примечание 3: Перед чтением лексемы все начальные пропуски отбрасываются.

Примечание 4: Аналогичные методы существуют для чтения значений типа byte, short, long и float.

Примечание 5: Все методы чтения данных выдают исключение времени выполнения, если им не удается прочитать следующее значение (из-за отсутствия данных или из-за того, что ввод не соответствует нужному типу).

public class StdDraw	
`Draw()`	Создает новый объект Draw
Команды рисования	
`void line(double x0, double y0, double x1, double y1)`	
`void point(double x, double y)`	
`void circle(double x, double y, double radius)`	
`void filledCircle(double x, double y, double radius)`	
`void square(double x, double y, double radius)`	
`void filledSquare(double x, double y, double radius)`	
`void rectangle(double x, double y, double r1, double r2)`	
`void filledRectangle(double x, double y, double r1, double r2)`	
`void polygon(double[] x, double[] y)`	
`void filledPolygon(double[] x, double[] y)`	
`void text(double x, double y, String s)`	
Управляющие команды	
`void setXscale(double x0, double x1)`	Выбирает масштаб по оси x (x0, x1)
`void setYscale(double y0, double y1)`	Выбирает масштаб по оси y (y0, y1)
`void setPenRadius(double radius)`	Назначает радиус пера radius
`void setPenColor(Color color)`	Назначает цвет пера color
`void setFont(Font font)`	Выбирает шрифт font
`void setCanvasSize(int w, int h)`	Назначает размер холста wґ
`void enableDoubleBuffering()`	Включает двойную буферизацию
`void disableDoubleBuffering()`	Отключает двойную буферизацию
`void show()`	Копирует содержимое вторичного холста в основной
`void clear(Color color)`	Заполняет холст цветом color
`void pause(int dt)`	Делает паузу продолжительностью dt миллисекунд
`void save(String filename)`	Сохраняет данные в формате .jpg или .png

Примечание 1: Для StdDraw методы являются статическими, а конструкторы не действуют.

Примечание 2: Одноименные методы без аргументов выполняют сброс к значениям по умолчанию.

public class StdAudio	
`void play(String filename)`	Воспроизводит заданный файл .wav
`void play(double[] a)`	Воспроизводит заданный набор звуковых данных
`void play(double x)`	Воспроизводит данные на 1/44 100 секунды
`void save(String filename, double[] a)`	Сохраняет данные в файле .wav
`double[] read(String filename)`	Читает данные из файла .wav

public class Stopwatch	
Stopwatch()	Создает новый объект секундомера и запускает его
double elapsedTime()	Возвращает время, прошедшее с момента его создания (в секундах)

public class Picture	
Picture(String filename)	Создает изображение по содержимому файла
Picture(int w, int h)	Создает пустое изображение wґ
int width()	Возвращает ширину изображения
int height()	Возвращает высоту изображения
Color get(int col, int row)	Возвращает цвет пиксела (col, row)
void set(int col, int row, Color c)	Назначает пикселу (col, row) цвет c
void show()	Выводит изображение в окне
void save(String filename)	Сохраняет изображение в файле

public class StdRandom	
void setSeed(long seed)	Инициализирует генератор для получения воспроизводимых результатов
int uniform(int n)	целое число в диапазоне от 0 до n-1
double uniform(double lo, double hi)	число с плавающей точкой в диапазоне от lo до hi
boolean bernoulli(double p)	true с вероятностью p, false в противном случае
double gaussian()	распределение Гаусса, математическое ожидание 0, среднеквадратическое отклонение 1
double gaussian(double mu, double sigma)	распределение Гаусса, математическое ожидание mu, среднеквадратическое отклонение sigma
int discrete(double[] p)	i с вероятностью p[i]
void shuffle(double[] a)	случайным образом переставляет элементы массива a[]

public class StdArrayIO	
double[] readDouble1D()	Читает одномерный массив с элементами типа double
double[][] readDouble2D()	Читает двумерный массив с элементами типа double
void print(double[] a)	Выводит одномерный массив с элементами типа double
void print(double[][] a)	Выводит двумерный массив с элементами типа double

Примечание 1. Формат для одномерных версий: целое значение n, за которым следуют n значений.

Примечание 2. Формат для двумерных версий: два целых значения m и n, за которыми следуют m×n значений по строкам.

Примечание 3. Также включены методы для int и boolean.

public class StdStats	
double max(double[] a)	Наибольшее значение
double min(double[] a)	Наименьшее значение
double mean(double[] a)	Математическое ожидание
double var(double[] a)	Дисперсия выборки
double stddev(double[] a)	Выборочное среднеквадратическое отклонение
double median(double[] a)	Медиана
void plotPoints(double[] a)	Рисует точки (i, a[i])
void plotLines(double[] a)	Рисует линии, соединяющие точки (i, a[i])
void plotBars(double[] a)	Рисует столбцы для точек (i, a[i])

Примечание: В библиотеку включены перегруженные реализации для других числовых типов.

public class Stack<Item> implements Iterable<Item>	
Stack()	Создает пустой стек
boolean isEmpty()	Стек пуст?
int size()	Количество элементов в стеке
void push(Item item)	Вставляет элемент в стек
Item pop()	Возвращает и извлекает элемент, который был вставлен последним

public class Queue<Item> implements Iterable<Item>	
Queue()	Создает пустую очередь
boolean isEmpty()	Очередь пуста?
int size()	Количество элементов в стеке
void enqueue(Item item)	Вставляет элемент в очередь
Item dequeue()	Возвращает и извлекает элемент, который был вставлен последним

public class SET<Key extends Comparable<Key>> implements Iterable<Key>	
SET()	Создает пустое множество
boolean isEmpty()	Множество пусто?
int size()	Количество элементов во множестве
void add(Key key)	Добавляет ключ key в множество
void remove(Key key)	Удаляет ключ key из множества
boolean contains(Key key)	Ключ key встречается во множестве?

public class ST<Key extends Comparable<Key>, Value>

`ST()`	Создает пустую символическую таблицу
`void put(Key key, Value val)`	Связывает val с key
`Value get(Key key)`	Значение, связанное с ключом key
`void remove(Key key)`	Удаление ключа key (и связанного с ним значения)
`boolean contains(Key key)`	Связано ли с ключом key какое-либо значение?
`int size()`	Количество пар «ключ-значение»
`Iterable<Key> keys()`	Все ключи в символической таблице
`Key min()`	Наименьший ключ
`Key max()`	Наибольший ключ
`int rank(Key key)`	Количество ключей, меньших key
`Key select(int k)`	k-й меньший ключ в символической таблице
`Key floor(Key key)`	Наибольший ключ, меньший либо равный key
`Key ceiling(Key key)`	Наименьший ключ, больший либо равный key

public class Graph

`Graph()`	Создает пустой граф
`Graph(In in, String delimiter)`	Читает данные графа из входного потока in с ограничителем delimiter
`void addEdge(String v, String w)`	Добавляет ребро v-w
`int V()`	Количество вершин
`int E()`	Количество ребер
`Iterable<String> vertices()`	Вершины графа
`Iterable<String> adjacentTo(String v)`	Соседи v
`int degree(String v)`	Количество соседей v
`boolean hasVertex(String v)`	Является ли v вершиной графа?
`boolean hasEdge(String v, String w)`	Является ли v-w ребром графа?

Роберт Седжвик, Кевин Уэйн

Computer Science: основы программирования на Java, ООП, алгоритмы и структуры данных

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Научный редактор	*С. Сиротко*
Художественный редактор	*С. Заматевская*
Корректоры	*Н. Викторова, Н. Витько*
Верстка	*Л. Егорова*

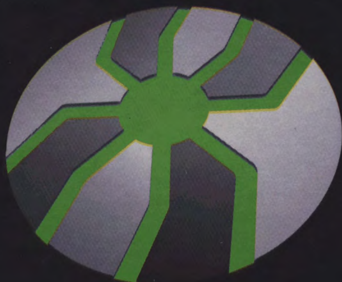
Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург,
улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 02.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 25.01.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 86,430. Тираж 2000. Заказ № ВЗК-00455-18.

Отпечатано в АО «Первая Образцовая типография», филиал «Дом печати — ВЯТКА»
в полном соответствии с качеством предоставленных материалов.
610033, г. Киров, ул. Московская, 122. Факс: (8332) 53-53-80, 62-10-36
http://www.gipp.kirov.ru; e-mail: order@gipp.kirov.ru



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные
программные продукты

КЛАССИКА COMPUTER SCIENCE

Преподаватели Принстонского университета Роберт Седжвик и Кевин Уэйн создали универсальное введение в Computer Science на языке Java, которое идеально подходит как студентам, так и профессионалам. Вы начнете с основ, освоите современный курс объектно-ориентированного программирования и перейдете к концепциям более высокого уровня: алгоритмам и структурам данных, теории вычислений и архитектуре компьютеров.

И главное — вся теория рассматривается на практических и ярких примерах: прикладная математика, физика и биология, числовые методы, визуализация данных, синтез звука, обработка графики, финансовое моделирование и многое другое.

«Наша главная цель — дать конкретные знания и навыки, необходимые для разработки эффективных решений любой задачи по программированию».

Роберт Седжвик и Кевин Уэйн

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг
и интернет-магазин

vk.com/piterbooks
instagram.com/piterbooks
facebook.com/piterbooks
youtube.com/ThePiterBooks

ISBN 978-5-496-02700-7