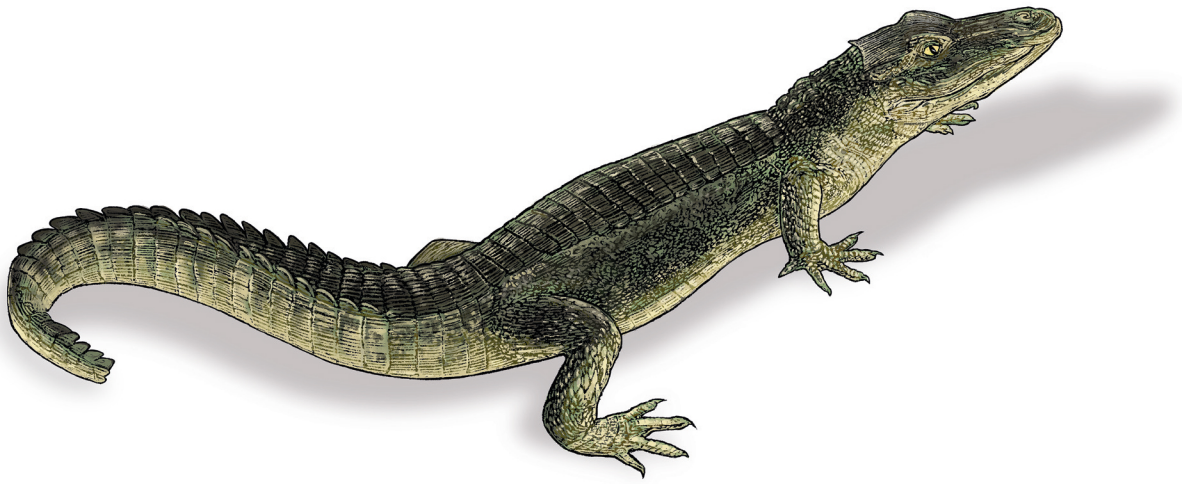
O'REILLY®

Элегантный код Java

Дженерики и коллекции



Морис Нафтален, Филип Уодлер
при участии Стюарта Маркса

BOOKS.KZ

Морис Нафтален, Филип Уодлер
при участии Стюарта Маркса

Элегантный код Java: дженерики и коллекции

Java Generics and Collections

Fundamentals and Recommended Practices

Maurice Naftalin and Philip Wadler
with Stuart Marks

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Элегантный код Java: дженерики и коллекции

Теоретические основы
и практическое применение

Морис Нафтален, Филип Уодлер
при участии Стюарта Маркса

УДК 004.438Java
ББК 32.973.2
Н34

Морис Нафтален, Филип Уодлер
Н34 Элегантный код Java: дженерики и коллекции / пер. с англ. А. А. Слинкина. – Алматы.: Books.kz, 2025. – 344 с.: ил.

ISBN 978-6-01140-648-2

Это обновленное издание самого известного руководства охватывает Java 21 и предлагает актуальную и точную информацию о параметризованных типах (дженериках). Вы узнаете все необходимое для эффективного использования и написания параметризованных API. В книге подробно описана библиотека коллекций, проведено сравнение потоков и коллекций, рассмотрен выбор оптимальной модели. Это поможет читателю получить максимум пользы от платформенной библиотеки.

Книга адресована всем, кто хочет узнать больше о Java. Предполагается, что читатель владеет начальными навыками программирования на этом языке и знаком с основными понятиями.

УДК 004.438Java
ББК 32.973.2

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Authorized Russian translation of the English edition of Java Generics and Collections, 2nd Edition ISBN 9781098136727

ISBN (анг.) 978-1-09813-672-7
ISBN (казах.) 978-6-01140-648-2

Посвящаем эту книгу
Джойс Нафтален, Лайонелю Нафталену,
Адаму Уодлеру и Леоре Уодлер

Морис Нафтален, Филип Уодлер

Оглавление

Об авторах

Морис Нафтален имеет за плечами более 50 лет работы в ИТ-индустрии в качестве разработчика, проектировщика, архитектора, руководителя, преподавателя и автора. Он использует и преподает Java со времен JDK 1.0. Он обладатель звания Java Champion, автор книги «Mastering Lambdas» (2015), часто выступает на конференциях по всему миру. Он дезорганизует ежегодную неконференцию на тему Java – JAlba.

Филип Уодлер – профессор теоретической информатики в Эдинбургском университете, Шотландия, занимается исследованиями в области функционального и логического программирования и проектирования языков программирования. Являлся главным проектировщиком языка программирования Haskell и одним из проектировщиков GJ, работы, положенной в основу дженериков Java. Профессор Уодлер защитил докторскую диссертацию по информатике в университете Карнеги–Меллона.

Стюарт Маркс – руководитель проекта JDK Core Libraries в группе платформе Java в компании Oracle. В настоящее время занимается сопровождением каркаса коллекций Java. Поскольку его второе я – «доктор Депрекатор», он также работает над механизмом вывода из эксплуатации в Java. У него за плечами свыше 30 лет работы над разработками платформенных программных продуктов в области оконных систем, интерактивной графики, а также мобильных и встраиваемых систем. Имеет степень магистра информатики, полученную в Стэнфордском университете.

Об иллюстрации на обложке

На обложке книги изображен аллигатор. Аллигаторы встречаются только в южных частях США и в Китае. В Китае они редки и водятся только в бассейне реки Янцзы. Как правило, аллигаторы не переносят соленую воду, поэтому обитают в пресноводных прудах, болотах и других водоемах.

Новорожденные аллигаторы совсем небольшие, около 15 см длиной. Но в первые годы жизни они растут очень быстро – по 30 см в год. Длина взрослой самки примерно 270 см, а вес – от 68 до 91 кг. Длина взрослого самца примерно 330 см, а вес – от 159 до 181 кг. Самый крупный аллигатор, найденный в Луизиане в начале 1900-х годов, достигал в длину 584 см. Основной внешний признак аллигатора – короткая широкая морда. Шкура взрослого аллигатора серо-черного цвета, при намокании становится черной. Брюхо белое. На спине у молодых аллигаторов видны желтые и белые полосы. Форма морды и цвет шкуры – физические признаки, отличающие аллигаторов от крокодилов, у которых морда тонкая, а цвет рыжевато-коричневый.

Аллигаторы ведут преимущественно ночной образ жизни, они охотятся и питаются после захода солнца. Они хищники, в рационе которых разнообразная пища: черепахи, рыбы, лягушки, птицы, змеи, небольшие млекопитающие и даже менее крупные аллигаторы. У взрослого аллигатора практически нет врагов – кроме человека.

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу *dmkpress@gmail.com*, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

О переводе терминов

В данной книге говорится как о парадигме параметризованных типов в целом, так и о дженериках как о механизме реализации типобезопасности в Java. Но авторы предпочитают использовать общий термин generic, что создает некоторую путаницу. В переводе применяются оба варианта, чтобы обеспечить плавный переход от общего понятия типобезопасности к дженерикам и коллекциям Java.

Предисловие

Второе издание этой книги значительно переработано для версии Java 21. Главным побудительным мотивом для него было появление последовательных коллекций и первые последствия потоков и лямбда-выражений, поэтому еще до начала работы я ожидал, что в основном изменениям придется подвергнуть часть II – из-за последовательных коллекций. Но, как оказалось, я сильно недооценил масштаб изменений, произошедших с Java за последние 20 лет, и в итоге понадобилось вносить многочисленные поправки и в часть I тоже. Язык изменился; теперь он поддерживает записи, вывод типов локальных переменных, выражения сопоставления с образцом и параметризованные типы в выражениях сравнения типов. Кроме того, большую часть примеров в части I пришлось переработать с учетом изменений в платформенных библиотеках, в том числе объявления нерекомендуемыми конструкторов `Integer`, а также появления статических методов в интерфейсе `Comparator`, немодифицируемых коллекций и потоков.

Важной причиной для написания второго издания стало документирование некоторых выработанных сообществом Java с 2005 года приемов работы с параметризованными типами и коллекциями. По большей части они собраны в ретроспективном разделе «О дизайне параметризованных типов в Java» и в двух новых главах о коллекциях: в главе 17, в которой приведены наставления по использованию каркаса коллекций, и в главе 18, в которой дается обзор самых важных – и вместе с тем самых спорных – решений, положенных в основу дизайна каркаса. Кроме того, главы о сбережении (глава 5) и об эффективном использовании параметризованных типов (глава 7) подверглись существенной переработке, глава о паттернах проектирования исключена, а глава о миграции кода, написанного до появления параметризованных типов, доступна только в сети в виде приложения (*https://mauricenaftalin.github.io/JGC_2e_Book_Code/appendix.html*) как представляющая лишь технический и исторический интерес.

В конце этого раздела я поместил, хотя и немного в измененной форме, предисловие к первому изданию. Мне казалось важным передать то чувство волнения, которое охватывало меня при виде тщательно продуманного набора функциональности, с которым версия Java 5 шагнула в XXI век.

— Морис Нафталин,
Эдинбург, февраль 2025

Благодарности за второе издание

Перво-наперво я хочу поблагодарить всех тех, кто, прочитав первое издание, все же побуждал меня приняться за его пересмотр. На протяжении всего проекта я ощущал поддержку со стороны сообщества Java; я считаю, что мне повезло быть частью такого благосклонного и отзывчивого сообщества.

Сотрудник издательства O'Reilly Зэн Маккуэйд оказал мне неоценимую помощь в подготовке обоснования нового издания. Мой редактор, Сара Хантер, всегда была готова прийти на помощь, когда она была особенно необходима. Мой корректор, Рэйчел Хэд, обладает потрясающим шестым чувством, позволяющим ей находить технические проблемы. Мой выпускающий редактор, Алия Рахман, доблестно исправляла многочисленные мелкие, но существенные погрешности в тексте.

Преимуществом работы над следующим изданием является возможность исправить ошибки, оставшиеся в предыдущем (и попытаться не насажать при этом новых). Из читателей, указавших мне на опечатки в первом издании, хочу отметить Валентина Бака, Томаса Костика, Свейна Эгила Нильсена, Эмануэля Фруа, Андерса Грэнлунда, Равиндра Ранвала, Рэндольфа Ротфусса, Яна де Рюйтера и Кристофера Занвальдта.

Я благодарен многим пожертвовавшим своим временем, чтобы оставить отзыв о книге, предложить исправления, поделиться полезными идеями, крупными и не очень. В некоторых случаях они очень щедро делились со мной временем. Это Мохамед Аниис, Ларри Кейбл, Рэй Джаджадината, Брайан Гётц, Мэри Гоусети, Андржей Гржечик, Джородж Хейнеман, Кэй Хорстманн, Тимми Джоуз, Хайнц Кабутц, Марк Лой, Крис Ньюлэнд, Скотт Оукс, Саймон Парк, Жозе Помар, Саймон Риттер, Роберт Шольте, Дэниэл Шайя, Иван Шипка, Фред Смит, Тушар Сривастава и Йенс-Хаген Сирбе. Разумеется, все оставшиеся ошибки и упущения – целиком моя вина.

Счастлив повторить, что спустя 19 лет включенная в первое издание рекомендация бюллетеня Хайнца Кабутца «The Java Specialists' Newsletter» все еще остается в силе – и это после 325 выпусков!

С удовольствием признаю, что многие идеи, изложенные в этом издании, рождались в сотрудничестве со Стюартом Марксом. Они оттачивались в многочисленных долгих обсуждениях и совместных презентациях. Опыт, знания и поддержка делают его лучшим техническим редактором, о котором любой автор может только мечтать.

Фил Уодлер был и остается моим добрым приятелем. Как обычно, свидетельствую свою любовь Бену, Дэниэлю, Айзеку, Джо и Рут.

Для кого написана эта книга

Эта книга предназначена всем, кто что-то знает о Java и хотел бы узнать больше. Это не учебное пособие по Java, поэтому мы предполагаем, что читатель знаком с такими базовыми понятиями Java, как классы, статические методы, методы экземпляра и т. д. Мы используем синтаксис Java 21, но большинство синтаксических конструкций, относящихся конкретно к параметризованным

классам, объясняется в тексте. Если вы будете испытывать затруднения с пониманием синтаксиса какого-то кода, то, возможно, ваша интегрированная среда разработки сможет предложить вариант рефакторинга, который прояснит его назначение.

Мы предполагаем наличие знаний самого начального уровня об API коллекций. Если вам доводилось писать программы с использованием `ArrayList` или `HashSet`, то никаких трудностей не возникнет. Обсуждая поведение тех или иных классов коллекций и их методов, мы часто ссылаемся на документацию – она всегда дает исчерпывающую спецификацию.

Где взять примеры кода

Примеры кода для этого издания имеются в проекте Maven по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code*. Отметим, что некоторые примеры не компилируются по причинам, указанным в относящемся к ним тексте.

Вопросы технического характера, а также замечания по примерам кода следует отправлять по адресу *support@oreilly.com*.

Вы можете использовать приведенный в книге код в собственных программах и в документации. Спрашивать у O'Reilly Media разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Java Generics and Collections, 2nd Edition, by Maurice Naftalin and Philip Wadler (O'Reilly). Copyright 2025 Morningside Light, 978-1-098-13672-7».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Графические выделения

В книге применяются следующие графические выделения:

- код набран моноширинным шрифтом, а для дополнительного выделения используется полужирный шрифт:

```
class InnocentClient {
   public static void main(String[] args) {
      List<Integer>[] intLists = DeceptiveLibrary.createIntLists(1);
      List<? extends Number>[] numLists = intLists;
      numLists[0] = List.of(1.01);
      int i = intLists[0].get(0);  // исключение приведения к типу класса!
   }
}
```

- мы часто включаем код, принадлежащий телу метода `main`:

```
List<Integer>[] intLists = DeceptiveLibrary.createIntLists(1);
List<? extends Number>[] numLists = intLists;
numLists[0] = List.of(1.01);
int i = intLists[0].get(0); // исключение приведения к типу класса!
```

  В репозитории кода эти фрагменты собраны в методы `main`. Зачастую несколько фрагментов собираются в один класс. Отметим, что не все классы в репозитории компилируются: объяснения ошибок компиляции приводятся в тексте;

- включенные в основной текст фрагменты кода, в в том числе такие элементы программы, как имена переменных и функций, баз данных, типов данных и переменных окружения, предложения и ключевые слова языка, печатаются `моноширинным` шрифтом (как в примере имени метода `main` в предыдущем абзаце);
- директивы импорта в тексте опущены. Но они имеются в репозитории;
- примеры интерактивных сеансов, содержащие командную строку и соответствующий вывод, набраны `моноширинным` шрифтом, а данным, вводимым пользователем, предшествует знак процента:

```
% javac g/Stack.java g/ArrayStack.java g/Stacks.java l/Client.java
Note: Client.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

  Иногда мы приводим фрагмент сеанса JShell, в котором за приглашением и введенными пользователем данными следует вывод JShell:

```
jshell> List.of("Larry", "Curly", "Larry", "Moe")
$2 ==> [Larry, Curly, Larry, Moe]
```

- когда пользовательские данные не умещаются в одной строке, первая строка заканчивается обратной косой чертой:

```
% javac -Xlint:unchecked g/Stack.java g/ArrayStack.java \
%    g/Stacks.java l/Client.java
l/Client.java:4: warning: [unchecked] unchecked call
to push(E) as a member of the raw type Stack
      for (int i = 0; i<4; i++) stack.push(new Integer(i));
```

- новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов набраны *курсивом*;
- текст, набранный *моноширинным курсивом*, следует заменить предоставленными пользователем данными или значениями, вытекающими из контекста.

Так обозначается замечание общего характера.

Утверждения

Для большей ясности мы щедро уснастили код предложениями `assert` и ожидаем, что утверждения включены. По умолчанию они выключены, поэтому, чтобы добиться ожидаемого поведения, вы должны включить их, вызвав виртуальную машину Java (JVM) с флагом `-ea` или `-enableassertions`. За каждым словом `assert` следует булево выражение, которое в нормальной ситуации должно принимать значение `true`. Если утверждения включены, а выражение равно `false`, то возбуждается исключение `AssertionError`, показывающее, в частности, где произошла ошибка.

Мы включаем только утверждения, которые должны принимать значение `true`. Поскольку они могут быть выключены, утверждение никогда не должно иметь побочных эффектов, от которых зависит не входящий в состав утверждения код. При проверке условия, которое должно быть выполнено (например, что аргументы метода допустимы), мы используем условное предложение и явно возбуждаем исключение.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (в США и Канаде)
707-827-7019 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *https://www.oreil.ly/fluent-c*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *https://oreil.ly/java-generics-and-collections2e*.

Новости и информация о наших книгах и курсах публикуются на сайте *https://oreilly.com*.

Ищите нас в LinkedIn: *https://linkedin.com/company/oreilly-media*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Предисловие к первому изданию

Включение *параметризованных типов* в язык Java, сопровождаемое рядом других улучшений, кардинально изменило практику программирования на Java. Самый очевидный эффект – работа с типами коллекций, например `List`. Эти классы всегда можно было использовать для хранения объектов разных типов, например `String` или `Integer`, но параметризация типов добавила воз-

можность точно сказать, элементы какого типа должна содержать коллекция, и проверить выполнение этого ограничения.

Допустим, вы хотите обрабатывать списки. Одни из них могут содержать целые, другие – строки, а третьи – списки строк. До появления в Java параметризованных типов это было просто. Все три списка можно было представить одним и тем же классом `List`, содержащим элементы класса `Object`:

```
список целых          List
список строк          List
список списков строк  List
```

Чтобы язык оставался простым, вы были вынуждены проделывать часть работы самостоятельно: нужно было помнить, что данный список содержит целые (или строки или списки строк), и после извлечения из него элемента приводить его от типа `Object` обратно к типу `Integer` (или `String`, или `List`). В каркасе коллекций (Collections Framework), существовавшем до появления параметризованных типов, эта идиома широко использовалась.

Как говорил Эйнштейн, «все должно быть изложено настолько просто, насколько возможно, но не проще»[1]. И кто-то скажет, что описанный выше подход слишком прост. В Java с параметризованными классами можно различить типы списков:

```
список целых          List<Integer>
список строк          List<String>
список списков строк  List<List<String>>
```

Теперь компилятор помнит, что вы создали список целых (или строк, или списков строк), и явного приведения обратно к типу `Integer` (или `String`, или `List<String>`) не требуется. В определенных отношениях это похоже на *дженерики* (generics) в Ada или *шаблоны* в C++, но истинные корни лежат в *параметрическом полиморфизме*, встречающемся в таких функциональных языках, как ML или Haskell.

Часть I этой книги содержит подробное введение в параметризованные типы. Мы обсудим взаимодействие между ними и подтипами, использование джокеров (wildcard) и границ. Мы опишем приемы развития своего кода, остановимся на тонкостях, связанных с приведением типов и массивами, рассмотрим такие продвинутые темы, как взаимодействие между параметризованными типами и безопасностью и обеспечение двоичной совместимости. А также модифицируем хорошо известные паттерны проектирования с учетом дженериков.

Включение параметризованных типов в Java вызвало споры. Разумеется, при проектировании механизма дженериков были допущены компромиссы: чтобы развивать код было легко, объекты не должны *сберегать* (reify) информацию времени компиляции о параметрах типов, т. е. не должны сохранять эту

[1] Это популярное переложение. На самом деле Эйнштейн говорил, что теория должна быть настолько простой, насколько возможно, «не жертвуя при этом адекватным представлением никаких экспериментальных данных» (Calaprice 2011, 384-5). Код, написанный до появления параметризованных классов, этому критерию не удовлетворяет: он не способен представить важные аспекты реальности, а именно тип объектов, хранящихся в списке.

информацию для использования на этапе выполнения; однако ее отсутствие на этапе выполнения приводит к появлению некоторых пограничных случаев в таких операциях, как приведение типов и создание массивов. Мы предлагаем сбалансированное изложение идеологии параметризованных типов, объясняя, как обращать себе во благо их сильные стороны и обходить слабые.

Часть II содержит полное введение в каркас коллекций. Английский ученый-энциклопедист Исаак Ньютон в 1675 году писал: «Если я видел дальше, то это потому, что стоял на плечах гигантов». Лучшие программисты работают в соответствии с этим девизом – пишут код на базе уже имеющихся каркасов и библиотек и повторно используют код, когда это разумно. Каркас коллекций в Java предлагает повторно используемые интерфейсы и реализации для многих типичных типов коллекций, включая списки, множества, очереди и отображения[1].

По сравнению с написанным ранее кодом, код, где используются параметризованные коллекции, проще читать, а компилятор обнаруживает больше ошибок типизации. Кроме того, коллекции дают прекрасные примеры использования дженериков. Можно сказать, что дженерики и коллекции созданы друг для друга, и действительно желание упростить работу с коллекциями было одной из основных причин внедрения параметризованных типов. В части II мы опишем принципы, на которых построен каркас коллекций, чтобы вы могли использовать коллекции более эффективно.

На дизайн дженериков в Java оказали влияние многие более ранние предложения и прежде всего язык GJ (см. Bracha et al., 1998), добавление джокеров в GJ, предложенное в работе Igarashi and Viroli (2006), и дальнейшее развитие идеи джокеров в работе Torgersen et al. (2004). Проектирование параметризованных типов шло в рамках процесса Java Community Process силами группы под руководством Брача, которая включала Одерски, Торупа и Уолдера (части JSR 14 и JSR 201). Компилятор GJ, написанный Одерски, был положен в основу оригинального компилятора `javac`, поддерживавшего дженерики.

Благодарности, включенные в первое издание

Сотрудники Oracle, ранее работавшие в Sun Microsystems, неукоснительно отвечали на наши вопросы. Они с удовольствием объясняли сложные моменты и рассказывали о спорах по поводу компромиссов. Мы благодарим Джошуа Блоха, Гилада Брача, Мартина Бухгольца, Джозефа Д. Дарси, Нила М. Гафтера, Марка Рейнгольда, Дэвида Стоутамайра, Скотта Вайолета и Питера фон дер Ахе.

Мы были рады работать со следующими исследователями, внесшими вклад в дизайн дженериков в Java: Эрик Эрнст, Кристиан Плезнер Хансен, Атсуши Игараши, Мартин Одерски, Мадс Торгерсен и Мирко Вироли.

Мы получили отзывы и комментарии от многих людей. Благодарим Брайана Гётца, Дэвида Холмса, Хайнца М. Кабутца, Деепти Калра, Анжелику Лангер, Стефана Либега, Дуга Леа, Тима Монро, Стива Мэрфи и К. К. Шибина.

[1] Конечно, не все программисты так уважительно относятся к работе своих предшественников. Иногда, как говорил Ричард Хэмминг о специалистах по информатике, «вместо того чтобы стоять на плечах других, мы наступаем друг другу на пятки».

Мы с удовольствием читали бюллетень «The Java Specialists' Newsletter» Хайнца М. Кабутца и «Java Generics FAQ» Анжелики Лангер; обе работы доступны в сети.

Наш редактор, Майкл Лукидес, всегда был готов дать хороший совет. Пол К. Анагностопулос из компании Windfall Software превратил наш текст на LaTeX в оригинал-макет, а Джереми Яллоп составил предметный указатель.

Наши семьи помогли нам не сойти с ума (и сводили с ума). Адам, Бен, Кэтрин, Дэниэл, Айзек, Джо, Леора, Лайонел и Рут – мы вас любим.

Часть I

Дженерики

Дженерики – мощное, а временами спорное средство языка программирования Java. В этой части книги описываются дженерики с использованием каркаса коллекций в качестве источника примеров. Полное введение в каркас коллекций составляет предмет второй части книги.

Первые четыре главы посвящены теоретическим основам параметризации типов. Глава 1 содержит общий обзор параметризованных типов и методов. В главе 2 рассказывается, как работает подтипизация, и объясняется, как джокеры позволяют использовать подтипизацию в контексте параметризованных типов. В главе 3 описывается работа параметризованных типов с интерфейсом `Comparable`, для чего требуется понятие *границ* для переменных-типов. В главе 4 рассматривается работа параметризованных типов с различными объявлениями, в том числе конструкторами, статическими членами и вложенными классами. Освоив весь этот материал, вы сможете эффективно использовать дженерики Java в простых ситуациях.

В следующих трех главах рассматриваются продвинутые вопросы. В главе 5 объясняется, почему тот самый дизайн, который упрощает развитие кода, по необходимости приводит к нескольким неприятным пограничным случаям при работе с приведениями типов, исключениями и массивами. Сочетание дженериков с массивами – один из самых нехороших аспектов языка, и мы сформулируем два принципа, которые помогут вам обходить проблемы стороной. В главе 6 описываются механизмы соединения дженериков с рефлексией, в том числе тип `Class<T>` и добавления в библиотеку Java для поддержки рефлексии параметризованных типов. Глава 7 содержит рекомендации по применению дженериков на практике. Мы рассматриваем различные приемы, выкованные в процессе практической разработки с использованием дженериков. Здесь же имеется раздел, посвященный некоторым проектным решениям, сформировавшим облик механизма дженериков в Java.

В приложении объясняется, как Java удалось поддержать параметризованные типы, не отказываясь от совместимости с унаследованным кодом.

Глава 1

Введение

Цель этой главы – показать, как один и тот же код можно повторно использовать для создания или обработки объектов разных типов. Например, `List<String>` и `List<Integer>` – разные типы, но параметризация позволяет реализовать их с помощью одного и того же кода, причем механизм типобезопасности предотвратит любую возможность перепутать их. В Java параметризовать можно код двух видов: типы, например классы и интерфейсы коллекций, и методы, например статические методы в служебном классе `java.util.Collections`. Мы рассмотрим эти варианты по очереди.

Примеры кода к этой главе можно скачать по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter01*.

Параметризованные типы

В объявлении интерфейса или класса можно указать, что он принимает один или несколько *параметров-типов*, которые записываются в угловых скобках и должны быть предоставлены при объявлении переменной, принадлежащей этому интерфейсу или классу, а также при создании нового экземпляра класса.

Приведем пример[1]:

org/jgcbook/chapter01/A_generic_types/Program_1

```
List<String> words = new ArrayList<String>(); (1)
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);(2)
assert s.equals("Hello world!");
```

В каркасе коллекций класс `ArrayList<E>` реализует интерфейс `List<E>`, где `E` – *переменная-тип* – маркер, вместо которого будет подставлен тип при использовании `List` или `ArrayList` в программе. В этом тривиальном фрагменте объявлена переменная `words`, которая будет использоваться для ссылки на спи-

[1] Если вы смотрите на этот код в IDE, то можете увидеть сообщение – совершенно правильное – о том, что правая часть эквивалентна `new ArrayList<>()`. Такая запись означает, что компилятор должен подставить тип, указанный в объявлении переменной в левой части.

сок, содержащий строки. Далее создается экземпляр `ArrayList`, ссылка на него присваивается переменной `words`, в список добавляются две строки, которые затем извлекаются из него.

Не следует делать из этого примера вывод, будто переменная типа `SomeClass<T>` обязательно ссылается на объект `SomeClass`, *содержащий* объекты типа `T`. Если `SomeClass` – тип коллекции, то это действительно так, но сама идея параметризованных типов шире. Например, объекты, реализующие интерфейс `Comparable<T>`, не содержат объектов типа `T`, а обладают способностью сравнивать себя с объектами типа `T` (см. раздел «Тип Comparable» ниже). Другой пример дает класс `java.util.ServiceLoader<S>`, который находит поставщиков службы типа `S`, так что `ServiceLoader<CharsetProvider>` находит службы кодирования и декодирования наборов символов, `ServiceLoader<FileSystemProvider>` находит службы управления файловыми системами и т. д. Именно это мы имели в виду, говоря, что один и тот же код – *параметризованного класса* `SomeClass` – можно повторно использовать для создания или обработки объектов разных типов (т. е. типов, являющихся различными конкретизациями *параметра-типа* `T`). Поскольку мы в основном будем обсуждать параметризованные типы в контексте коллекций, извинительно думать, что они означают включение, но вообще-то идея находит более широкое применение.

Прежде чем генерировать байт-код, компилятор проверяет исходный код на предмет согласованного использования параметров-типов. В предыдущем примере он убеждается, что ссылку на экземпляр `ArrayList<String>` можно присвоить переменной типа `List<String>`. Если проверка прошла успешно, то компилятор отбрасывает параметры-типы и только потом начинает генерировать байт-код. Поэтому исходный код строчки, которая компилируется на самом деле, имеет вид

```
List words = new ArrayList();
```

Но теперь `words` является списком `Object`. Это означает, что `words::get` возвращает значение типа `Object`, и компилятор должен вставлять приведения к типу `String`, чтобы строка успешно компилировалась.

До появления в Java поддержки параметризованных типов тот же код нужно было бы написать следующим образом:

```
org/jgcbook/chapter01/A_generic_types/Program_2
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1));
assert s.equals("Hello world!");
```

По существу, это тот самый код, который компилятор параметризованных типов продолжает использовать для генерирования байт-кода после описанных выше преобразований, поэтому в обоих случаях байт-код будет одинаков. Мы говорим, что параметризованные типы реализованы посредством *стирания* типа, так как типы `List<Integer>` и `List<String>` – да и `List<List<String>>`, если на то пошло, – во время выполнения представлены одним и тем же типом `List`. Мы также используем термин «стирание», чтобы описать процесс преобразования первой программы во вторую. Более подробное описание стирания см. в разделе «Как работает стирание» ниже, а пока можете представлять его себе следующим образом:

стирание: выполняемый на этапе компиляции процесс, в ходе которого аннотации типов удаляются до начала генерирования байт-кода.

Термин «стирание» не совсем удачный, потому что включает в себя не только стирание параметров-типов, но и добавление приведения типов.

Как показывает этот пример, параметризованные типы неявно выполняют то же самое приведение, что и в явном коде без использования параметризации. Если бы такое преобразование завершилось неудачно, то отладить код, написанный с использованием параметризованных типов, было бы нелегко. Поэтому утешает, что механизм параметризованных типов включает следующую гарантию:

гарантия успешного приведения: неявные операции приведения, добавленные в ходе компиляции параметризованных типов, никогда не завершаются неудачно.

По поводу этой гарантии необходимо добавить примечание мелким шрифтом: она применима, только если не было *предупреждений о невозможности проверки*. Это предупреждения, которые компилятор выдает в ситуациях, когда не может гарантировать типобезопасность. Позже мы подробнее обсудим, что вызывает такие предупреждения и как минимизировать их влияние.

У реализации параметризованных типов посредством стирания типа был ряд важных последствий. Она позволила упростить систему, потому что параметризованные типы не требовали внесения изменений в JVM или в байт-код. Размер программы оставался под контролем, потому что существует ровно одна реализация `List`, а не по одной версии для каждого типа. И развивать код тоже было проще, потому что одну и ту же библиотеку можно было использовать как при использовании параметризованных типов, так и без них.

Последний момент нуждается в пояснениях. Он означает, что вы никогда не столкнетесь с неприятными проблемами из-за необходимости сопровождать две версии библиотек: *унаследованную*, которая работала, когда в Java не было параметризованных типов, и *параметризованную*. На уровне байт-кода код, в котором параметризованные типы используются, ничем не отличается от кода, в котором они не используются. А значит, нет никакой необходимости переходить на параметризованные типы одномоментно – можно постепенно развивать код, модифицируя его по одному пакету, классу или методу за раз, чтобы привыкнуть к параметризованным типам[1]. (Конечно, вышеупомянутая гарантия успешного приведения работает, только если вы добавляете параметризованные типы, совместимые с унаследованным кодом.)

Еще одно следствие реализации параметризованных типов посредством стирания заключается в том, что типы массивов в некоторых существенных отношениях отличаются от параметризованных типов. При выполнении строчки

```
new String[size]
```

выделяется память для массива, и в этом массиве сохраняется указание на то, что его элементы имеют тип `String`. С другой стороны, при выполнении строчки

```
new ArrayList<String>()
```

[1] В приложении мы даже объясняем, как можно было бы объявить параметризованные типы в унаследованном коде.

память для списка выделяется, но в самом списке нет никаких указаний на тип его элементов. Мы говорим, что Java *сберегает* тип элементов массива, т. е. сохраняет его для использования на этапе выполнения, но не сберегает тип элементов списка (или других параметризованных типов). Как уже было сказано выше, такой дизайн важен для постепенного развития кода, а стало быть, и для сохранения популярности Java. Но с годами он продолжает усложнять приведения, тесты экземпляров и создание массивов, как мы увидим в главе 5, в последнем разделе которой, «О дизайне параметризованных типов в Java», рассматриваются аргументы за и против стирания и обсуждается, остается ли это проектное решение правильным и сегодня.

Дженерики и шаблоны

Дженерики в Java напоминают шаблоны в C++. Но есть две важные вещи, о которых нужно помнить, рассуждая о связи между дженериками и шаблонами: синтаксис и семантика. Синтаксис сознательно сделан похожим, а семантика – столь же сознательно – иной.

Синтаксически угловые скобки были выбраны, потому что они знакомы пользователям C++ и потому что разбирать квадратные скобки было бы трудно. Семантически дженерики Java определены посредством стирания, а шаблоны C++ – посредством *расширения*. В C++ каждый экземпляр шаблона с новым типом компилируется отдельно. Если в программе имеется список целых, список строк и список списков строк, то будет три версии кода. Если вы пользуетесь списками элементов сотни разных типов, то будет сотня версий кода – эта проблема известна под названием *разбухание кода*. В Java сколько бы типов списков ни было, версия кода всегда одна, так что код не разбухает.

Расширение может приводить к более эффективной реализации, чем стирание, поскольку предлагает больше возможностей для оптимизации, особенно для примитивных типов, например `int`. Если программа работает с большими объемами данных – например, с большими массивами в научных расчетах, – то различие может быть существенным. Но для большинства практических целей это различие в эффективности не важно, тогда как как проблемы, вызванные разбуханием кода, могут оказаться критическими.

C++ также предлагает возможность манипулировать значениями, а не только типами. Эта техника, получившая название *метапрограммирование шаблонов*, позволяет использовать шаблоны как своего рода «накачанный макропроцессор», способный выполнять сколь угодно сложные вычисления на этапе компиляции. Параметризация в Java сознательно ограничена типами, чтобы избежать чрезмерной сложности.

Параметризованные методы и varargs-аргументы

В предыдущем разделе было сказано, что интерфейсы и классы могут принимать аргумент-тип. Отдельные методы тоже могут быть параметризованными. Ниже показан метод, который принимает массив произвольного типа и преобразует его в список того же типа.

org/jgcbook/chapter01/B_generic_methods_and_varargs/Lists_1

```
class Lists_1 {
  public static <T> List<T> toList(T[] arr) {
    List<T> list = new ArrayList<T>();
    for (T elt : arr) list.add(elt);
    return list;
  }
}
```

Статический метод `toList` принимает массив типа `T[]` и возвращает список типа `List<T>`, причем делает это для *любого* ссылочного типа `T`. Чтобы показать это, мы пишем `<T>` в начале объявления метода и тем самым объявляем `T` как новую переменную-тип. Переменная `T` может быть любым допустимым идентификатором Java, но по соглашению идентификаторы типов обычно обозначаются одиночными заглавными буквами: `T` для параметра-типа, `R` для типа возвращаемого значения, `U` для второго параметра-типа и т. д. Метод, в котором переменная-тип объявлена таким образом, называется *параметризованным* (или обобщенным) *методом*. Область видимости переменной-типа `T` ограничена самим методом; она может встречаться в объявлении метода, но не может вне метода.

Метод вызывается следующим образом:

org/jgcbook/chapter01/B_generic_methods_and_varargs/Lists_1

```
List<Integer> ints = Lists_1.toList(new Integer[] {1, 2, 3});
List<String> words = Lists_1.toList(new String[] { "Hello", "world!" });
```

В первой строке упаковка (boxing) (см. раздел «Примитивные и ссылочные типы» ниже) преобразует значения `1`, `2` и `3` в тип `Integer`.

Упаковка аргументов в массив выглядит громоздко. *Параметры переменной арности*, которые обычно называются просто *varargs*, предлагают специальный, более удобный синтаксис для случая, когда последний аргумент метода является массивом. Чтобы воспользоваться этой возможностью, мы заменяем `T[]` на `T...` в объявлении метода, так что получается объявление, очень похожее на `java.util.List::of`:

org/jgcbook/chapter01/B_generic_methods_and_varargs/Lists_2

```
class Lists_2 {
  public static <T> List<T> toList(T... arr) {
    List<T> list = new ArrayList<T>();
    for (T elt : arr) list.add(elt);
    return list;
  }
}
```

Теперь метод можно вызвать следующим образом:

org/jgcbook/chapter01/B_generic_methods_and_varargs/Lists_2

```
List<Integer> ints1 = Lists_2.toList(1, 2, 3);
List<String> words = Lists_2.toList("Hello", "world!");
```

Это просто сокращенная запись предыдущего кода. На этапе выполнения аргументы упаковываются в массив, который передается методу, – точно так же, как мы видели ранее.

Последнему varargs-аргументу может предшествовать произвольное число аргументов. Ниже показана упрощенная версия метода `java.util.Collections::addAll`, которая принимает список и добавляет все дополнительные аргументы в конец списка:

org/jgcbook/chapter01/B_generic_methods_and_varargs/Lists_3

```
public static <T> void addAll(List<T> list, T... arr) {
  for (T elt : arr) list.add(elt);
}
```

Методу с varargs-параметром можно передать либо список аргументов для неявной упаковки в массив, либо явно передать массив. Таким образом, метод `addAll` можно вызвать и так:

org/jgcbook/chapter01/B_generic_methods_and_varargs/Lists_3

```
List<Integer> ints = new ArrayList<Integer>();
Lists_3.addAll(ints, 1, 2);
Lists_3.addAll(ints, new Integer[] { 3, 4 });
assert ints.equals(List.of(1, 2, 3, 4));
```

Позже мы увидим, что при попытке создать массив, содержащий параметризованный тип, мы всегда получаем предупреждение о невозможности проверки. Поскольку varargs-аргументы всегда создают массив, их следует использовать с осторожностью, когда тип аргумента параметризован (см. раздел «Создание массива и varargs-аргументы» ниже).

В этих примерах параметр-тип обобщенного метода выводится, что иллюстрирует обычную ситуацию, когда один или несколько аргументов, соответствующих параметру-типу, все имеют один и тот же тип. Когда аргументов нет или аргументы принадлежат разным подтипам желаемого типа, параметр-тип, возможно, придется задать явно. Например:

org/jgcbook/chapter01/B_generic_methods_and_varargs/Lists_2

```
var ints = Lists_2.<Integer>toList();
var objs = Lists_2.<Object>toList(1, "two");
```

В первом случае если бы параметр-тип не был задан явно, то компилятор вывел бы тип `Object`. Во втором случае выведенный тип не только наследовал бы `Object`, но и должен был бы реализовать все интерфейсы, которые реализуют и `Integer`, и `String`, включая в числе прочих `Serializable` и `Comparable`. Это *тип-пересечение*, мы будем подробно рассматривать такие типы в разделе «Несколько границ» ниже.

Когда параметр-тип передается обобщенному методу, он указывается в угловых скобках слева, как в объявлении метода. Грамматика Java требует, чтобы параметры-типы встречались только в вызовах методов в форме с точкой. Даже если метод `toList` определен в том же классе, где вызывается, или импортирован как статический метод, мы не можем сократить его следующим образом:

```
List<Integer> ints = <Integer>toList(); // ошибка компиляции
```

Примитивные и ссылочные типы

И последняя тема, которую мы хотим рассмотреть в этой вступительной главе, – сравнение примитивных и ссылочных типов. *Ссылочным типом* является тип любого класса, интерфейса или массива, а *примитивные типы* перечислены в табл. 1.1. Различие между ними имеет фундаментальное значение для реализации параметризованных типов в Java, поскольку только ссылочные типы можно использовать в качестве параметров-типов, а примитивные типы в этой роли выступать не могут. Так, вместо `List<int>` мы обязаны писать `List<Integer>`[1]. Все ссылочные типы являются подтипами класса `Object`, и любой переменной ссылочного типа можно присвоить значение `null`. В табл. 1.1 для каждого из восьми примитивных типов указан соответствующий библиотечный класс ссылочного типа, находящийся в пакете `java.lang`.

Таблица 1.1. Примитивные и соответствующие им ссылочные типы

Примитивный	Ссылочный
`byte`	`Byte`
`short`	`Short`
`Int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Преобразование примитивного типа в соответствующий ссылочный тип называется *упаковкой* (boxing). Компилятор часто вставляет код, выполняющий это преобразование автоматически; это называется *автоупаковкой*. Обратное преобразование, когда из ссылочного значения извлекается соответствующий примитивный тип, называется *распаковкой* (unboxing).

Упаковка и распаковка применяются автоматически там, где это возможно и необходимо. Если выражение `e` типа `int` встречается там, где ожидается значение типа `Integer`, то оно упаковывается в `Integer.valueOf(e)`. Если выражение `e` типа `Integer` встречается там, где ожидается значение типа `int`, то оно распаковывается в `e.intValue()`. Например, последовательность:

org/jgcbook/chapter01/C_primitive_and_reference_types/Program_1

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
int n = ints.getFirst();
```

транслируется в код, эквивалентный последовательности, порожденной кодом:

[1] Это различие может исчезнуть в будущей версии Java, разрабатываемой в рамках проекта Вальгалла, но до этого нужно проделать еще много подготовительной работы.

org/jgcbook/chapter01/C_primitive_and_reference_types/Program_2

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

Вызов `Integer.valueOf(1)` возвращает экземпляр `Integer`, представляющий значение `1` типа `int`. Фабричный метод `Integer::valueOf` предпочтительнее конструктора `Integer::new`, который объявлен нерекомендуемым в версии Java 11 и запланирован для удаления в Java 16, потому что открывает возможность повторного использования кешированных объектов `Integer`[1]. На самом деле спецификация языка Java (Gosling et al. 2023, § 5.1.7) обязывает кешировать некоторые значения: требуется, чтобы любые два преобразования упаковки этих значений возвращали одну и ту же ссылку. Речь идет о числах типа `int` и `short` в диапазоне между `-128` и `127` включительно, символах типа `char` от `'\u0000'` до `'\u007f'`, а также значениях типа `byte` и `boolean`. Поэтому следующее утверждение всегда истинно:

org/jgcbook/chapter01/C_primitive_and_reference_types/Program_3

```
assert Integer.valueOf(5) == Integer.valueOf(5);
```

тогда как утверждение

org/jgcbook/chapter01/C_primitive_and_reference_types/Program_4

```
assert Integer.valueOf(500) != Integer.valueOf(500);
```

обычно истинно, но это необязательно – все зависит от политики кеширования, применяемой JVM.

Ниже приведен код, удобно оформленный в виде статического метода, который вычисляет сумму списка целых чисел:

org/jgcbook/chapter01/C_primitive_and_reference_types/Program_5

```
public static int sum(List<Integer> ints) {
  int s = 0;
  for (int n : ints) { s += n; }
  return s;
}
```

Почему аргумент здесь имеет тип `List<Integer>`, а не `List<int>`? Потому что параметры-типы всегда должны быть привязаны к ссылочным, а не примитивным типам. А почему метод возвращает значение типа `int`, а не `Integer`? Потому что результат может иметь как примитивный, так и ссылочный тип, но первый эффективнее. Всякий раз, как `Integer` в списке `int` привязывается к переменной `n` типа `int`, производится распаковка. Мы могли бы переписать метод, заменив все вхождения `int` на `Integer`:

org/jgcbook/chapter01/C_primitive_and_reference_types/Program_6

```
public static Integer sumInteger(List<Integer> ints) {
  Integer s = 0;
  for (Integer n : ints) { s += n; }
  return s;
}
```

[1] Это решение может быть в конечном итоге пересмотрено, также в рамках проекта Вальгалла. Такая свистопляска немного напрягает.

Этот код компилируется и правильно работает, но выполняет много лишних операций. На каждой итерации цикла распаковываются значения `s` и `n`, выполняется их сложение, а затем результат снова упаковывается. Если такой код встречается в приложении на критическом пути, где выполняется очень часто, то упаковка и распаковка могут значительно снизить производительность.

Существует одно значение типа `Integer`, которому не соответствует никакое `int`. Это значение `null`, являющее членом каждого ссылочного типа. Если любой версии метода суммирования передать список, содержащий `null`, то будет возбуждено исключение `NullPointerException`. Возьмем для примера второй вариант:

```
jshell> chapter01.C_primitive_and_reference_types.SumInteger.sumInteger(
   ...> Arrays.asList(1, 2, 3, null))
| Exception java.lang.NullPointerException: Cannot invoke \
"java.lang.Integer.intValue()" because "<local3>" is null
| at SumInteger.sumInteger (SumInteger.java:8)
| at (#3:1)
```

Заключение

В этой главе мы познакомились с основной идеей параметризованных типов в Java и уяснили их назначение: дать возможность использовать один и тот же код для объектов разных типов типобезопасным способом. После компиляции информация о параметризованном типе отбрасывается; этот процесс называется стиранием, и нам придется изучить его последствия во всех деталях.

В следующей главе мы продолжим рассмотрение взаимодействия параметризованных типов с объектно ориентированной полиморфной системой подтипизации в Java.

Глава 2

Подтипизация и джокеры

Познакомившись с основами, мы можем приступить к изучению более продвинутых возможностей параметризованных типов, а именно подтипизации и джокеров. В этой главе мы поговорим о том, как работает подтипизация, и увидим, как джокеры позволяют соединить подтипизацию с параметризованными типами. Источником примеров нам послужит каркас коллекций Java; о конкретных деталях API коллекций рассказано в части II.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter02*.

Подтипизация и принцип подстановки

Подтипизация – важнейшее свойство объектно ориентированных языков и Java в том числе. В Java один тип является *подтипом* другого, если они связаны фразой `extends` или `implements`. Вот несколько примеров:

```
Integer       является подтипом  Number
Double                           является подтипом
Number
ArrayList<E>  является подтипом  List<E>
List<E>       является подтипом  Collection<E>
Collection<E> является подтипом  Iterable<E>
```

Отношение подтипизации транзитивно. Это означает, что если один тип является подтипом второго, а второй подтипом третьего, то первый тип также является подтипом третьего. Поэтому из двух последних строк в списке выше следует, что `List<E>` является подтипом `Iterable<E>`. Если один тип является подтипом другого, то говорят также, что второй тип является *супертипом* первого. Отношение подтипизации рефлексивно, т. е. любой тип является подтипом – а значит, и супертипом – самого себя. Любой ссылочный тип является подтипом `Object`, а `Object` является супертипом любого ссылочного типа.

Самое важное свойство подтипизации выражает принцип подстановки[1]:

[1] В работе Liskov (1987) предложено следующее определение поведенческой подтипизации: подстановка значений одного типа вместо значений другого не изменяет поведения программы. Применение этого определения в обратном направлении приводит к принципу проектирования, который часто называют принципом подстановки Лисков. Его применение к каркасу коллекций мы обсудим в главе 18.

принцип подстановки: всюду, где ожидает значение типа `T`, можно подставить значение подтипа `T`.

Это означает, что переменной заданного типа можно присвоить значение любого его подтипа; например, переменной типа `Number` можно присвоить значение типа `Integer` или `Double`. Аналогично методу с параметром заданного типа можно передать аргумент любого его подтипа.

Рассмотрим интерфейс `Collection<E>`. В нем имеется метод `add`, который принимает параметр типа `E`:

```
interface Collection<E> {
  ...
  boolean add(E elt);
  ...
}
```

Согласно принципу подстановки, в коллекцию чисел можно добавлять как целые числа, так и числа с двойной точностью, потому что `Integer` и `Double` являются подтипами `Number`:

org/jgcbook/chapter02/A_subtyping_and_the_substitution_principle/Program_1

```
List<Number> nums = new ArrayList<>();
nums.add(2);
nums.add(0.25);
assert nums.equals(List.of(2, 0.25));
```

Здесь подтипизация используется двумя способами. Первый вызов допустим, потому что `nums` имеет тип `List<Number>`, являющийся подтипом `Collection<Number>`, а `2` имеет тип `Integer` (благодаря автоупаковке), являющийся подтипом `Number`. Второй вызов также допустим: значение `0.25` упаковано в тип `Double`, также являющийся подтипом `Number`. В обоих случаях в качестве типа `E` в `List<E>` принимается `Number`.

Кажется разумным ожидать, что коль скоро `Integer` является подтипом `Number`, то и `List<Integer>` будет подтипом `List<Number>`. Но это *не* так, потому что иначе мы тут же вступили бы в конфликт с принципом подстановки. Чтобы понять, почему небезопасно присваивать значение типа `List<Integer>` переменной типа `List<Number>`, рассмотрим следующий код:

org/jgcbook/chapter02/A_subtyping_and_the_substitution_principle/Program_2

```
List<Integer> ints = new ArrayList<>();
List<Number> nums = ints;  // этого разрешать нельзя
nums.add(3.14);
```

Здесь переменной `ints` присваивается указатель на список целых, а затем переменной `nums` присваивается указатель на *тот же самый* список. Если бы мы потребовали, чтобы принцип подстановки выполнялся, то вызов в следующей строчке – безусловно, корректный – добавил бы в этот список `Double`, а значит, переменная `ints`, имеющая тип `List<Integer>`, указывала бы на список, содержащий `Double`, и система типов Java оказалась бы нарушенной. Очевидно, что этого допустить нельзя, а следовательно, такое присваивание должно быть вне закона. Если `List<Integer>` не является подтипом `List<Number>`, то принцип

подстановки неприменим, и о втором предложении присваивания компилятор может сообщить как об ошибке.

Можно ли считать, что `List<Number>` является подтипом `List<Integer>`? Тоже не получится, как показывает следующий код:

org/jgcbook/chapter02/A_subtyping_and_the_substitution_principle/Program_3

```
List<Number> nums = new ArrayList<>();
nums.add(3.14);
List<Integer> ints = nums;  // этого разрешать нельзя
```

Если бы мы здесь разрешили второе присваивание, то получили бы ту же самую проблему – переменная, объявленная как `List<Integer>`, указывает на список, содержащий `Double`. Итак, `List<Number>` не является подтипом `List<Integer>`, а поскольку мы уже видели, что и `List<Integer>` не является подтипом `List<Number>`, то остается лишь тривиальный случай – тип `List<Integer>` является подтипом самого себя. (Также имеет место интуитивно более понятное отношение: `List<Integer>` является подтипом `Collection<Integer>`.)

Массивы ведут себя совершенно иначе: `Integer[]` *является* подтипом `Number[]`. Мы сравним обработку массивов и списков позже (см. раздел «Массивы» ниже).

Иногда мы хотели бы, чтобы списки вели себя в большей степени как массивы, т. е. хотелось бы принимать не только список элементов данного типа, но и список элементов любого его подтипа. Для этой цели используются *джокеры* (wildcard).

Джокеры

В интерфейсе `Collection` имеется также метод `containsAll`, который проверяет, что каждый член переданной ему коллекции принадлежит той, от имени которой он вызван:

```
interface Collection<E> {
  ...
  public boolean containsAll(Collection<?> c);
  ...
}
```

Символ `?` является джокером, он обозначает некоторый *неизвестный, но фиксированный* тип. При вызове `containsAll` можно было бы передать `Collection<String>`, `Collection<Number>` или коллекцию любого другого типа (в том числе `Collection<Object>`). Если мы можем передать `Collection<String>`, то, согласно принципу подстановки, можем также передать `List<String>` или `Set<String>`. Возможно, вам кажется странным, почему определение `containsAll` позволяет проверять, является ли каждый член `Collection<String>` также членом `Collection<Number>`. Мы изучим этот вопрос позже, в разделе «Сравнение Object и E» главы 18.

Если дано значение типа `Collection<?>`, то что можно сказать о типе его элементов на этапе компиляции? Ответ «очень мало»: определение джокера говорит, что это неизвестный тип, так что о нем можно сказать лишь то, что мы знаем о любом типе Java: это подтип `Object`. Например, это означает, что в `Collection<?>` нельзя поместить элемент любого типа (кроме `null`). Чтобы это сделать, не нарушая согласованность типов, нужно было бы знать тип элемен-

та, но это как раз то, что *неограниченный джокер*, т. е. `?`, сообщить не может. *Ограниченные джокеры* предоставляют больше информации. Есть два вида таких джокеров: с `extends` и с `super`.

Джокеры с extends

Еще один метод интерфейса `Collection`, `addAll`, добавляет все элементы одной коллекции в другую:

```
interface Collection<E> {
  ...
  public boolean addAll(Collection<? extends E> c);
  ...
}
```

Очевидно, что если дана коллекция элементов типа `E`, то мы должны иметь возможность добавить все члены другой коллекции с элементами типа `E`. Например, если дана коллекция `Collection<Number>`, то должна быть возможность добавить в нее все элементы другой коллекции `Number`. Также было бы безопасно добавить все элементы коллекции `Integer` или `Long`, потому что значения этих типов являются также значениями типа `Number`. Но добавлять элементы `Collection<Object>` или `Collection<String>` должно быть запрещено, потому что они могут не быть (а во втором случае точно не являются) экземплярами `Number`. Поэтому для определения типа коллекции, которая может выступать в роли аргумента `addAll`, нам необходим способ выразить следующее ограничение: тип элемента должен быть подтипом `Number`. Именно для этого предназначен *ограниченный джокер* `? extends E`: как и неограниченный джокер `?`, он обозначает фиксированный тип, но кое-что про него мы знаем, а именно что он должен быть подтипом `E`. Про такой джокер говорят, что он *ограничен сверху* (см. рис. 2.1).

Метод `addAll` ожидает получить коллекцию элементов некоторого подтипа `E`. Какие типы можно ему передать? В качестве конкретного примера возьмем метод, объявленный с параметром типа `Collection<? extends Number>`: он мог бы принять, скажем, `Collection<Integer>` или `Collection<Double>`. Так что они являются подтипами `Collection<? extends Number>`. Вообще, если `F` является подтипом `E`, то `Collection<F>` является подтипом `Collection<? extends E>`.

Рассмотрим пример. Мы создаем пустой список чисел и сначала добавляем в него список целых, а затем список чисел с двойной точностью:

org/jgcbook/chapter02/C_wildcards_with_extends/Program_1

```
List<Number> nums = new ArrayList<>();
List<Integer> ints = List.of(1, 2);
List<Double> dbls = List.of(1.0, 0.5);
nums.addAll(ints);
nums.addAll(dbls);
assert nums.equals(List.of(1, 2, 1.0, 0.5));
```

Первый вызов разрешен, потому что `nums` имеет тип `List<Number>`, являющийся подтипом `Collection<Number>`, а `ints` имеет тип `List<Integer>`, являющийся подтипом `Collection<Integer>`, который является подтипом `Collection<? extends Number>`. Второй вызов разрешен по тем же причинам. В обоих случаях в каче-

стве E принимается Number. Если бы объявление addAll было написано без джокера, то вызовы для добавления списков целых и чисел с двойной точностью в список чисел были бы запрещены; мы смогли бы добавить только список, явно объявленный как список Number.

Джокеры можно использовать при объявлении переменных. Ниже приведен вариант вступительного примера из предыдущего раздела, отличающийся тем, что во вторую строчку добавлен джокер:

org/jgcbook/chapter02/C_wildcards_with_extends/Program_2

```
List<Integer> ints = new ArrayList<>();
List<? extends Number> nums = ints; // теперь допустимо
nums.add(3.14);                     // этого разрешать нельзя
```

Раньше второе присваивание считалось ошибкой компиляции (потому что List<Integer> не является подтипом List<Number>), но вызов add не вызывал возражений (потому что число с двойной точностью – это число, так что его можно добавить в List<Number>). Теперь со вторым присваиванием все в порядке (потому что List<Integer> является подтипом List<? extends Number>), но вызов add приводит к ошибке компиляции. Добавить число с двойной точностью в List<? extends Number> нельзя, так как мы не знаем, какой тип представлен джокером; известно лишь, что это какой-то подтип Number. Если бы такой код был разрешен, как раньше, то из-за последней строчки переменная ints, объявленная как List<Integer>, указывала бы на список, содержащий число с двойной точностью.

Поскольку в Java любой тип является подтипом Object, термин «неограниченный» не вполне правилен: на самом деле Collection<?> – это сокращенная запись Collection<? extends Object>. Но этот частный случай настолько важен и используется так часто, что термин «неограниченный» и его сокращенная запись употребляются всеми.

В общем случае если структура содержит элементы, имеющие тип вида ? extends E, то мы можем извлекать из нее элементы, но не можем ничего добавлять. Для помещения элементов в структуру нам необходим другой вид джокера, о котором мы поговорим в следующем разделе.

Джокеры с super

Ниже приведен метод из служебного класса Collections, который копирует элементы из исходного списка в конечный.

org/jgcbook/chapter02/D_wildcards_with_super/Program_1

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
 for (int i = 0; i < src.size(); i++) {
    dest.set(i, src.get(i));
  }
}
```

Фраза ? super T означает, что конечный список может содержать элементы любого типа, являющегося *супертипом* T, тогда как исходный список может содержать элементы любого типа, являющегося *подтипом* T. Про такой джокер говорят, что он *ограничен снизу* (см. рис. 2.1).

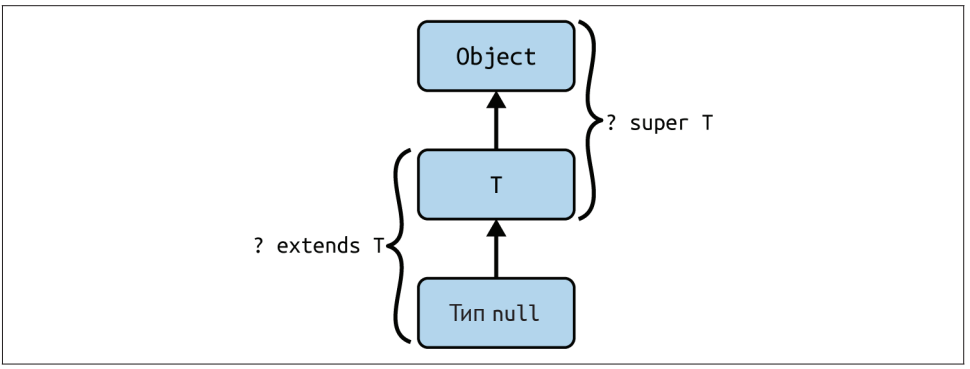


Рис. 2.1. Ограниченные типы: `extends` и `super`

Вот пример вызова:

org/jgcbook/chapter02/D_wildcards_with_super/Program_1

```
List<Object> objs = Stream.of(2, 3.14, "four")
   .collect(Collectors.toCollection(ArrayList::new));
List<Integer> ints = List.of(5, 6);
Collections.copy(objs, ints);
assert objs.equals(List.of(5, 6, "four"));
```

Как и для любого обобщенного метода, параметр-тип можно вывести или указать явно. В данном случае возможно четыре варианта, причем во всех типы проверяются и все дают одинаковый результат.

org/jgcbook/chapter02/D_wildcards_with_super/Program_1

```
Collections.copy(objs, ints);
Collections.<Object>copy(objs, ints);
Collections.<Number>copy(objs, ints);
Collections.<Integer>copy(objs, ints);
```

В первом случае параметр-тип неявный, он выводится как `Integer`, потому что это самый узкий из возможных типов. В третьем случае явно задан тип `Number`. Вызов разрешен, потому что `objs` имеет тип `List<Object>`, являющийся подтипом `List<?super Number>` (так как `Object` является супертипом `Number`, как того требует джокер с `super`), а `ints` имеет тип `List<Integer>`, являющийся подтипом `List<? extends Number>` (так как `Integer` является подтипом `Number`, как того требует джокер с `extends`).

Мы могли бы также объявить этот метод с несколькими возможными *сигнатурами* (сигнатурой метода называется комбинация идентификатора метода, его параметров-типов, а также типов и порядка его параметров)[1]:

```
public static <T> void copy(List<T> dst, List<T> src)
public static <T> void copy(List<T> dst, List<? extends T> src)
public static <T> void copy(List<? super T> dst, List<T> src)
public static <T> void copy(List<? super T> dst, List<? extends T> src)
```

[1] Чтобы параметры-типы различали две сигнатуры метода, должно различаться количество параметров или их ограниченность; одного лишь изменения имени недостаточно.

Первая сигнатура слишком ограничительна, так как допускает только вызовы, в которых источник и приемник имеют в точности одинаковые типы. Остальные три эквивалентны для вызовов с неявными параметрами-типами, но различаются, когда параметры-типы заданы явно. В примерах вызовов, показанных выше, вторая сигнатура работает, только когда параметр-тип равен `Object`, третья – когда параметр-тип равен `Integer`, а последняя (как мы видели) – для всех трех параметров-типов: `Object`, `Number` и `Integer`. Записывая сигнатуру метода, всюду, где возможно, используйте джокеры, так как это обеспечивает самый широкий диапазон вызовов.

Принцип получения и вставки

Конечно, вставлять джокеры всюду, где возможно, – идея хорошая, но как решить, *какой* джокер использовать? Где нужен `extends`, где `super`, а где лучше обойтись вовсе без джокера?

По счастью, для определения того, что где подходит, есть простой принцип:

> *принцип получения и вставки*: используйте джокер с `extends`, когда нужно только *получать* значения из структуры; используйте джокер с `super`, когда нужно только *вставлять* значения в структуру, и не используйте джокер вообще, когда нужно и получать, и вставлять.

В книге Joshua Bloch «Effective Java» (2017, совет 31) этому принципу дано мнемоническое название PECS (producer-`extends`, consumer-`super`). Мы уже видели его в деле на примере сигнатуры метода `copy`:

```
public static <T> void copy(List<? super T> dst, List<? extends T> src)
```

Метод получает значения из исходного списка `src`, поэтому в объявлении указан джокер с `extends`, и вставляет значение в конечный список `dst`, поэтому в объявлении указан джокер с `super`.

Используя итератор или поток, вы получаете значения из структуры, поэтому должны использовать джокер с `extends`. Ниже показана итераторная версия метода, который принимает коллекцию чисел, преобразует их в числа с двойной точностью и вычисляет сумму.

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_1

```
public static double sum(Collection<? extends Number> nums) {
  double s = 0.0;
  for (Number num : nums) s += num.doubleValue();
  return s;
}
```

В эквивалентной потоковой версии переменная типа `Number` явно не объявлена, но ограничение на тип аргумента точно такое же:

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_2

```
public static double sum(Collection<? extends Number> nums) {
return nums.stream().mapToDouble(Number::doubleValue).sum();
}
```

Поскольку в этих методах используется `extends`, все показанные ниже вызовы допустимы:

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_2

```
List<Integer> ints = List.of(1, 2, 3);
assert sum(ints) == 6.0;

List<Double> doubles = List.of(2.5, 3.5);
assert sum(doubles) == 6.0;

List<Number> nums = List.of(1, 2, 2.5, 3.5);
assert sum(nums) == 9.0;
```

Первые два вызова были бы недопустимы без `extends`.

При любом использовании метода типа `add`, который вставляет значения в структуру, следует употреблять джокер с `super`. Показанный ниже метод принимает коллекцию элементов какого-то супертипа `Integer` и целое число `n` и вставляет в коллекцию первые `n` целых, начиная с нуля:

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_3

```
public static void storeIntegers(Collection<? super Integer> ints, int n) {
  for (int i = 0; i < n; i++) ints.add(i);
}
```

У этого метода нет простого потокового эквивалента[1]. Поскольку в объявлении параметра метода использован джокер с `super`, следующие вызовы допустимы:

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_3

```
List<Integer> ints1 = new ArrayList<>();
storeIntegers(ints1, 5);
assert ints1.equals(List.of(0, 1, 2, 3, 4));

List<Number> nums1 = new ArrayList<>();
storeIntegers(nums1, 5);
nums1.add(5.0);
assert nums1.equals(List.of(0, 1, 2, 3, 4, 5.0));

List<Object> objs1 = new ArrayList<>();
storeIntegers(objs1, 5); objs1.add("five");
assert objs1.equals(List.of(0, 1, 2, 3, 4, "five"));
```

Последние два вызова были бы недопустимы без `super`.

Если вы хотите и вставлять значения в структуру, и получать из нее, как в примере ниже, то использовать джокер не следует.

---

[1] Возможно, у вас возникла мысль написать в качестве простого эквивалента `IntStream.range(0,n).forEach(ints::add);` – но не поддавайтесь этому искушению! При таком использовании `Stream::forEach` не будет работать в условиях конкурентного доступа. Даже если вы уверены, что сейчас никаких проблем с конкурентностью нет, в будущем ваш код может быть подвергнут рефакторингу, вносящему конкурентность, например если кто-то захочет итеративно обойти целевую коллекцию. Использование коллектора защитит ваш код от ошибок при будущих изменениях и станет инвестицией в улучшение своего стиля кодирования. Более подробно о причинах см. работу Naftalin, 2015.

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_4

```
public static double sumValues(Collection<Number> nums, int n) {
  storeIntegers(nums, n);
  return sum(nums);
}
```

Коллекция передается методам `sum` и `storeIntegers`, поэтому тип ее элементов должен одновременно расширять `Number` (этого требует `sum`) и быть супертипом `Integer` (этого требует `storeIntegers`). Только два класса удовлетворяют обоим этим ограничениям – `Number` и `Integer`; мы выбрали первый из них.

Вот пример вызова:

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_4

```
List<Number> nums2 = new ArrayList<>();
double sum = sumValues(nums2, 5);
assert sum == 10;
```

Поскольку джокера нет, аргумент должен быть коллекцией `Number`.

Если вам не нравится выбирать между `Number` и `Integer`, то, возможно, вам пришло в голову, что если бы Java позволял написать джокер, включающий одновременно и `extends`, и `super`, то и выбирать бы не пришлось. Например, что, если написать:

```
double sumValues(Collection<? extends Number super Integer> coll, int n)
// недопустимо в Java!
```

Тогда мы могли бы вызвать `sumValues` как для коллекции чисел, так и для коллекции целых. Но Java этого *не* разрешает. Единственная причина – стремление сохранить простоту реализации языка, поэтому вполне возможно, что в будущем такая нотация может быть узаконена. Ну а пока, если вам нужно и получение, и вставка, то не используйте джокеры.

Принцип получения и вставки работает и в обратном направлении. Если имеется джокер с `extends`, то практически единственное, что вы можете делать, – получать значения такого типа, но не вставлять их. А если присутствует джокер с `super`, то вы можете вставлять значения такого типа, но не получать их. Рассмотрим, к примеру, следующий код, в котором используется список, объявленный с джокером с `extends`:

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_5

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
double dbl = sum(nums); // ok
nums.add(3.14);         // ошибка компиляции
```

Вызов `sum` корректен, потому что он получает значения из списка, но вызов `add` запрещен, потому что он вставляет значения в список. И это правильно, потому что иначе мы могли бы вставить число с плавающей точкой в список целых!

Обратно, рассмотрим код, в котором в объявлении списка присутствует джокер с `super`:

```
List<Object> objs = new ArrayList<>();
```

```
objs.add(1);
objs.add("two");
List<? super Integer> ints = objs;
ints.add(3);            // ok
double dbl = sum(ints); // ошибка компиляции
ints.add("three");      // ошибка компиляции
```

Теперь первый вызов `ints.add` корректен, потому что он вставляет значение в список, но вызов `sum` недопустим, потому что получает значение из списка. И снова это хорошо, потому что сумма элементов списка, содержащего строки, не имеет смысла! Последний вызов `add` тоже не компилируется, потому что, хотя `ints` ссылается на `ArrayList<Object>`, важен только тип на этапе компиляции, а принцип получения и вставки говорит, что в списковую переменную с типом на этапе компиляции `List<? super Integer>` можно вставить только `Integer` (строго говоря, подтип `Integer`).

Возможно, вам будет проще рассматривать конструкцию `? extends T` как неизвестный тип в интервале, ограниченном типом `null` снизу и типом `T` сверху (где тип `null` является подтипом любого ссылочного типа). Аналогично `? super T` можно рассматривать как неизвестный тип в интервале, ограниченном типом `T` снизу и типом `Object` сверху (см. рис. 2.1).

Значение, вставляемое в любой из этих интервалов, должно принадлежать нижнему типу. Поэтому единственное значение, которое можно вставить в тип, объявленный с джокером с `extends`, – это `null`, поскольку только это значение принадлежит любому ссылочному типу.

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_7

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(null);   // ok
assert nums.equals(Arrays.asList(1, 2, null));
```

А значения, которые можно получить из любого из этих интервалов, должны принадлежать верхнему типу. Это значит, что единственные значения, которые можно получить из типа, объявленного с джокером с `super`, – это значения типа `Object`, являющегося супертипом любого ссылочного типа:

org/jgcbook/chapter02/E_the_get_and_put_principle/Program_8

```
List<Object> objs = List.of(1,"two");
List<? super Integer> ints = objs;
String str = "";
for (Object obj : ints) str += obj.toString();
assert str.equals("1two");
```

Возникает соблазн подумать, что джокер с `extends` гарантирует неизменяемость или, по крайней мере, немодифицируемость, но это не так. Как мы видели, даже в список типа `List<? extends Number>` можно вставлять значения `null`. Можно также удалять элементы из списка (методами `remove`, `removeAll`, `retainAll` или `clear`) и менять порядок элементов списка (с помощью методов `swap`, `sort` или `shuffle` служебного класса `Collections`; см. раздел «Изменение порядка элементов списка» главы 16). Проблемы и преимущества достижения немодифи-

цируемости и неизменяемости исследуются в разделе «Неизменяемость и немодифицируемость» главы 9 и далее в главе 18.

Поскольку `String` – финальный тип, не имеющий подтипов, можно было бы ожидать, что `List<String>` – тот же тип, что `List<? extends String>`. Но на самом деле первый является подтипом второго, а не тем же самым типом, в чем легко убедиться, применив наши принципы. Принцип подстановки говорит, что это подтип, потому что разрешается передавать значение первого типа туда, где ожидается значение второго. Принцип получения и вставки говорит, что это не тот же самый тип, потому что мы можем вставить строку в экземпляр первого, но не второго типа.

Массивы

Поучительно сравнить, как Java обращается с массивами и со списками, памятуя о принципах подстановки и получения и вставки.

В Java подтипизация массивов *ковариантна*, т. е. тип `S[]` является подтипом `T[]`, если `S` – подтип `T`. Рассмотрим следующий фрагмент кода, где выделяется память для массива целых, он присваивается переменной, объявленной как массив `Number`, а затем производится попытка сохранить в массиве число с плавающей точкой:

org/jgcbook/chapter02/F_arrays/Program_1

```
Integer[] ints = {0};
Number[] nums = ints;
nums[0] = 3.14; // исключение при сохранении в массиве (подтип RuntimeExcep-
tion)
```

С этой программой что-то не так, потому что если бы она откомпилировалась успешно, то переменная `ints`, объявленная как `Integer[]`, указывала бы на массив, содержащий `Double`, и система типов Java оказалась бы нарушенной. В чем же проблема? Поскольку `Integer[]` – подтип `Number[]`, то, согласно принципу подстановки, присваивание во второй строчке должно быть допустимо. Вместо этого проблема обнаруживается в третьей строчке. Когда для массива выделяется память (как в первой строчке), она помечается его сбереженным типом (представление типа компоненты на этапе выполнения, в данном случае `Integer`), и при сохранении значения в массиве (как в третьей строчке) возникает исключение, если сбереженный тип не совместим с типом значения (в данном случае `Double` нельзя сохранить в массиве целых).

С другой стороны, отношение подтипизации для параметризованных типов *инвариантно*, т. е. тип `List<S>` *не* является подтипом `List<T>`, кроме тривиального случая, когда `S` и `T` совпадают. Следующий фрагмент аналогичен предыдущему, только массивы заменены списками.

org/jgcbook/chapter02/F_arrays/Program_2

```
List<Integer> ints = Arrays.asList(0);
List<Number> nums = ints; // ошибка компиляции
nums.set(0, 3.14);
```

Поскольку `List<Integer>` не является подтипом `List<Number>`, проблема обнаруживается во второй, а не в третьей строчке и на этапе компиляции, а не выполнения.

Джокеры восстанавливают ковариантную подтипизацию для параметризованных типов в том смысле, что `List<S>` *является* подтипом `List<? extends T>`, когда `S` является подтипом `T`. Ниже показан третий вариант того же фрагмента.

org/jgcbook/chapter02/F_arrays/Program_3

```
List<Integer> ints = Arrays.asList(0);
List<? extends Number> nums = ints;
nums.set(0, 3.14); // ошибка компиляции
```

Как и в случае массивов, ошибка присутствует в третьей строчке, но теперь она обнаруживается на этапе компиляции, а не выполнения. Присваивание нарушает принцип получения и вставки, потому что в тип, объявленный с джокером с `extends` нельзя вставлять значения.

Джокеры также приносят с собой *контравариантную* подтипизацию для параметризованных типов, а именно: тип `List<S>` является *подтипом* `List<? super T>`, если `S` – *супертип* `T` (а не подтип). Массивы не поддерживают контравариантную подтипизацию. Например, вспомним метод `storeIntegers`:

org/jgcbook/chapter02/F_arrays/Program_4

```
public static void storeIntegers(Collection<? super Integer> ints, int n) {
  for (int i = 0; i < n; i++) ints.add(i);
}
```

Он принимает параметр типа `Collection<? super Integer>` и заполняет его целыми числами. Эквивалентного способа проделать то же самое с массивами не существует, потому что Java запрещает конструкцию `(? super Integer)[]`.

Обнаружение проблем на этапе компиляции, а не выполнения имеет два преимущества: большое и малое. Малое преимущество – повышение эффективности. Системе не нужно тащить описание типа элемента вплоть до этапа выполнения и проверять это описание при каждой записи в массив. Большое же преимущество заключается в том, что целое семейство распространенных ошибок обнаруживается компилятором. Это улучшает все стадии жизненного цикла программы: кодирование, отладка, тестирование и сопровождение становятся проще, быстрее и дешевле. Но и помимо более раннего обнаружения ошибок типизации, есть много других причин предпочесть классы коллекций массивам.

- Коллекции обладают большей гибкостью; массивы поддерживают только две операции: чтение и установку элемента, а число элементов массива фиксировано.
- Коллекции поддерживают много дополнительных операций, включая проверку принадлежности, добавление и удаление элементов и объединение двух коллекций.
- Коллекции могут быть списками (порядок имеет значение, одно и то же значение может встречаться несколько раз, имеются другие операции), множествами (порядок несуществен, элементы не могут повторяться) или очередями. Для каждого из этих типов данных возможны различные представления, в том числе массивы, связанные списки, деревья и хеш-таблицы.
- Для ситуаций, когда необходим эффективный конкурентный доступ, имеются специализированные коллекции.

- Наконец, хотя это и не есть внутренне присущее преимущество коллекций перед массивами, служебный класс `Collections` предлагает такие операции, как циклический сдвиг и тасование списка, поиск максимального элемента, преобразование коллекции в немодифицируемую или синхронизированную и др. – гораздо больше, чем имеется в соответствующем служебном классе `Arrays`.

Тем не менее в некоторых ситуациях массивы предпочтительнее коллекций. В частности, массивы примитивных типов гораздо эффективнее, чем массивы или коллекции ссылочных типов, поскольку не требуют упаковки, потребляют меньше памяти и обладают гораздо лучшей пространственной локальностью (см. раздел «Производительность» главы 9). Кроме того, присваивание элементам примитивного массива не требует проверки возможности сохранения в массиве, потому что у массива примитивных типов нет подтипов. Эти преимущества могут побудить вас заменить коллекции массивами в тех частях программы, где необходима максимальная производительность. Как обычно, для обоснования такого дизайна необходимо проводить сравнительное тестирование производительности, не забывая о том, что будущие усовершенствования компилятора и библиотек могут изменить баланс. Наконец, в некоторых случаях массивы могут оказаться предпочтительнее из соображений совместимости.

Короче говоря, обнаруживать ошибки на этапе компиляции лучше, чем на этапе выполнения, но чтобы обеспечить ковариантую подтипизацию для массивов, в Java было принято решение обнаруживать некоторые ошибки работы с массивами на этапе выполнения. Было ли это решение разумным? До появления параметризованных типов оно было абсолютно необходимым. Например, взгляните на следующие методы, которые применяются для сортировки произвольного массива и для заполнения массива заданным значением:

```
public static void sort(Object[] a);
public static void fill(Object[] a, Object val);
```

Благодаря ковариантности эти методы можно использовать для сортировки или заполнения массивов любого ссылочного типа. Без ковариантности и без параметризованных типов было бы невозможно объявить методы, применимые ко всем типам. Но с появлением параметризованных типов необходимость в ковариантных массивах отпала, поскольку эти методы теперь могли бы иметь следующие сигнатуры, прямо указывающие, что они работают для всех типов:

```
public static <T> void sort(T[] a);
public static <T> void fill(T[] a, T val);
```

В каком-то смысле ковариантные массивы – пережиток отсутствия параметризованных типов в ранних версиях Java. Теперь, когда параметризованные типы имеются, ковариантные массивы, пожалуй, следует счесть ошибочным проектным решением, и единственная причина сохранить их – обратная совместимость. Различающиеся системы типов для параметризованных типов и массивов приводят к разного рода неудобствам взаимодействия, которые мы подробно рассмотрим в главе 5.

Ограниченные или неограниченные?

Выше в этой главе мы видели метод `containsAll` интерфейса `Collection`, который проверяет, включает ли данная коллекция все элементы другой коллекции. Это обобщение другого метода `Collection`, `contains`, который проверяет, содержит ли коллекция один заданный объект. В этом разделе описываются два альтернативных подхода к записи параметризованных сигнатур этих методов. В первом используются неограниченные джокеры, именно он применяется в каркасе коллекций Java. Во втором, более точном, используются джокеры, ограниченные параметром-типом интерфейса.

Использование неограниченного джокера

Ниже показаны типы метода `contains` в Java с параметрическими типами:

```
interface Collection<E> {
  ...
  public boolean contains(Object o);
  public boolean containsAll(Collection<?> c);
  ...
}
```

Параметр второго метода не противоречит первому, потому что неограниченный джокер `?` – не что иное, как сокращение `? extends Object`.

Эти методы позволяют проверять членство и включение.

org/jgcbook/chapter02/G_wildcards_versus_type_parameters/Program_1

```
Object obj = "one";
List<Object> objs = List.of("one", 2, 3.5, 4);
List<Integer> ints = List.of(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert ! ints.contains(obj);
assert ! ints.containsAll(objs);
```

Заданный список объектов содержит строку `"one"` и список целых, но заданный список целых не содержит ни строки `"one"`, ни списка объектов.

Проверки `ints.contains(obj)` и `ints.containsAll(objs)` могут показаться глупыми. Разумеется, список целых не будет содержать произвольный объект, например строку `"one"`. Но они разрешены, потому что иногда такие проверки проходят.

org/jgcbook/chapter02/G_wildcards_versus_type_parameters/Program_2

```
Object obj = 1;
List<Object> objs = List.of(1, 3);
List<Integer> ints = List.of(1, 2, 3, 4);
assert ints.contains(obj);
assert ints.containsAll(objs);
```

В этом случае объект может содержаться в списке целых, потому что в действительности является целым, а список объектов может содержаться в списке целых, потому что каждый элемент списка в действительности является целым.

В разделе «Сравнение Object и E» главы 18 мы подробно рассмотрим причины, по которым проектировщики параметризованных типов выбрали именно такие типы параметров для `contains` и `containsAll`, равно как и для других методов в интерфейсах `Collection`, `List` и `Map`, которые проверяют или удаляют значения из коллекций. Мы приведем аргументы за и против такого выбора.

Использование ограниченного джокера

Можно было бы выбрать альтернативный разумный дизайн коллекций – когда проверять на включение можно только подтипы типа элементов:

```
interface MyCollection<E> { // альтернативный дизайн
  ...
  public boolean contains(E o);
  public boolean containsAll(Collection<? extends E> c);
  ...
}
```

Пусть имеется класс `MyList`, реализующий интерфейс `MyCollection`. Теперь для проверок допустимо только одно направление:

```
Object obj = "one";
MyList<Object> objs = MyList.of("one", 2, 3.5, 4);
MyList<Integer> ints = MyList.of(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert ! ints.contains(obj);      // ошибка компиляции
assert ! ints.containsAll(objs);  // ошибка компиляции
```

Последние две проверки недопустимы, потому что из объявления типа с непреложностью следует, что проверять можно только, содержит ли список элементы, принадлежащие подтипу параметра-типа этого списка. То есть можно проверить, является ли список целых частью списка объектов, но не наоборот.

Запоминание джокера

У джокеров есть аспект, который на первый взгляд может показаться странным. Обычно безопасно предполагать, что если два типа записаны одинаково, то это один и тот же тип. Это так для констант-типов наподобие `Integer` и для переменных-типов наподобие `E` из предыдущего раздела. Но для типов с джокерами это уже не так. Рассмотрим пример:

```
List<? extends Number> list1;
List<? extends Number> list2;
```

Мы знаем, что любое вхождение `? extends Number` обозначает какой-то неизвестный подтип `Number`. Так, например, переменной `list1` можно было бы присвоить значение типа `List<Integer>`, а переменной `list2` – значение типа `List<Double>`. Оба вхождения `? extends Number` представляют потенциально *различные* типы, поэтому присваивание

```
list1.add(list2.get(0));
```

приводит к следующему сообщению в JShell для версии Java 21:

```
incompatible types: java.lang.Number cannot be converted \
to capture#1 of ? extends java.lang.Number
```

Для выражения `list2.get(0)` выведен тип `Number`, потому что это наиболее точная информация из имеющейся. Но теперь Java не может добавить значение этого типа в `list1`, потому что типом элементов `list1` является некий неизвестный подтип `Number`. Именно об этом и говорит сообщение. Но почему неизвестный тип назван `capture#1 of ? extends java.lang.Number`?

Дело в том, что в ходе нормального процесса компиляции компилятор выводит тип для каждого джокера: эта процедура, называемая *запоминанием джокера* (wildcard capture), обычно не видна программисту. Но в таких случаях, как этот, компилятор сообщает внутреннее имя, которое он присвоил типу, чтобы вы понимали, в чем состоит проблема несовместимости типов.

Более содержательный пример дает метод `reverse` из класса `Collections`, который принимает список любого типа и меняет порядок его элементов на противоположный. Объявлен он следующим образом:

```
public static void reverse(List<?> list);
```

Как могла бы выглядеть наивная попытка реализации? Скопировать аргумент во временный список, а затем записать копию в исходный список в обратном порядке:

org/jgcbook/chapter02/H_wildcard_capture/Program_1

```
public static void reverse(List<?> list) {
  List<Object> tmp = new ArrayList<>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1)); // ошибка компиляции
  }
}
```

Это не проходит, потому что записать копию обратно в оригинал нельзя: мы пытаемся копировать элементы из списка объектов в список неизвестного типа. При компиляции с помощью OpenJDK 21 этой версии `reverse` выдается следующее сообщение об ошибке (чуть более понятный вариант показанного выше сообщения JShell):

```
Capture.java:8: error: incompatible types: Object cannot be converted to CAP#1
            list.set(i, tmp.get(list.size()-i-1)); // compile-time error
                                ^
where CAP#1 is a fresh type-variable: CAP#1 extends Object from capture of ?
```

`CAP#1` (сокращение «capture#1») – имя, присвоенное компилятором типу элементов `list`. Если джокеров несколько, то каждый представляет свой неизвестный тип, поэтому им будут присвоены разные имена, даже если с каждым ассоциирован один и тот же тип, – например, `capture of ?` или, в случае ограниченного джокера, `capture of ? extends Number`.

В данном случае сообщение только раздражает, а не приносит пользу, поскольку операция, которую пытается выполнить программа, безопасна. О том, как обойти эту проблему, см. раздел «Используйте обобщенные вспомогательные методы для запоминания джокера» главы 7.

Ограничения на джокеры

Джокеры не могут встречаться на верхнем уровне выражений создания экземпляров класса (`new`), в явных параметрах-типах при вызове обобщенного метода и в супертипах (`extends` и `implements`). Рассмотрим эти случаи поочередно.

Создание экземпляра

В выражении создания экземпляра класса, если тип параметризованный, то ни один из параметров-типов не может быть джокером. Например, следующие примеры кода некорректны:

org/jgcbook/chapter02/l_restrictions_on_wildcards/Program_1

```
List<?> list = new ArrayList<?>(); // ошибка компиляции
Map<String, ? extends Number> map =
  new HashMap<String, ? extends Number>(); // ошибка компиляции
```

Обычно это не вызывает трудностей. Принцип получения и вставки говорит нам, что если структура содержит джокер, то мы можем только получать из нее значения (если это джокер с `extends`) или только вставлять в нее значения (если это джокер с `super`). Чтобы структура была полезной, мы должны уметь делать и то и другое. Поэтому обычно создается структура точного типа, даже если мы используем типы с джокерами, чтобы получать или вставлять в нее значения, как показано в следующем примере.

```
List<Number> nums = new ArrayList<Number>();
List<? super Number> sink = nums;
List<? extends Number> source = nums;
for (int i=0; i<5; i++) sink.add(i);
int sum = source.stream().mapToInt(Number::intValue).sum();
assert sum == 10;
```

Здесь джокер встречается во второй и третьей строчках, но не в первой строчке, где список создается.

Только параметры верхнего уровня в выражении создания экземпляра не могут содержать джокеров. Вложенные джокеры разрешены. Поэтому следующий код корректен:

org/jgcbook/chapter02/l_restrictions_on_wildcards/Program_3

```
List<List<?>> lists = new ArrayList<List<?>>();
lists.add(List.of(1,2,3));
lists.add(List.of("four","five"));
assert lists.equals(List.of(List.of(1, 2, 3), List.of("four", "five")));
assert lists.getFirst().getFirst().toString().equals("1");
```

Хотя список списков создается как тип с джокером, у каждого списка внутри него тип конкретный: первый элемент является списком целых, а второй – списком строк. Тип с джокером позволяет извлекать элементы из внутренних списков только как объекты типа `Object`; в последней строке примера именно таков тип выражения

```
lists.getFirst().getFirst().
```

Запомнить это ограничение можно, уяснив, что между джокерами и обыкновенными типами отношение примерно такое же, как между интерфейсами и классами, – джокеры и интерфейсы более общие, обыкновенные типы и класс более конкретные, а для создания экземпляра требуется более конкретная информация. Рассмотрим следующие три предложения:

org/jgcbook/chapter02/l_restrictions_on_wildcards/Program_4

```
List<?> list1 = new ArrayList<Object>(); // ok
List<?> list2 = new List<Object>();      // ошибка компиляции
List<?> list3 = new ArrayList<?>();      // ошибка компиляции
```

Первое предложение корректно, второе некорректно, потому что для создания экземпляра необходим класс, а не интерфейс, а третье некорректно, потому что в выражении создания экземпляра требуется обыкновенный тип, а не джокер.

Возможно, вам непонятно, зачем нужно это ограничение. Проектировщики Java считали, что любой тип с джокером является сокращенной записью какого-то обыкновенного типа, поэтому думали, что в конечном счете при создании любого объекта нужно указывать обыкновенный тип. Неясно, так ли уж необходимо это ограничение, но проблем оно, скорее всего, не вызывает. (Мы всячески пытались придумать ситуацию, когда проблема все-таки есть, но потерпели неудачу!)

Вызовы обобщенных методов

Если вызов обобщенного метод включает явные параметры-типы, то эти параметры не должны быть джокерами. Например, пусть имеется следующий обобщенный метод:

org/jgcbook/chapter02/l_restrictions_on_wildcards/Lists

```
class Lists {
  public static <T> List<T> factory() { return new ArrayList<T>(); }
}
```

Мы можем поручить компилятору вывести параметр-тип, а можем передать его явно. Допустимы оба показанных ниже варианта.

org/jgcbook/chapter02/l_restrictions_on_wildcards/Lists

```
List<?> list1 = Lists.factory();
List<?> list2 = Lists.<Object>factory();
```

Если передан явный параметр-тип, то он не должен быть джокером.

org/jgcbook/chapter02/l_restrictions_on_wildcards/Lists

```
List<?> list3 = Lists.<?>factory(); // ошибка компиляции
```

Как и раньше, вложенные джокеры разрешены.

org/jgcbook/chapter02/l_restrictions_on_wildcards/Lists

```
List<List<?>> list4 = Lists.<List<?>>factory(); // ok
```

Обоснование этого ограничения такое же, как предыдущего. И снова непонятно, так ли оно необходимо, но проблем, скорее всего, не вызывает.

Супертипы

При создании экземпляра класса вызывается конструктор его супертипа. Поэтому любое ограничение, применимое к созданию экземпляра, должно быть применимо и к супертипам. Если супертип или какой-либо суперинтерфейс в объявлении класса имеет параметры-типы, то эти типы не должны быть джокерами.

Например, следующее объявление недопустимо:

```
class AnyList extends ArrayList<?> {...} // ошибка компиляции
```

Так же, как и это:

```
class AnotherList implements List<?> {...} // ошибка компиляции
```

Но, как и прежде, вложенные джокеры разрешены:

```
class NestedList extends ArrayList<List<?>> {...} // ok
```

Обоснование этого ограничения такое же, как двух предыдущих. Как и раньше, его необходимость не очевидна, но и проблем оно, скорее всего, не вызывает.

Заключение

В этой главе мы рассмотрели различные правила подтипизации для массивов и параметризованных типов: массивы ковариантны, а параметризованные типы – нет. Это значит, что в программах с параметризованными типами сам компилятор может обнаруживать ошибки, которые при использовании массивов были бы обнаружены только на этапе выполнения. Для замены ковариантной подтипизации используются параметризованные типы с ограниченными джокерами – в полном соответствии с принципом подстановки. Принцип получения и вставки улавливает связь между ограниченным типом структуры и ее способностью принимать или отдавать значения в диапазоне границ ее типа.

В главе 3 мы узнаем, как параметризованные типы работают с интерфейсами `Comparable<T>` и `Comparator<T>`, играющими очень важную роль в практическом программировании на Java.

Глава 3

Сравнение и границы

Освоив основы, рассмотрим некоторые более продвинутые способы применения параметризованных типов. В этой главе описываются интерфейсы `Comparable<T>` и `Comparator<T>`, которые используются для поддержки сравнения объектов по их элементам. Эти интерфейсы полезны, например, если вы хотите найти максимальный элемент в коллекции или отсортировать список. Попутно мы познакомимся с границами переменных-типов – важной особенностью дженериков Java, которая особенно полезна в сочетании с интерфейсом `Comparable<T>`.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter03*.

Интерфейс Comparable

Интерфейс `Comparable<T>` объявляет один метод экземпляра для сравнения одного объекта с другим:

```
interface Comparable<T> {
  public int compareTo(T other);
}
```

Метод `compareTo` возвращает целое значение, которое может быть отрицательным, нулевым или положительным в зависимости от того, как данный объект соотносится с аргументом: больше, равен или меньше. Если класс реализует `Comparable`, то порядок, индуцированный этим интерфейсом, называется *естественным порядком* для этого класса.

Как правило, объект, принадлежащий некоторому классу, можно сравнивать только с объектами того же класса. Например, класс `Integer` реализует интерфейс `Comparable<Integer>`:

org/jgcbook/chapter03/A_comparable/Program_1

```
Integer int0 = 0;
Integer int1 = 1;
assert int0.compareTo(int1) < 0;
```

Это сравнение возвращает отрицательное число, потому что `0` предшествует `1` в числовом порядке. Аналогично `String` реализует `Comparable<String>`:

org/jgcbook/chapter03/A_comparable/Program_2

```
String str0 = "zero";
String str1 = "one";
assert str0.compareTo(str1) > 0;
```

Это сравнение возвращает положительное число, потому что `"zero"` следует за `"one"` в алфавитном порядке.

Параметр-тип интерфейса позволяет перехватывать бессмысленные сравнения на этапе компиляции:

org/jgcbook/chapter03/A_comparable/Program_3

```
Integer i = 0;
String s = "one";
assert i.compareTo(s) < 0; // ошибка компиляции
```

Можно сравнивать целое с целым или строку со строкой, но попытка сравнить целое со строкой является ошибкой компиляции:

```
Snippet_3.java:8: error: incompatible types: String cannot be converted to
Integer
            assert i.compareTo(s) < 0; // ошибка компиляции
                               ^
```

Сравнение произвольных числовых типов не поддерживается:

org/jgcbook/chapter03/A_comparable/Program_4

```
Number m = Integer.valueOf(2);
Number n = Double.valueOf(3.14);
assert m.compareTo(n) < 0; // ошибка компиляции
```

Здесь сравнение недопустимо, потомку что класс `Number` не реализует интерфейс `Comparable`.

Сравнение отличается от равенства тем, что не принимает аргумент `null`. Если `x` не `null`, то `x.equals(null)` должно возвращать `false`, тогда как `x.compareTo(null)` должно возбуждать исключение `NullPointerException`.

Мы адаптируем стандартные идиомы сравнения, когда пишем `x.compareTo(y) < 0` вместо `x < y` и `x.compareTo(y) <= 0` вместо `x <= y`.

Контракт интерфейса Comparable

Контракт[1] интерфейса `Comparable<T>` определяет три свойства. Свойства определяются в терминах функции `sgn(x)`, которая, по определению, возвращает `-1`, `0` или `1`, если `x` соответственно отрицательно, равно нулю или положительно.

Во-первых, сравнение *антисимметрично*. При изменении порядка аргументов результат меняется на противоположный:

```
sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
```

[1] В программировании контракт определяет предусловия, которым должен удовлетворять клиент (например, сторона, вызывающая метод), и действия, которые должна выполнить служба (например, значения, которые метод возвращает); см. раздел «Контракты» главы 9.

Это обобщение свойства чисел: `x < y` тогда и только тогда, когда `y > x`. Требуется также, чтобы вызов `x.compareTo(y)` возбуждал исключение тогда и только тогда, когда возбуждает исключение вызов `y.compareTo(x)`. Если `x` и `y` одинаковы, то это условие принимает вид:

```
sgn(x.compareTo(x)) == -sgn(x.compareTo(x))
```

Отсюда следует, что

```
x.compareTo(x) == 0
```

поэтому сравнение рефлексивно, т. е. всякое значение равно самому себе.

Во-вторых, сравнение *транзитивно*. Если первое значение меньше второго, а второе меньше третьего, то первое меньше третьего:

```
if x.compareTo(y) < 0 and y.compareTo(z) < 0 then x.compareTo(z) < 0
```

Это обобщение свойства чисел: если `x < y` и `y < z`, то `x < z`.

В-третьих, сравнение является *конгруэнцией*. Это означает, что если два значения равны, результаты сравнения каждого из них с любым третьим значением одинаковы:

```
if x.compareTo(y) == 0 then sgn(x.compareTo(z)) == sgn(y.compareTo(z))
```

Это обобщение свойства чисел: если `x == y`, то `x < z` тогда и только тогда, когда `y < z`. Предполагается, хотя это и не оговорено явно, что если `x.compareTo(y) == 0`, то сравнение `x.compareTo(z)` должно возбуждать исключение тогда и только тогда, когда возбуждает исключение `y.compareTo(z)`.

Согласованность с equals

Контракт `Comparable` настоятельно рекомендует, чтобы метод `compareTo` был *согласован с equals*: два объекта должны удовлетворять методу `equals` тогда и только тогда, когда сравнение говорит, что они одинаковы:

```
x.equals(y) тогда и только тогда, когда x.compareTo(y) == 0
```

Заметим, что это рекомендация, а не обязательная часть контракта. Но на самом деле это обычный случай: для большинства классов естественный порядок отвечает этой рекомендации. Если вы проектируете класс, реализующий интерфейс `Comparable`, не игнорируйте эту рекомендацию без основательных причин. Одна такая причина состоит в том, что различные представления одного и того же значения могут повлиять на результаты вычислений. Самый известный пример этого явления в платформенной библиотеке – класс `java.math.BigDecimal`: два экземпляра этого класса, представляющие одно и то же значение, но с разной точностью, например `4.0` и `4.00`, сравниваются как одинаковые, но метод `equals` возвращает для них `false`.

Одно из следствий такого проектного решения заключается в том, что внутренне упорядоченные коллекции, например `NavigableSet` и `NavigableMap`, ведут себя иначе, чем «типичные» множества, когда содержат такие значения. Мы убедимся в этом во введении к главе 12. В разделе «Несогласованность с equals» главы 18 обсуждаются возможные причины, понуждающие выбрать несогласованность с `equals`, и более общие следствия такого выбора.

Сравнение целочисленных значений

Обратим внимание на одну тонкость в определении *сравнения*. Пусть имеется ряд объектов `Event`, каждый из которых описывается своим именем и числом миллисекунд между моментом возникновения события и текущим моментом (оно может быть как положительно, так и отрицательно). Можно определить естественный порядок для `Event` совпадающим с упорядочением по времени:

org/jgcbook/chapter03/A_comparable/Event_1

```
record Event_1(String name, int millisecs) implements Comparable<Event_1> {
  public int compareTo(Event_1 other) {
    return this.millisecs < other.millisecs ? -1
      : this.millisecs == other.millisecs ? 0
      : 1;
  }
}
```

Условное выражение возвращает `-1`, `0` или `1` в зависимости от того, является получатель (объект `this`) меньшим, равным или большим аргумента. На первый взгляд может показаться, что следующий код тоже будет работать, потому что `compareTo` разрешено возвращать произвольное отрицательное число, если получатель меньше аргумента, и любое положительное, если он больше.

org/jgcbook/chapter03/A_comparable/Event_2

```
record Event_2(String name, int millisecs) implements Comparable<Event_2> {
  public int compareTo(Event_2 other) {
    // плохая реализация – не поступайте так
    return this.millisecs - other.millisecs;
  }
}
```

Но если два сравниваемых значения имеют противоположные знаки, то вычисление разности между ними может привести к переполнению, т. е. результат выйдет за пределы диапазона, который можно сохранить в целом числе – от `Integer.MIN_VALUE` до `Integer.MAX_VALUE`. На самом деле нет нужды выбирать между этими плохими альтернативами: все числовые обертки предлагают статический метод `compare`. Поэтому следующая реализация лучше обеих предыдущих:

org/jgcbook/chapter03/A_comparable/Event_3

```
record Event_3(String name, int millisecs) implements Comparable<Event_3> {
  public int compareTo(Event_3 other) {
    return Integer.compare(this.millisecs, other.millisecs);
  }
}
```

В разделе «Методы интерфейса Comparator» ниже в этой главе мы увидим, что фабричные методы `Comparator` предлагают еще более краткие и гибкие способы сравнения двух объектов.

Максимальный элемент коллекции

В этом разделе мы покажем, как использовать интерфейс `Comparable` для нахождения максимального элемента коллекции. Начнем с упрощенной версии;

настоящая сигнатура метода в каркасе коллекций сложнее, и позднее мы увидим почему.

Следующий код, немного упрощенный по сравнению с тем, что есть в классе `Collections`, ищет максимальный элемент в непустой коллекции.

org/jgcbook/chapter03/B_maximum_of_a_collection/Program_1

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {
  Iterator<? extends T> i = coll.iterator();
  T candidate = i.next();
  while (i.hasNext()) {
    T next = i.next();
    if (next.compareTo(candidate) > 0)
      candidate = next;
  }
  return candidate;
}
```

Впервые мы увидели обобщенные методы, которые объявляют новые переменные-типы в сигнатуре, в разделе «Параметризованные методы и varargs-аргументы» главы 1. Например, метод `asList` принимает массив типа `E[]` и возвращает результат типа `List<E>`, причем делает это для *любого* ссылочного типа `E`. Здесь мы имеем параметризованный метод, который объявляет *границу* переменной-типа. Метод `max` принимает коллекцию типа `Collection<T>` и возвращает значение типа `T` для *любого* типа `T`, являющегося подтипом `Comparable<T>`.

Выделенная фраза в угловых скобках объявляет переменную-тип `T`. Как и в случае джокеров, мы говорим, что `T` ограничен типом `Comparable<T>`. И точно так же переменные-типы, ограниченные сверху, всегда обозначаются ключевым словом `extends`, даже если границей является интерфейс, а не класс, как в данном случае. Но, в отличие от джокеров, переменные-типы могут быть ограничены только с помощью `extends`, но не `super`.

В данном случае граница *рекурсивна* в том смысле, что граница `T` сама зависит от `T`. Разрешены даже взаимно рекурсивные границы, например:

```
<T extends C<T,U>, U extends D<T,U>>
```

Более подробно мы будем изучать рекурсивные границы в разделе «Типы перечислений» ниже в этой главе.

В теле метода `max` мы сначала получаем итератор для коллекции, а затем вызываем метод `next`, чтобы выбрать первый элемент в качестве кандидата на роль максимального; спецификация метода позволяет возбудить исключение `NoSuchElementException`, если на вход ему подана пустая коллекция. Затем кандидат сравнивается с каждым элементом коллекции, и если оказывается, что текущий элемент больше кандидата, то он становится новым кандидатом.

Потоковая альтернатива

Код метода `max`, приведенный в начале этого раздела, упрощенный по сравнению с текущей версией JDK (Java 21), был написан еще до того, как в Java 8 появились потоки и статические методы в интерфейсах, которые позволяют реализовать его короче и, пожалуй, эффективнее. Сигнатура остается той же

самой, но возвращаемое значение имеет тип `Optional<T>`, чтобы обработать случай пустой входной коллекции:

org/jgcbook/chapter03/B_maximum_of_a_collection/Program_2

```
public static <T extends Comparable<T>> Optional<T> max(Collection<T> coll) {
  return coll.stream().max(Comparator.naturalOrder());
   }
```

Мы исследуем свойства интерфейса `Comparator`, благодаря которым это возможно, в разделе «Интерфейс `Comparator`» ниже в этой главе.

При вызове метода класса `Collections` можно в качестве типа `T` указать `Integer` (поскольку `Integer` реализует `Comparable<Integer>`) или `String` (поскольку `String` реализует `Comparable<String>`):

org/jgcbook/chapter03/B_maximum_of_a_collection/Program_3

```
List<Integer> ints = Arrays.asList(0, 1, 2);
assert Collections.max(ints) == 2;

List<String> strs = Arrays.asList("zero", "one", "two");
assert Collections.max(strs).equals("zero");
```

Но `T` не может быть типом `Number` (поскольку `Number` не реализует `Comparable`).

org/jgcbook/chapter03/B_maximum_of_a_collection/Program_4

```
List<Number> nums = Arrays.asList(0, 1, 2, 3.14);
assert Collections.max(nums) == 3.14; // ошибка компиляции
```

Как и следовало ожидать, вызов `max` здесь недопустим.

Объявления методов должны быть настолько общими, насколько возможно; тогда они будут полезнее. Если вы можете заменить параметр-тип джокером, сделайте это. Мы можем улучшить объявление `max`, заменив:

```
<T extends Comparable<T>> T max(Collection<T> coll)
```

на:

```
<T extends Comparable<? super T>> T max(Collection<? extends T> coll)
```

Следуя принципу получения и вставки, мы используем `extends` для `Collection`, потому что *получаем* из коллекции значения типа `T`, но `super` для `Comparable`, потому что *вставляем* значения типа `T` в метод `compareTo`. В следующем разделе мы увидим пример, который не прошел бы проверку типов, если бы мы опустили в этом объявлении фразу `super`.

Взглянув на сигнатуру этого метода в библиотеке Java, вы увидите, что она выглядит еще страшнее, чем в коде выше:

```
<T extends Object & Comparable<? super T>> T max(Collection<? extends T>
coll)
```

Выделенная шрифтом граница понадобилась для обратной совместимости с кодом, написанным до появления параметризованных типов. Мы объясним это в конце раздела «Множественные границы» ниже в этой главе.

Фруктовый пример

Интерфейс `Comparable<T>` позволяет точно управлять тем, что с чем можно и нельзя сравнивать. Допустим, что имеется класс `Fruit` и два его подкласса `Apple` и `Orange`. В зависимости от нашего желания мы можем *запретить* или *разрешить* сравнение яблок с апельсинами.

В примере 3.1 сравнивать яблоки с апельсинами запрещено. Вот какие три класса там объявлены:

```
class Fruit {...}
class Apple extends Fruit implements Comparable<Apple> {...}
class Orange extends Fruit implements Comparable<Orange> {...}
```

У каждого фрукта есть название и размер, и два фрукта считаются равными, если название и размер одинаковы. Поскольку мы переопределили метод `equals`, пришлось также переопределить метод `hashCode`, чтобы равные объекты имели равные хеш-коды (см. врезку «Хеш-таблицы» в разделе «Реализации» главы 9, где объясняется, почему это важно). Яблоки сравниваются по размеру, как и апельсины. Поскольку класс `Apple` реализует интерфейс `Comparable<Apple>`, ясно, что мы можем сравнивать яблоки с яблоками, но не с апельсинами. В тестовом коде строятся три списка: яблок, апельсинов и смеси фруктов. В первых двух списках мы можем найти максимальный элемент, а попытка сделать это в третьем списке приводит к ошибке компиляции:

```
% javac Example31.java
Example31.java:43: error: no suitable method found for max(List<Fruit>)
Collections.max(mixed); ❶ // ошибка компиляции
           ^
method Collections.<T#1>max(Collection<? extends T#1>) is not applicable ❷
  (inference variable T#1 has incompatible bounds
    upper bounds: Object,Comparable<? super T#1>
    lower bounds: Fruit)
method Collections.<T#2>max(Collection<? extends T#2>, \
Comparator<? super T#2>) is not applicable ❸
  (cannot infer type-variable(s) T#2
    (actual and formal argument lists differ in length))
where T#1,T#2 are type-variables:
T#1 extends Object,Comparable<? super T#1> declared in method \
<T#1>max(Collection<? extends T#1>)
T#2 extends Object declared in method \
<T#2>max(Collection<? extends T#2>,Comparator<? super T#2>)
1 error
```

Сообщение может показаться пугающим, но разобраться в нем стоит. Компилятор объясняет почему, видя объявление `mixed` как `List<Fruit>`, он не смог откомпилировать последнюю строку ❶ примера 3.1. Он предпринял две попытки. Сначала он пытался применить метод класса `Collections`:

```
<T extends Object & Comparable<? super T>> T max(Collection<? extends T>
coll)
```

но это не получилось, как констатируется в точке ❷, потому что ни одна возможная подстановка вместо `T` не совместима одновременно с `Fruit` (параметр-тип `mixed`) и `Comparable`. Затем в точке ❸ он сообщает, что не смог применить перегруженный вариант метода `Collections::max` с двумя параметрами.

В примере 3.2 сравнение яблок с апельсинами разрешено. Сравните эти три объявления класса с предыдущими (все различия между примерами 3.1 и 3.2 выделены полужирным шрифтом и здесь, и в коде примеров):

```
class Fruit implements Comparable<Fruit> {...}
class Apple extends Fruit {...}
class Orange extends Fruit {...}
```

Как и прежде, у каждого фрукта есть название и размер, и два фрукта считаются равными, если их названия и размеры одинаковы. Но теперь, поскольку `Fruit` реализует `Comparable<Fruit>`, любые два фрукта можно сравнивать по размеру. Поэтому тестовый код может найти максимум во всех трех списках, включая и тот, где яблоки перемешаны с апельсинами.

Напомним, что в конце предыдущего раздела мы расширили сигнатуру типа метода `max`, включив в нее `super`:

```
<T extends Comparable<? super T>> T max(Collection<? extends T> coll)
```

В примере 3.2 показано, почему этот джокер необходим. Если мы хотим сравнить два апельсина, то вместо `T` в предыдущем коде подставляется `Orange`:

```
Orange extends Comparable<? super Orange>
```

И это правильно, потому что имеют место оба следующих утверждения:

```
Orange extends Comparable<Fruit> и Fruit super Orange
```

Без джокера с `super` поиск максимума в списке `List<Orange>` был бы невозможен, хотя в списке `List<Fruit>` он разрешен. Отметим также, что здесь естественный порядок не согласован с `equals` (см. раздел «Интерфейс Comparable» выше). Два фрукта с разными названиями, но одинаковыми размерами считаются одинаковыми, но они не равны.

Пример 3.1. *Сравнение яблок с апельсинами запрещено*

org/jgcbook/chapter03/C_a_fruity_example/example_3_1/Example31

```
abstract class Fruit {
  protected String name;
  protected int size;
  protected Fruit(String name, int size) {
    this.name = Objects.requireNonNull(name);
    this.size = size;
  }
  public boolean equals(Object o) {
    if (o == null) return false;
    if (o instanceof Fruit that) {
      return this.name.equals(that.name) && this.size == that.size;
    } else return false;
  }
  public int hashCode() {
    return Objects.hash(name, size);
  }
  protected int compareTo(Fruit that) {
    return Integer.compare(this.size, that.size);
  }
}
```

```
class Apple extends Fruit implements Comparable<Apple> {
  public Apple(int size) { super("Apple", size); }
  public int compareTo(Apple a) { return super.compareTo(a); }
}
class Orange extends Fruit implements Comparable<Orange> {
  public Orange(int size) { super("Orange", size); }
  public int compareTo(Orange o) { return super.compareTo(o); }
}
class Test {
  public static void main(String[] args) {
    Apple a1 = new Apple(1); Apple a2 = new Apple(2);
    Orange o3 = new Orange(3); Orange o4 = new Orange(4);

    List<Apple> apples = Arrays.asList(a1, a2);
    assert Collections.max(apples).equals(a2);

    List<Orange> oranges = Arrays.asList(o3, o4);
    assert Collections.max(oranges).equals(o4);

    List<Fruit> mixed = List.of(a1, o3);
    Collections.max(mixed); // ошибка компиляции
  }
}
```

Пример 3.2. *Сравнение яблок с апельсинами разрешено*

org/jgcbook/chapter03/C_a_fruity_example/example_3_2/Example32

```
abstract class Fruit implements Comparable<Fruit> {
  protected String name;
  protected int size;
  protected Fruit(String name, int size) {
    this.name = Objects.requireNonNull(name);
    this.size = size;
  }
  public boolean equals(Object o) {
    if (o == null) return false;
    if (o instanceof Fruit that) {
      return this.name.equals(that.name) && this.size == that.size;
    } else return false;
  }
  public int hashCode() {
    return Objects.hash(name, size);
  }
  public int compareTo(Fruit that) {
    return Integer.compare(this.size, that.size);
  }
}
class Apple extends Fruit {
  public Apple(int size) { super("Apple", size); }
}
class Orange extends Fruit {
  public Orange(int size) { super("Orange", size); }
}
class Test {
  public static void main(String[] args) {
    Apple a1 = new Apple(1); Apple a2 = new Apple(2);
```

```
        Orange o3 = new Orange(3); Orange o4 = new Orange(4);

        List<Apple> apples = Arrays.asList(a1, a2);
        assert Collections.max(apples).equals(a2);

        List<Orange> oranges = Arrays.asList(o3, o4);
        assert Collections.max(oranges).equals(o4);

        List<Fruit> mixed = List.of(a1, o3);
        Collections.max(mixed); // ok
      }
    }
```

Интерфейс Comparator

Иногда нам нужно сравнивать объекты, которые не реализуют интерфейс `Comparable` или сравнивать их по критериям, отличающимся от тех, что определены этим интерфейсом. Порядок, определенный интерфейсом `Comparable`, называется *естественным*, так что интерфейс `Comparator` предлагает в некотором смысле неестественный порядок.

Дополнительные способы упорядочения определяются с помощью интерфейса `Comparator`, в котором объявлен всего один метод[1]:

```
interface Comparator<T> {
  public int compare(T o1, T o2);
}
```

Метод `compare` возвращает отрицательное, нулевое или положительное значение в зависимости от соотношения между первым и вторым объектом: меньше, равен или больше – точно так же, как в случае `compareTo`.

Ниже показан компаратор, который считает более короткую строку меньшей. Только если две строки имеют одинаковую длину, они сравниваются в естественном (алфавитном) порядке.

org/jgcbook/chapter03/D_comparator/Program_1

```
Comparator<String> sizeOrder = new Comparator<>() {
  public int compare(String s1, String s2) {
    if (s1.length() < s2.length())
      return -1;
    if (s1.length() > s2.length())
      return 1;
    return s1.compareTo(s2);
  }
};
```

А вот пример его использования.

---

[1] Заглянув в исходный код `Comparator`, вы увидите, что в нем объявлен также абстрактный метод `equals`. Но все же `Comparator` считается интерфейсом с «единственным абстрактным методом» (или *функциональным*), потому что любой конкретный экземпляр либо наследует реализацию от `Object`, либо переопределяет ее. Метод `equals` – тот самый, что знаком нам по классу `Object`; он включен в интерфейс, чтобы напомнить разработчикам о том, что в равных компараторах метод `compare` должен индуцировать один и тот же порядок.

org/jgcbook/chapter03/D_comparator/Program_1

```
assert "two".compareTo("three") > 0;
assert sizeOrder.compare("two", "three") < 0;
```

В естественном алфавитном порядке строка `"two"` больше `"three"`, тогда как при упорядочении по длине она меньше.

Библиотеки Java всегда предоставляют выбор между `Comparable` и `Comparator`. Для любого обобщенного метода, в котором переменная-тип ограничена `Comparable`, существует другой обобщенный метод с дополнительным аргументом типа `Comparator`. Например, помимо метода

```
public static <T extends Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

имеется метод

```
public static <T>
    T max(Collection<? extends T> coll, Comparator<? super T> comp)
```

Аналогичные методы существуют для нахождения минимального элемента. Вот, например, как можно найти максимальный и минимальный элемент списка с естественным упорядочением и упорядочением по длине строки:

org/jgcbook/chapter03/D_comparator/Program_1

```
Collection<String> strings = Arrays.asList("from","aaa","to","zzz");
assert Collections.max(strings).equals("zzz");
assert Collections.min(strings).equals("aaa");
assert Collections.max(strings, sizeOrder).equals("from");
assert Collections.min(strings, sizeOrder).equals("to");
```

Строка `"from"` максимальна при упорядочении по длине, потому что является самой длинной, а строка `"to"` минимальна, потому что она самая короткая.

Ниже приведен несколько упрощенный, по сравнению с классом `Collection`, код того варианта `max`, который принимает `Comparator`:

org/jgcbook/chapter03/D_comparator/Program_2

```
public static <T extends Comparable<T>>
    T max(Collection<T> coll, Comparator<? super T> comp) {
  Iterator<? extends T> i = coll.iterator();
  T candidate = i.next();
  while (i.hasNext()) {
    T next = i.next();
    if (comp.compare(next, candidate) > 0)
      candidate = next;
  }
  return candidate;
}
```

По сравнению с предыдущей версией (в начале раздела «Максимальный элемент коллекции» выше), изменилось только одно: там, где раньше мы писали `next.compareTo(candidate)`, теперь пишем `comp.compare(next, candidate)`.

Легко определить компаратор, дающий естественный порядок. Ниже приведена немного упрощенная версия кода статического метода `naturalOrder` из интерфейса `Comparator`:

org/jgcbook/chapter03/D_comparator/Program_3

```
public static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
{
   return (o1, o2) -> o1.compareTo(o2);
}
```

С его помощью мы можем определить версию `max`, которая использует естественный порядок в терминах версии с компаратором. Показанный ниже код – перегруженный вариант `max`, который использовался в потоковой версии метода `Collections` в разделе «Интерфейс `Comparable`» выше:

org/jgcbook/chapter03/D_comparator/Program_4

```
public static <T extends Comparable<? super T>> T max(Collection<? extends
T> coll) {
  return Collections.max(coll, Comparator.naturalOrder());
}
```

Методы интерфейса Comparator

В повседневной практике вы, вероятно, будете создавать экземпляры `Comparator` с помощью одного из статических или подразумеваемых по умолчанию методов интерфейса либо комбинации того и другого. Посмотрим, как это работает для простого типа записи.

org/jgcbook/chapter03/D_comparator/Program_6

```
record Person(String name, int age) {}
```

Пожалуй, чаще всего используется метод `comparing` и его варианты, которые принимают функцию, извлекающую ключ для сравнения двух объектов. Например, если мы хотим отсортировать списков объектов `Person` по имени, то можем использовать следующий `Comparator`:

org/jgcbook/chapter03/D_comparator/Program_6

```
Comparator<Person> compareByName = Comparator.comparing(Person::name);
```

Метод `comparing` принимает функцию, которая извлекает ключ сортировки из подлежащих сортировке значений. Как правило, эта функция является методом доступа, реализацией функционального интерфейса `java.util.function.Function`[1]. Она возвращает компаратор, который, будучи применен к двум объектам подлежащего сортировке типа, возвращает значение, соответствующее естественному порядку соответствующих ключей сортировки. Например, если определен список объектов `Person`:

org/jgcbook/chapter03/D_comparator/Program_6

```
Person a32 = new Person("Alice", 32);
Person b23 = new Person("Bob", 23);
List<Person> l = new ArrayList<>(List.of(a32, b23));
```

то мы можем написать:

[1] См. Naftalin (без даты или 2014).

org/jgcbook/chapter03/D_comparator/Program_6

```
l.sort(compareByName);
assert l.equals(List.of(a32, b23));
```

Даже в упрощенном виде объявление метода `Comparator.comparing` в JDK кажется очень сложным:

org/jgcbook/chapter03/D_comparator/Program_5

```
public static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> keyExtractor) {
  return (c1, c2) ->
    keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

На самом деле с большинством из этих трудностей мы уже сталкивались. `T` – тип класса сравниваемых элементов, а `U` – тип ключа сортировки. Как мы видели в конце раздела «Максимальный элемент коллекции» выше, наиболее полезная граница типа допускающих сравнение значений – `U extends Comparable<? super U>`: иными словами, если любой супертип `U` определяет метод `compareTo`, то `U` наследует это определение. Сигнатуру типа интерфейса `Function` можно объяснить, опираясь на принцип подстановки: функция, которая может быть применена к какому-то супертипу `T`, безусловно, может быть применена и к значению типа `T`; обратно, если она возвращает подтип `U`, то, безусловно, допускает использование всюду, где требуется `U`.

Внушающий трепет вид этого объявления резко контрастирует с простотой использования так объявленного метода. И такой контраст присущ параметризованным типам Java: проектировщик API должен тщательно обдумывать, как сделать его максимально общим и удобным. Если дизайн удачен, то разработчики клиентского кода его высоко оценят – ведь им не нужно задумываться о том, как его использовать, поскольку лучшие API отличаются, как это ни парадоксально, своей ненавязчивостью.

Параметр `Function` метода `comparing` применим только к объектам ссылочного типа. Поэтому для примитивных типов нам нужны специализированные версии `comparing`, например:

org/jgcbook/chapter03/D_comparator/Program_6

```
Comparator<Person> compareByAge = Comparator.comparingInt(Person::age);
l.sort(compareByAge);
assert l.equals(List.of(b23, a32));
```

В разделе «Сравнение целочисленных значений» выше мы видели два решения проблемы сравнения объектов `Event`, одно громоздкое, другое дефектное. Мы можем использовать метод `comparingInt` для определения версии `compareTo`, более удобочитаемой и менее подверженной ошибкам, чем первая версия, но без проблемы переполнения, терзающей вторую.

org/jgcbook/chapter03/D_comparator/Event

```
record Event(String name, int millisecs) implements Comparable<Event> {
  private static final Comparator<Event> eventComparator =
    Comparator.comparingInt(Event::millisecs);
  public int compareTo(Event other) {
    return eventComparator.compare(this, other);
  }
}
```

Порядок, определенный компаратором, можно без труда изменить на противоположный, воспользовавшись методом экземпляра `reversed`:

org/jgcbook/chapter03/D_comparator/Program_6

```
l.sort(compareByAge.reversed());
assert l.equals(List.of(a32, b23));
```

Часто сортировка должна быть многоуровневой. Например, когда сортировка по одному ключу дает группы результатов, в каждой из которых значение ключа одно и то же, каждую группу бывает необходимо отсортировать по вторичному ключу. `Comparator` поддерживает такую возможность с помощью метода экземпляра `thenComparing`, создающего новый компаратор, который сравнивает свои аргументы, сначала пользуясь получателем (данным компаратором), а затем, если они равны, компаратором, предоставленным в качестве аргумента. Чтобы продемонстрировать это в действии, добавим новый объект `Person` с таким же именем, как у одного имеющегося человека, и таким же возрастом, как у другого:

org/jgcbook/chapter03/D_comparator/Program_6

```
Person a23 = new Person("Alice", 23);
l.add(a23);
l.sort(compareByName.thenComparing(compareByAge));
assert l.equals(List.of(a23, a32, b23));
```

`Comparator` API спроектирован так, что эти методы допускают компоновку. Предположим, к примеру, что нужно отсортировать список объектов `Person` сначала в порядке убывания возраста, а затем одинаковые объекты в порядке убывания имени. Это просто.

org/jgcbook/chapter03/D_comparator/Program_6

```
l.sort(compareByAge.reversed().thenComparing(compareByName.reversed()));
assert l.equals(List.of(a32, b23, a23));
```

В отличие от `Comparable::compareTo` метод `Comparator::compare` допускает сравнение со значением `null`. Статические методы `nullsFirst` и `nullsLast` принимают компаратор, не допускающий `null`, и возвращают компаратор, допускающий `null`. Этот новый компаратор обращается со значениями, отличными от `null`, так же, как переданный ему компаратор, а к значениям `null` относится спокойно, не возбуждая исключения `NullPointerException`; два значения `null` считаются равными, а всякое значение `null` считается меньше (в случае `nullsFirst`) или больше (в случае `nullsLast`) любого отличного от `null` значения:

org/jgcbook/chapter03/D_comparator/Program_6

```
l.add(null);
l.sort(Comparator.nullsFirst(compareByAge));
assert l.equals(Arrays.asList(null, b23, a23, a32));
```

И напоследок приведем метод, который принимает компаратор элементов, а возвращает компаратор списков элементов.

org/jgcbook/chapter03/D_comparator/Program_7

```
public static <E>
    Comparator<List<E>> listComparator(final Comparator<? super E> comp) {
  return new Comparator<>() {
```

```
    public int compare(List<E> list1, List<E> list2) {
      int n1 = list1.size();
      int n2 = list2.size();
      for (int i = 0; i < Math.min(n1, n2); i++) {
        int k = comp.compare(list1.get(i), list2.get(i));
        if (k != 0) return k;
      }
      return Integer.compare(n1, n2);
    }
  };
}
```

В цикле сравниваются соответственные элементы обоих списков, цикл завершается, когда обнаружены неравные элементы (в этом случае список с меньшим элементом считается меньшим) или когда достигнут конец любого списка (в этом случае более короткий список считается меньшим). Это обычное упорядочение списков; если преобразовать строку в список символов, то оно будет соответствовать обычному (алфавитному) упорядочению строк.

Типы перечислений

В Java 5 включена поддержка типов перечислений, или *enum*, значениями которых являются фиксированные множества констант. Вот два простых примера:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY }
```

Объявление любого типа перечисления можно развернуть в стилизованный класс. Соответствующий класс устроен так, что для каждого элемента перечисления (константы) имеется ровно один экземпляр, привязанный к подходящей статической финальной переменной. Например, первое из приведенных выше объявлений enum развертывается в класс `Season`, имеющий ровно четыре экземпляра, привязанные к статическим финальным переменным с именами `WINTER`, `SPRING`, `SUMMER` и `FALL`. И никакого способа создать дополнительные экземпляры `Season` не существует.

Главным из многих полезных свойств перечислений в Java является типобезопасность: например, нельзя присвоить значение `Weekday` переменной типа `Season` или сравнить значения этих разных типов. Типобезопасность гарантируется тем, что тип перечисления является подклассом класса `java.lang.Enum`, объявленного в примере 3.3. А в примере 3.4 показано объявление одного такого подкласса, `Season`[1].

Класс `Enum` предоставляет свойства, необходимые любому типу перечисления: имя и порядковый номер, т. е. положение в последовательности элементов перечисления. Кроме того, благодаря реализации `Comparable` он гарантирует, что и производные классы также реализуют `Comparable`. Но для типобезопас-

[1] Код в примере 3.3 не во всех деталях совпадает с исходным кодом в библиотеке Java, а большая часть кода, соответствующего примеру 3.4, на самом деле синтезирована компилятором Java – это вообще не корректный Java-код, – но функционально эти примеры эквивалентны коду, который фактически выполняется.

ности недостаточно того, что `Season` реализует `Comparable`; сам по себе этот факт не препятствует сравнению `Season` с `Weekday`. Чтобы это предотвратить, `Season` должен реализовывать интерфейс `Comparable<Season>`.

Это немного приближает нас к пониманию объявления `Enum`:

```
class Enum<E extends Enum<E>> implements Comparable<E>
```

Поначалу это может повергнуть в ужас – с авторами этой книги так и произошло! Но спокойно, без паники! Сопоставление с уже известным покажет, почему класс, производный от `Enum`, будет обладать нужным нам свойством. Из того, что нам известно, следует, что

```
class Season extends ... implements Comparable<Season>
```

где ... – какая-то параметризация `Enum`. Если в качестве этой параметризации мы возьмем `Enum<Season>`, то получим:

```
class Season extends Enum<Season>
class Enum<Season> implements Comparable<Season>
```

и подстановка объявления `Season` в первой строчке вместо первого вхождения `Season` во вторую строчку дает:

```
class Enum<Season extends Enum<Season>> implements Comparable<Season>
```

Конечно, `Enum` должен работать с любым типом перечисления, а не только `Season`, поэтому его объявление в библиотеке классов выглядит так:

```
class Enum<E extends Enum<E>> implements Comparable<E>
```

Пример 3.3. *Базовый класс типов перечислений*

org/jgcbook/chapter03/E_enumerated_types/Enum

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E> {
  private final String name;
  private final int ordinal;
  protected Enum(String name, int ordinal) {
    this.name = name; this.ordinal = ordinal;
  }
  public final String name() { return name; }
  public final int ordinal() { return ordinal; }
  public String toString() { return name; }
  public final int compareTo(E o) {
    return ordinal - ((Enum<?>)o).ordinal;
  }
}
```

Пример 3.4. *Класс, соответствующий типу перечисления*

org/jgcbook/chapter03/E_enumerated_types/Season

```
/*
  Соответствует enum Season { WINTER, SPRING, SUMMER, FALL }
*/
final class Season extends Enum<Season> {
  private Season(String name, int ordinal) { super(name, ordinal); }
  public static final Season WINTER = new Season("WINTER", 0);
  public static final Season SPRING = new Season("SPRING", 1);
  public static final Season SUMMER = new Season("SUMMER", 2);
```

```
    public static final Season FALL = new Season("FALL", 3);
    private static final Season[] VALUES = { WINTER, SPRING, SUMMER, FALL };
    public static Season[] values() { return VALUES.clone(); }
    public static Season valueOf(String name) {
      for (Season e : VALUES) if (e.name().equals(name)) return e;
      throw new IllegalArgumentException();
    }
  }
```

Рекурсивно ограниченные типы наподобие `T extends Comparable<T>` и `E extends Enum<E>` встречаются, когда мы хотим, чтобы параметр мог быть только подтипом определенного типа. Например, в любом подтипе `T` базового класса `Enum` метод `compareTo` может применяться только к аргументу типа `T` и никакого другого подтипа `Enum`; определение предоставляет способ поименовать сам тип в теле класса. (Иногда в обсуждениях рекурсивно ограниченных типов используется термин «автотип» (self-type).)

Остальные определения понятны. Базовый класс `Enum` определяет два поля: строковое имя `name` и целый порядковый номер `ordinal`, имеющиеся в любом экземпляре типа перечисления; эти поля финальные, потому что после инициализации их значения никогда не изменяются. Конструктор класса защищенный, так чтобы его можно было использовать только в подклассах этого класса. В любом классе перечисления конструктор закрытый; это нужно для того, чтобы он мог использоваться только для создания элементов перечисления. Например, в классе `Season` имеется закрытый конструктор, который вызывается только четыре раза для инициализации финальных переменных `WINTER`, `SPRING`, `SUMMER` и `FALL`.

В базовом классе определены методы доступа к полям `name` и `ordinal`. Метод `toString` возвращает имя, а метод `compareTo` – разность порядковых номеров двух элементов перечисления. (В отличие от определения `Event` в разделе «Интерфейс `Comparable`» выше, это безопасно; поскольку порядковые номера всегда положительны, возможность переполнения исключена.) Отсюда следует, что элементы перечисления упорядочены так же, как их порядковые номера, – например, `WINTER` предшествует `SUMMER`.

Наконец, в каждом классе, соответствующем типу перечисления, есть два статических метода. Метод `values` возвращает (поверхностный) клон внутреннего массива, содержащего ссылки на все элементы перечисления. Клонирование необходимо, чтобы клиент не мог изменить внутренний массив. Заметим, что при вызове метода `clone` приведение типа не нужно, потому что клонирование массивов пользуется ковариантностью типа возвращаемого значения (см. раздел «Ковариантное переопределение» ниже в этой главе). Метод `valueOf` принимает строку и возвращает соответствующий элемент перечисления, который ищется в словаре, инициализированном на основе внутреннего массива. Он возбуждает исключение `IllegalArgumentException`, если в перечислении нет элемента с таким именем.

Множественные границы

Мы видели много примеров, когда переменная-тип или джокер ограничены одним классом или интерфейсом. Редко, но бывает, что нужно иметь несколько границ, и мы сейчас покажем, как это сделать.

Для демонстрации будем использовать три интерфейса из библиотеки Java. В интерфейсе `Readable` объявлен метод `read` для чтения в буфер из источника символов, в интерфейсе `Appendable` – метод `append` для копирования из буфера в приемник, способный принимать символы, а в интерфейсе `Closeable` – метод `close` для закрытия источника или приемника. Источниками и приемниками могут служить символьные файлы, буфер, потоки и т. д.

Для максимальной гибкости нам мог бы понадобиться метод `copy`, который принимает поставщика `Supplier` произвольного источника, реализующего интерфейсы `Readable` и `Closeable`, и поставщика `Supplier` произвольного приемника, реализующего интерфейсы `Appendable` и `Closeable`. Метод должен был бы копировать содержимое источника в приемник. К сожалению, типичные читатели и писатели символов не могут быть открыты без риска получить исключение `IOException`, поэтому нам нужен специализированный интерфейс поставщика:

org/jgcbook/chapter03/F_multiple_bounds/IoeThrowingSupplier

```
public interface IoeThrowingSupplier<S> {
  S get() throws IOException;
}
```

Теперь можно объявить метод `copy`:

org/jgcbook/chapter03/F_multiple_bounds/Program_1

```
public static <S extends Readable & Closeable, T extends Appendable & Close-
able
    void copy(IoeThrowingSupplier<S> src, IoeThrowingSupplier<T> tgt, int
size)
        throws IOException {
  try (S s = src.get(); T t = tgt.get()) { ❶
    CharBuffer buf = CharBuffer.allocate(size);
    int i = s.read(buf);
    while (i >= 0) {
      buf.flip(); // подготовить буфер для записи
      t.append(buf);
      buf.clear(); // подготовить буфер для чтения
      i = s.read(buf);
    }
  }
}
```

В объявлении этого метода указано, что `S` может расширять любой тип, реализующий интерфейсы `Readable` и `Closeable`, а `T` может расширять любой тип, реализующий интерфейсы `Appendable` и `Closeable`. Когда для переменной-типа существует несколько границ, они разделяются знаками амперсанда. Типы, определенные несколькими границами, называются *типами-пересечениями*.

Предложение `try`-с-ресурсами ❶ вычисляет указанные лямбда-выражения, которые открывают и возвращают источник и приемник символов. Затем в теле этого предложения в цикле производится чтение из источника в буфер и дописывание из буфера в приемник. Когда источник становится пустым, происходит выход из блока `try` с закрытием источника и приемника. Например, этот код можно вызвать, указав `FileReader` и `FileWriter` в качестве источника и приемника соответственно.

org/jgcbook/chapter03/F_multiple_bounds/Program_1

```
public static void main(String[] args) throws IOException {
  int size = 32;
  Files.writeString(Path.of("file.in"), "hello world");
  copy(() -> new FileReader("file.in"),
       () -> new FileWriter("file.out"), size);
  assert Files.readString(Path.of("File.out")).equals("hello world");
}
```

Из других возможных источников упомянем `FilterReader`, `PipedReader` и `StringReader`, а из приемников – `FilterWriter`, `PipedWriter` и `PrintStream`. Но использовать `StringBuffer` в качестве приемника не получится, потому что этот класс реализует интерфейс `Appendable`, но не `Closeable`.

Если вы знакомы с библиотекой `java.io`, то, наверное, знаете, что все классы, реализующие одновременно `Readable` и `Closeable`, являются подклассами `Reader`, и что почти все классы, реализующие одновременно `Appendable` и `Closeable`, являются подклассами `Writer`. Поэтому может возникнуть вопрос, почему бы не упростить сигнатуру метода следующим образом:

```
public static void copy(Reader src, Writer trg, int size)
```

Это действительно включило бы почти все те же самые классы, но только почти. Например, `PrintStream` реализует `Appendable` и `Closeable`, но не является подклассом `Writer`. Кроме того, нельзя исключить возможность, что программист, использующий ваш код, мог написать собственный класс, который, скажем, реализует `Readable` и `Closeable`, но не является подклассом `Reader`.

Если имеется несколько границ, то первая используется для стирания. Мы видели это в разделе «Максимальный элемент коллекций» выше:

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

Не будь выделенного текста, стертая сигнатура типа для `max` возвращала бы тип `Comparable`, что было бы несовместимо с тем фактом, что до появления дженериков в Java этот метод возвращал тип `Object`. Обеспечение совместимости с унаследованными библиотеками более подробно обсуждается в приложении.

Мостовые методы

Как было объяснено в разделе «Параметризованные типы» главы 3, параметризованные методы реализованы посредством стирания типа: откомпилированное представление кода, написанного с параметризованными типами, почти ничем не отличается от кода, где они не используются. Однако в случае параметризованного супертипа – например, такого интерфейса, как `Comparable<T>`, – стирание может потребовать от компилятора вставки дополнительных методов, которые называются *мостовыми методами*, или просто *мостами*.

В примере 3.5 показан интерфейс `Comparable` и простая запись `Point`, реализующая этот интерфейс. Естественный порядок на множестве точек `Point` определен на основе их расстояний от начала координат (на самом деле удобнее

сравнивать квадраты расстояний, суть дела от этого не меняется). Объявление метода `compareTo` в классе `Point` переопределяет его объявление в `Comparable`, потому что сигнатуры совпадают.

Пример 3.5. *Обобщенный код, реализующий параметризованный интерфейс*

org/jgcbook/chapter03/G_bridges/example_3_5/Example35

```
interface Comparable<T> {
  public int compareTo(T other);
}
record Point(double x, double y) implements Comparable<Point> {
  public int compareTo(Point p) {
    return Double.compare(this.x * this.x + this.y * this.y,
                          p.x * p.x + p.y * p.y);
  }
}
```

Но стирание изменяет эту ситуацию, как показано в примере 3.6.

Пример 3.6. *Параметризованный интерфейс и его реализация после стирания*

org/jgcbook/chapter03/G_bridges/example_3_6/Example36

```
interface Comparable {
  public int compareTo(Object other);
}
record Point(double x, double y) implements Comparable { // не компилируется
  public int compareTo(Point p) {
    return Double.compare(this.x * this.x + this.y * this.y,
                          p.x * p.x + p.y * p.y);
  }
}
```

Сигнатура `compareTo` в интерфейсе изменилась, и, чтобы восстановить переопределение, компилятор должен добавить в реализацию дополнительный метод:

```
public int compareTo(Object other) {
  return compareTo((Point)other);
}
```

Убедиться в существовании этого моста можно с помощью рефлексии.

org/jgcbook/chapter03/G_bridges/example_3_5/Example35

```
Map<Class<?>, Boolean> typeToBridge = (Arrays.stream(Point.class.get-
Methods())
    .filter(m -> m.getName().equals("compareTo"))
    .collect(Collectors.toMap(m -> m.getParameterTypes()[0],
Method::isBridge)));
assert typeToBridge.size() == 2;
assert ! typeToBridge.get(Point.class); // compareTo(Point) - не мостовой
метод
assert typeToBridge.get(Object.class);  // compareTo(Object) - мостовой ме-
тод
```

А полностью объявление метода можно увидеть, декомпилировав файл

класса, например, командой `javap -verbose` или декомпилятором FernFlower с открытым исходным кодом. Код выглядит так:

```
....
// $FF: синтетический метод
// $FF: мостовой метод
public int compareTo(Object var1) {
  return this.compareTo((Point)var1);
}
....
```

Мостовые методы необходимы, когда класс или интерфейс реализует или расширяет параметризованный подтип путем конкретизации его параметра-типа[1].

Мостовые методы играли важную роль в процессе преобразования унаследованного кода с целью использования параметризованных типов, как описано в статье «Maintain Binary Compatibility» *(https://mauricenaftalin.github.io/JGC_2e_Book_Code/maintain_binary_compatibility.html)*.

Ковариантное переопределение

Одновременно с дженериками в Java была включена поддержка ковариантного переопределения методов. Эта возможность напрямую не связана с параметризованными типами, но мы обсудим ее здесь, потому что знать о ней полезно и потому что она реализована с помощью техники мостов, описанной в предыдущем разделе.

В ранних версиях Java один метод мог переопределять другой, только если их сигнатуры и типы возвращаемых значений в точности совпадали. С появлением ковариантного переопределения сигнатуры по-прежнему должны совпадать (после стирания), но требование о совпадении типов возвращаемых значений ослаблено – теперь тип значения, возвращаемого переопределяющим методом, может быть подтипом типа значения, возвращаемого переопределенным методом.

Метод `clone` класса `Object` иллюстрирует преимущества ковариантного переопределения:

```
class Object {
  ...
  protected Object clone() { ... }
}
```

Не будь ковариантного переопределения, любой класс, переопределяющий `clone`, должен был бы возвращать объект точно такого же типа, т. е. `Object`. Например, ниже показана запись `Point`, которая так и делает.

org/jgcbook/chapter03/H_covariant_overriding/Point_1

```
record Point_1(double x, double y) {
  public Object clone() { return new Point_1(x, y); }
}
```

Здесь, хотя `clone` всегда возвращает `Point_1`, старые правила без ковариант-

[1] Строго говоря, это верно, только если стирание изменяет сигнатуру какого-либо метода супертипа.

ности требовали, чтобы типом возвращаемого значения был `Object`. Это раздражало, потому что результат любого вызова `clone` требовалось приводить к истинному типу.

org/jgcbook/chapter03/H_covariant_overriding/Point_1

```
Point_1 p = new Point_1(1, 2);
Point_1 q = (Point_1)p.clone();
```

С появлением ковариантного переопределения стало возможно объявить тип значения, возвращаемого методом `clone`, более точно:

org/jgcbook/chapter03/H_covariant_overriding/Point_2

```
record Point_2(double x, double y) {
  public Point_2 clone() { return new Point_2(x, y); }
}
```

Теперь можно клонировать без приведения.

org/jgcbook/chapter03/H_covariant_overriding/Point_2

```
Point_2 p = new Point_2(1, 2);
Point_2 q = p.clone();
```

Ковариантное переопределение реализовано с использованием техники мостов, описанной в предыдущем разделе. Как и прежде, увидеть мост можно с помощью декомпиляции или рефлексии. Ниже приведен код, который находит все методы с именем `clone` в классе `Point_2`, а затем отображает тип значения, возвращаемого каждым перегруженным вариантом, в результат вызова метода `isBridge` класса `Method`:

org/jgcbook/chapter03/H_covariant_overriding/Point_2

```
Map<Class<?>,Boolean> returnToBridge = (Arrays.stream(Point_2.class.get-
Methods())
    .filter(m -> m.getName().equals("clone"))
    .collect(Collectors.toMap(Method::getReturnType, Method::isBridge)));
assert returnToBridge.size() == 2;
assert returnToBridge.get(Object.class); // Object clone(Point_2) - мост
assert !returnToBridge.get(Point_2.class); // Point_2 clone(Point_2) - не
мост
```

Здесь техника мостов пользуется тем фактом, что два метода в файле одного и того же класса могут иметь одинаковые сигнатуры аргументов, хотя в исходном Java-коде такое запрещено. Мост просто вызывает первый метод.

Заключение

В этой главе мы видели, как интерфейсы `Comparable<T>` и `Comparator<T>` работают со своими параметрами-типами, почему `Comparable<T>` нуждается в рекурсивной границе и как разобраться в более сложной рекурсивной границе типов перечислений.

Начиная с этого момента, мы будем рассматривать тему не с точки зрения программиста клиентов, который использует дженерики Java, а с точки зрения автора библиотеки, который их создает. В следующей главе мы изучим, как объявляется параметризованный класс.

Глава 4

Объявления

В этой главе обсуждается объявление параметризованного класса. Описываются конструкторы, статические члены и вложенные классы, а также дополняется информация о работе стирания.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter04*.

Конструкторы

В параметризованном классе параметры-типы встречаются в заголовке объявления класса, но не в конструкторе.

org/jgcbook/chapter04/A_constructors/Pair

```
class Pair<T,U> {
  private final T first;
  private final U second;
  public Pair(T first, U second) {
    this.first = first;
    this.second = second;
  }
  public T getFirst() { return first; }
  public U getSecond() { return second; }
}
```

Параметры-типы `T` и `U` объявлены в начале класса, а не в конструкторе, но фактические аргументы-типы передаются конструктору при вызове.

org/jgcbook/chapter04/A_constructors/Pair

```
Pair<String, Integer> pair1 = new Pair<String, Integer>("one", 2);
assert pair1.getFirst().equals("one") && pair1.getSecond() == 2;
```

Типичная ошибка – не указать параметры-типы при вызове конструктора:

```
org/jgcbook/chapter04/A_constructors/Pair
Pair pair2 = new Pair<String, Integer>("one", 2);
```

Эта ошибка приводит к выдаче предупреждения при компиляции с флагом `-Xlint`, включающим подробные предупреждения:

```
% javac -Xlint Pair.java
Pair.java:18: warning: [rawtypes] found raw type: Pair
                Pair pair2 = new Pair<String, Integer>("one", 2);
                ^
missing type arguments for generic class Pair<T,U>
where T,U are type-variables:
  T extends Object declared in class Pair
  U extends Object declared in class Pair
1 warning
```

Но компилятор не считает это ошибкой. Код считается допустимым, потому что `Pair` рассматривается как *простой тип*, т. е. тип, который не содержит информации о типах-параметрах, но может быть преобразован в соответствующий параметризованный тип, а это заслуживает только предупреждения об отсутствии проверки. В результате для всех последующих случаев использования `pair2` проверка типов отключается: присвоить значение типа `Pair` можно будет любому типу без дальнейшей выдачи предупреждений или ошибок. Продолжение обсуждения простых типов см. в приложении «Generic Library with Legacy Client» (*https://mauricenaftalin.github.io/JGC_2e_Book_Code/appendix.html*), а типов без проверки – в разделе «Непроверенные приведения» главы 5.

Записи можно параметризовать так же, как обычные классы:

```
record Pair<T,U>(T first, U second) {}
```

Статические члены

Компиляция посредством стирания означает, что во время выполнения параметризованные типы заменяются соответствующими простыми типами, например все интерфейсы `List<Integer>`, `List<String>` и `List<List<String>>` реализованы одним интерфейсом, а именно `List`:

org/jgcbook/chapter04/B_static_members/Program_1

```
List<Integer> ints = Arrays.asList(1, 2, 3);
List<String> strings = Arrays.asList("one","two");
assert ints.getClass() == strings.getClass();
```

Здесь мы видим, что во время выполнения со списком целых ассоциирован тот же объект класса, что и со списком строк.

Одно из следствий такого подхода – тот факт, что статические члены параметризованного класса разделяются между всеми конкретизациями этого класса, в том числе и конкретизациями разных типов. Статические члены класса не могут ссылаться на параметр-тип параметризованного класса, и при доступе к статическому классу имя класса не следует параметризовать.

Например, ниже показан класс `Cell<T>`, в котором каждая ячейка имеет целый идентификатор и значение типа `T`:

org/jgcbook/chapter04/B_static_members/Cell

```
class Cell<T> {
  private final int id;
  private final T value;
  private final static AtomicInteger count = new AtomicInteger();
  private static int nextId() { return count.getAndIncrement(); }
```

```
  public Cell(T value) {
    this.value = value;
    id = nextId();
  }
  public T getValue() { return value; }
  public int getId() { return id; }
  public static int getCount() { return count.get(); }
}
```

Статическое поле `count` служит для присвоения нового идентификатора каждой ячейке. Оно объявлено как `AtomicInteger`, чтобы при конкурентном доступе гарантированно генерировались уникальные идентификаторы. Статический метод `getCount` возвращает текущий счетчик.

Ниже показан код, который создает ячейку, содержащую строку, и другую ячейку, содержащую целое число. Этим ячейкам присвоены идентификаторы 0 и 1 соответственно.

org/jgcbook/chapter04/B_static_members/Cell

```
Cell<String> a = new Cell<String>("one");
Cell<Integer> b = new Cell<Integer>(2);
assert a.getId() == 0 && b.getId() == 1 && Cell.getCount() == 2;
```

Поскольку статические члены разделяются между всеми экземплярами класса, при создании любой ячейки – содержащей как строку, так и целое – инкрементируется один и тот же счетчик.

Поскольку статические члены не зависят от параметров-типов, запрещено указывать имя класса с параметрами-типами при доступе к статическому члену:

```
Cell.getCount();          // ok
Cell<Integer>.getCount(); // ошибка компиляции
Cell<?>.getCount();       // ошибка компиляции
```

Счетчик статический, т. е. является свойством класса в целом, а не его конкретного экземпляра.

По той же причине запрещено ссылаться на параметр-тип в объявлении статического члена. В следующей версии класса `Cell` сделана попытка использовать статическую переменную для хранения списка всех значений, встречающихся в каких-либо ячейках.

org/jgcbook/chapter04/B_static_members/Cell2

```
class Cell2<T> {
  private final T value;
  private static List<T> values = new ArrayList<T>(); // ошибка
  public Cell2(T value) {
    this.value=value;
    values.add(value);
  }
  public T getValue() { return value; }
  public static List<T> getValues() { return values; } // ошибка
}
```

Поскольку класс может использоваться с разными параметрами в разных местах, бессмысленно ссылаться на `T` в объявлении статического поля `values` или статического метода `getValues`, и компилятор считает эти строчки ошибоч-

ными. Если нам все же нужен список всех значений в ячейках, то придется использовать список объектов, как в следующем варианте.

org/jgcbook/chapter04/B_static_members/Cell3

```
class Cell3<T> {
  private final T value;
  private static List<Object> values = new ArrayList<Object>(); // ok
  public Cell3(T value) {
    this.value=value;
    values.add(value);
  }
  public T getValue() { return value; }
  public static List<Object> getValues() { return values; } // ok
}
```

Этот код компилируется и правильно работает.

org/jgcbook/chapter04/B_static_members/Cell3

```
Cell3<String> a = new Cell3<String>("one");
Cell3<Integer> b = new Cell3<Integer>(2);
assert Cell3.getValues().equals(List.of("one", 2));
```

Вложенные классы

Java позволяет вкладывать один класс в другой. Если внешний класс имеет параметры-типы, а внутренний является классом-членом – а не статическим, – то параметры-типы внешнего классы видны во внутреннем классе.

В примере 4.1 показан класс, реализующий коллекции в виде односвязного списка. Этот класс расширяет `java.util.AbstractCollection`, поэтому должен только определить методы `size`, `add` и `iterator` (см. главу 17). Он содержит внутренний класс `Node` для элементов списка и анонимный внутренний класс, реализующий `Iterator<E>`.

Пример 4.1. *Параметры-типы находятся в области видимости классов-членов*

org/jgcbook/chapter04/C_nested_classes/example_4_1/LinkedCollection

```
class LinkedCollection<E> extends AbstractCollection<E> {
  private class Node {
    private final E element;
    private Node next = null;
    private Node(E elt) { element = elt; }
  }
  private Node first = new Node(null);
  private Node last = first;
  private int size = 0;
  public LinkedCollection() {}
  public LinkedCollection(Collection<? extends E> c) { addAll(c); }
  public int size() { return size; }
  public boolean add(E elt) {
    last.next = new Node(elt); last = last.next; size++;
    return true;
  }
  public Iterator<E> iterator() {
    return new Iterator<E>() {
```

```
        private Node current = first;
        public boolean hasNext() {
          return current.next != null;
        }
        public E next() {
          if (current.next != null) {
            current = current.next;
            return current.element;
          } else throw new NoSuchElementException();
        }
        public void remove() {
          throw new UnsupportedOperationException();
        }
      };
    }
  }
```

Параметр-тип `E` находится в области видимости обоих классов. Этот момент важен: если вы его не поймете, то можете впасть в типичную ошибку – подумать, что необходимо распространить область видимости `E` во внутренний класс с помощью объявления вроде `class Node<E>`. Но на самом деле такое действие объявляет новую переменную-тип `E`, которая маскирует переменную-тип `E` из внешнего класса, что ведет к различным недоразумениям. Если ваш внутренний класс действительно нуждается в параметре-типе, отличном от внешнего класса, сделайте его статическим, как описано далее.

В примере 4.2 показана похожая реализация, но на этот раз внутренний класс `Node` статический, поэтому параметр-тип `E` *не* находится в области видимости этого класса. Вместо этого внутренний класс объявлен со своим собственным параметром-типом `T`. Там, где предыдущая версия ссылалась на `Node`, новая ссылается на `Node<E>`. Анонимный класс итератора заменен статическим внутренним классом, снова со своим собственным параметром-типом.

Пример 4.2. *Параметры-типы не находятся в области видимости статических внутренних классов*

org/jgcbook/chapter04/C_nested_classes/example_4_2/LinkedCollection

```
class LinkedCollection<E> extends AbstractCollection<E> {
  private static class Node<T> {
    private final T element;
    private Node<T> next = null;
    private Node(T elt) { element = elt; }
  }
  private Node<E> first = new Node<E>(null);
  private Node<E> last = first;
  private int size = 0;
  public LinkedCollection() {}
  public LinkedCollection(Collection<? extends E> c) { addAll(c); }
  public int size() { return size; }
  public boolean add(E elt) {
    last.next = new Node<E>(elt); last = last.next; size++;
    return true;
  }
  private static class LinkedIterator<T> implements Iterator<T> {
```

```
    private Node<T> current;
    public LinkedIterator(Node<T> first) { current = first; }
    public boolean hasNext() {
      return current.next != null;
    }
    public T next() {
      if (current.next != null) {
        current = current.next;
        return current.element;
      } else throw new NoSuchElementException();
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
  public Iterator<E> iterator() {
    return new LinkedIterator<E>(first);
  }
}
```

Если бы классы элементов были открытыми, а не закрытыми, то в первом примере мы сослались бы на класс элемента как `LinkedCollection<E>.Node`, тогда как во втором – как `LinkedCollection.Node<E>`.

Из двух описанных здесь вариантов мы рекомендуем второй. Классы-члены реализованы путем включения ссылки в объемлющий экземпляр, поскольку они могут в общем случае обращаться к членам этого экземпляра. Статические внутренние классы обычно и проще, и эффективнее.

Как работает стирание

Для стирания типа компилятор выполняет следующие действия: если тип параметризован, отбросить все параметры и заменить любую переменную-тип результатом стирания ее границы или `Object`, если границы нет, или результатом стирания самой левой границы, если границ несколько. Приведем несколько примеров.

- Результатом стирания `List<Integer>`, `List<String>` и `List<List<String>>` является `List`.
- Результатом стирания `List<Integer>[]` является `List[]`.
- Результатом стирания `List` является он сам (это относится к любому простому типу).
- Результатом стирания `int` является он сам (это относится к любому примитивному типу).
- Результатом стирания `Integer` является он сам (это относится к любому типу без параметров).
- Результатом стирания `T` в определении `toList` (см. раздел «Параметризованные методы и varargs-аргументы» главы 1) является `Object`, потому что `T` не имеет границы.
- Результатом стирания `T` в определении `max` (см. раздел «Максимальный элемент коллекции» главы 3) является `Comparable`, потому что `T` имеет границу `Comparable<? super T>`.

- Результатом стирания `T` в окончательном определении `max` (см. раздел «Множественные границы» главы 3) является `Object`, потому что `T` имеет границу `Object & Comparable<T>`, а за результат стирания принимается самая левая граница.
- Результатами стирания `S` и `T` в определении `copy` (см. раздел «Множественные границы» главы 3) являются `Readable` и `Appendable`, потому что `S` имеет границу `Readable & Closeable`, а `T` – `Appendable & Closeable`.
- Результатом стирания `LinkedCollection<E>.Node or LinkedCollection.Node<E>` (см. раздел «Вложенные классы» выше в этой главе) является `LinkedCollection.Node`.

В Java два метода одного и того же класса не могут иметь одинаковые сигнатуры, т. е. одинаковые имена и типы параметров. Поскольку параметризованные классы реализованы посредством стирания, отсюда следует, что два разных метода не могут иметь сигнатуры, оказывающиеся одинаковыми после стирания. В классе не может быть двух перегруженных методов с одинаковыми после стирания сигнатурами, и класс не может реализовывать два интерфейса, ставшие одинаковыми после стирания.

Например, ниже показан класс с двумя методами. Один складывает все элементы списка целых, а другой конкатенирует все элементы списка строк.

org/jgcbook/chapter04/D_how_erasure_works/Overloaded

```
class Overloaded {
  // ошибка компиляции, нельзя иметь два метода с одинаковыми сигнатурами
после
  // стирания
  public static int sum(List<Integer> ints) {
    int sum = 0;
    for (int i : ints) sum += i;
    return sum;
  }
  public static String sum(List<String> strings) {
    StringBuilder sum = new StringBuilder();
    for (String s : strings) sum.append(s);
    return sum.toString();
  }
}
```

После стирания объявления этих методов будут выглядеть так:

```
int sum(List)
String sum(List)
```

Но именно сигнатуры, а не типы возвращаемых значений позволяют компилятору Java различать перегруженные варианты методов. В данном случае после стирания сигнатуры обоих методов одинаковы:

```
sum(List)
```

поэтому компилятор сообщает о конфликте имен:

```
Overloaded.java:11: error: name clash: sum(List<String>) and
sum(List<Integer>) \
have the same erasure
  public static String sum(List<String> strings) {
                       ^
```

Рассмотрим другой пример. Ниже приведена некорректная версия класса `Integer`, в которой сделана попытка сравнивать целое с целым или с длинным целым:

org/jgcbook/chapter04/D_how_erasure_works/Integer

```
class Integer implements Comparable<Integer>, Comparable<Long> {
  // ошибка компиляции, нельзя реализовывать два интерфейса одинаковыми
  // сигнатурами после стирания
  private int value;
  ...
  public int compareTo(Integer i) {
    return (value < i.value) ? -1 : (value == i.value) ? 0 : 1;
  }
  public int compareTo(Long l) {
    return (value < l.intValue()) ? -1 : (value == l.intValue()) ? 0 : 1;
  }
  ...
}
```

Если бы эта возможность поддерживалась, то в общем случае потребовалось бы сложное и запутанное определение мостовых методов (см. раздел «Мостовые методы» главы 3). Намного проще и понятнее запретить ее вовсе.

Заключение

В этой главе мы видели, как объявить параметризованный класс с конструкторами, статическими членами и вложенными классами, и изучили дополнительные детали механизма стирания.

В следующей главе мы узнаем о других последствиях стирания, в частности о не вполне естественной связи между массивами и параметризованными типами.

Глава 5

Сберегаемые и несберегаемые типы

К моменту включения параметризованных типов в Java в 2004 году, спустя восемь лет после выхода версии 1.0, работающие коммерческие приложения насчитывали уже много миллионов строк кода. И за то, что внедрение прошло так успешно, нужно отдать должное дизайну дженериков, благодаря которому удалось перевести на них все основные библиотеки и большую часть клиентского кода за несколько лет без серьезных проблем совместимости. Это достижение описано в приложении (*https://mauricenaftalin.github.io/JGC_2e_Book_Code/appendix.html*). Ключом к успеху стало стирание типов, без которого Java не мог бы существовать в том виде, в каком мы его сегодня знаем. Унаследованный код не различает `List<Integer>`, `List<String>` и `List<List<String>>`, поэтому стирание параметров-типов необходимо, чтобы упростить эволюцию и обеспечить совместимость между унаследованным и новым кодом. Но все имеет цену, и в этой главе мы должны оплатить свои долги.

В оксфордском словаре английского языка слово *reify* определяется так: «мысленно овеществлять, материализовывать». Более простое слово с тем же значением – *thingify*. В информатике под *reification* понимается явное представление типа, т. е. информация о типе во время выполнения[1]. В Java сбереженный массив сохраняет информацию о типе своих элементов, тогда как сбереженный параметризованный тип не сохраняет информацию о своих параметрах-типах.

Сбережение играет критически важную роль в некоторых аспектах Java, и его отсутствие в параметризованных типах, как бы полезно оно ни было для эволюции, по необходимости приводит к некоторым осложнениям. В этой главе мы предупредим вас об ограничениях и расскажем, как их обойти. Речь пойдет о вещах, которые могут показаться вам необязательными для изучения, – и действительно, если вы никогда не используете параметризованные типы в приведениях, тестах экземпляров, исключениях или массивах, то вряд ли рассматриваемый здесь материал вам понадобится.

Мы начнем с точного определения того, что означает сберегаемый тип в Java. Затем мы рассмотрим пограничные случаи, связанные со сбережением,

[1] Я выбрал перевод «сбережение», поскольку речь идет о сохранении некоторой информации о типе компилятором. Выражение «материализуемый» тип никак не отражает этот смысл, поскольку непонятно из какого такого эфира он материализуется — *Прим. перев.*

включая тесты экземпляров, приведения, исключения и массивы. Сопряжение массивов с параметризованными типами – худший аспект языка, поэтому мы сформулируем два принципа, которые помогут вам избежать ловушек: принцип правдивости рекламы и принцип непристойного обнажения.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter05*.

Сберегаемые типы

В Java тип массива сберегается *с* типом элементов, а параметризованный тип – *без* параметров-типов. Например, массив чисел несет с собой сбереженный тип `Number[]`, тогда как тип `ArrayList<Number>` несет сбереженный тип `ArrayList`; сберегается простой, непараметризованный тип. Конечно, к каждому элементу списка присоединен сбереженный тип, скажем, `Integer` или `Double`, но это не то же самое, что сбережение параметра-типа. Если бы каждый элемент списка был целым, то во время выполнения мы не смогли бы сказать, что это: `ArrayList<Integer>`, `ArrayList<Number>` или `ArrayList<Object>`; если бы список был пустым, то мы ничего не смогли бы сказать о том, какого вида этот пустой список.

В Java говорят, что тип *сберегаемый*, если он полностью представлен во время выполнения, т. е. если стирание не уничтожает никакой полезной информации. Тип является сберегаемым в следующих случаях:

- примитивный тип (например, `int`);
- непараметризованный класс или тип интерфейса (например, `Number`, `String` или `Runnable`);
- простой тип (например, `List`, `ArrayList` или `Map`);
- параметризованный тип, в котором все аргументы-типы являются неограниченными джокерами (например, `List<?>`, `ArrayList<?>` или `Map<?,?>`);
- массив, содержащий сберегаемые элементы (например, `int[]`, `Number[]`, `List[]`, `List<?>[]` или `int[][]`).

Тип *не* является сберегаемым в следующих случаях:

- переменная-тип (например, `T`);
- параметризованный тип с параметрами, не являющимися джокерами (например, `List<Number>`, `ArrayList<String>` или `Map<String, Integer>`);
- параметризованный тип с параметром, являющимся ограниченным джокером (например, `List<? extends Number>` или `Comparable<T super String>`).

Так, например, тип `List<? extends Object>` *не* является сберегаемым, хотя он и эквивалентен `List<?>`. Такое определение сберегаемых типов упрощает их синтаксическую идентификацию.

Тесты экземпляров и приведения

До версии Java 16 тесты экземпляров и приведения опирались только на исследование типа во время выполнения, поэтому их можно было применять

только к сбереженным типам. Вот, например, образец метода `equals` для пользовательского метода `MyType`, предложенный в книге Bloch (2017, совет 10):

```
public boolean equals(Object o) {
  if (!(o instanceof MyType))
    return false;
  MyType mt = (MyType) o;
  // теперь сравнить значения полей этого объекта и mt
}
```

Этот метод принимает аргумент типа `Object`, проверяет, является ли объект экземпляром ссылочного типа `MyType`, и, если да, приводит его к этому типу и сравнивает значения полей обоих объектов. Если ссылочный тип сберегаемый, то вся информация, необходимая для проверки того, что объект является его экземпляром, доступна во время выполнения. Используемый таким образом оператор `instanceof` называется *оператором сравнения типов*.

В версии Java 16 оператор `instanceof` стал перегруженным. Теперь в зависимости от контекста он является либо оператором сравнения типов, либо оператором *сопоставления с образцом*. Ниже показан метод `equals`, переписанный с использованием оператора сравнения с образцом:

```
public boolean equals(Object o) {
  return o instanceof MyType mt && ...
  // теперь сравнить значения полей этого объекта и mt
}
```

Вычисление выражения образца `o instanceof MyType mt` по-прежнему возвращает значение истинности выражения сравнения типов `o instanceof MyType`. Но дополнительно если это значение равно `true`, то оно объявляет переменную `mt` типа `MyType` и привязывает его к значению `(MyType)o`. По счастью, правила для типов в таких выражениях сопоставления с типом-образцом настолько похожи на действующие для выражений сравнения типов, что мы можем ограничить обсуждение последними.

До Java 16 второй операнд оператора сравнения типов должен был быть сберегаемым. Но в Java 16 это ограничение было снято: теперь некоторые тесты экземпляров несберегаемых типов можно вычислить целиком на этапе компиляции. Например, следующий код компилируется (а значит, и выполняется) без проблем:

org/jgcbook/chapter05/B_instance_tests_and_casts/Program_1

```
// компилируется, начиная с Java 16
ArrayList<Integer> x = new ArrayList<>();
assert x instanceof List<? extends Number>;
```

Предложение `assert` здесь лишнее; компилятор самостоятельно может определить, что поскольку `ArrayList` является подтипом `List`, а `Integer` подтипом `Number`, то `ArrayList<Integer>` является подтипом `List<? extends Number>`.

Эта возможность была добавлена не для таких игрушечных примеров, а для использования в сочетании с вычислением `instanceof` во время выполнения. Например, следующий метод преобразует любую коллекцию в список:

org/jgcbook/chapter05/B_instance_tests_and_casts/Program_2

```
// компилируется, начиная с Java 16
public static <T> List<T> asList(Collection<T> cl) {
  return cl instanceof List<T> ? (List<T>)cl : cl.stream().toList();
}
```

Или с использованием выражения сопоставления с образцом:

org/jgcbook/chapter05/B_instance_tests_and_casts/Program_3

```
// компилируется, начиная с Java 16
public static <T> List<T> asList(Collection<T> cl) {
  return cl instanceof List<T> q ? q : cl.stream().toList();
}
```

И тест экземпляра, и приведение корректно типизированы, потому что параметр-тип в объявлении `cl` можно привести к типу `List<T>` – а в данном случае они просто совпадают. Затем во время выполнения если сбереженный тип `cl` оказывается подтипом `List`, то `cl` приводится к `List<T>`. В противном случае коллекция потоком копируется в новый список.

Однако часто тесты экземпляров и приведения, в которых участвуют несберегаемые типы, невозможно подвергнуть такой проверке типов. Если это так, то попытка откомпилировать тест экземпляра приводит к ошибке, тогда как компиляция приведения – к предупреждению. Например, рассмотрим возможное определение равенства списков, как в классе `AbstractList` из пакета `java.util`. Вот наивный – и неправильный – способ:

org/jgcbook/chapter05/B_instance_tests_and_casts/AbstractList_1

```
abstract class AbstractList_1<E> extends AbstractCollection<E> implements
List<E> {
  public boolean equals(Object o) {
    if (!(o instanceof List<E>)) { return false; } // ошибка компиляции
    ListIterator<E> it1 = listIterator();
    ListIterator<E> it2 = ((List<E>)o).listIterator(); // приведение без
проверки
    while (it1.hasNext() && it2.hasNext()) {
      E e1 = it1.next();
      E e2 = it2.next();
      if (! Objects.equals(e1, e2))
        return false;
    }
    return !(it1.hasNext() || it2.hasNext());
  }
  ...
}
```

И снова метод `equals` принимает аргумент типа `Object`, но на этот раз проверяет, является ли объект экземпляром несберегаемого типа, а именно `List<E>`. Если да, то он приводится к `List<E>` и сравнивается с соответственными элементами обоих списков. Код теста экземпляра не компилируется, но причины для версий, предшествующих 16 и следующих за ней, различны. До выхода Java 16 компилятор сообщал об ошибке для теста экземпляра *любого* несберегаемого типа:

```
% javac AbstractList_1.java
AbstractList_1.java:8: illegal generic type for instanceof
```

```
    if (!(o instanceof List<E>)) { return false; } // ошибка компиляции
                       ^
Note: AbstractList_1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
1 warning
```

Начиная с версии Java 16, проблемой становится типобезопасность теста:

```
% javac AbstractList_1.java
AbstractList_1.java:8: error: Object cannot be safely cast to List<E>
    if (!(o instanceof List<E>)) { return false; } // ошибка компиляции
          ^
  where E is a type-variable:
    E extends Object declared in class AbstractList_1
Note: AbstractList_1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
1 warning
```

Теперь проверка экземпляра порождает ошибку, потому что не существует никакого способа – ни на этапе компиляции, ни на этапе выполнения – проверить, принадлежит ли данный объект типу `List<E>`[1].

Приведение – другое дело. Компилятор выдает для непроверенных приведений предупреждения, а не ошибки – как для выражений сравнения типов без проверки, – потому что обычно не может прийти к определенному мнению об их типобезопасности. В этом случае компилятор выполнит приведение, но не сможет проверить, действительно ли элементы списка принадлежат типу `E`. Если перекомпилировать файл с флагом `-Xlint`, то будет выдано следующее сообщение с такой же диагностикой ошибки компиляции, как раньше:

```
AbstractList_1.java:11: warning: [unchecked] unchecked cast
   ListIterator<E> it2 = ((List<E>)o).listIterator(); // приведение без проверки
                                ^
  required: List<E>
  found:    Object
  where E is a type-variable:
        E extends Object declared in class AbstractList_1
1 error
1 warning
```

Предупреждения о невозможности проверки важны: очень может статься, что они сообщают о небезопасности типа, которая впоследствии приведет к ошибке времени выполнения (вспомните подвох, связанный с гарантией успешного приведения!). Мы обсудим, что с ними делать, в следующем разделе и в разделе «Искореняйте предупреждения о невозможности проверки» главы 7.

В данном случае и предупреждение о невозможности проверки, и ошибку компиляции можно устранить, заменив несберегаемый тип `List<E>` сберега-

[1] Даже если бы этот код работал, все равно есть еще одна проблема. В контракте о равенстве списков типы не упоминаются. `List<Integer>` должен быть равен `List<Object>`, если они содержат одинаковые значения в одном и том же порядке. Например, список `[1,2,3]` должен быть равен себе вне зависимости от того, рассматривается он как список целых или как список объектов.

емым типом `List<?>`. Вот исправленное определение (упрощенное по сравнению с настоящим исходным кодом):

org/jgcbook/chapter05/B_instance_tests_and_casts/AbstractList_2

```
abstract class AbstractList_2<E> extends AbstractCollection<E> implements
List<E> {
  public boolean equals(Object o) {
    if (!(o instanceof List<?>)) { return false; }
    ListIterator<E> it1 = listIterator();
    ListIterator<?> it2 = ((List<?>)o).listIterator();
    while (it1.hasNext() && it2.hasNext()) {
      E e1 = it1.next();
      Object e2 = it2.next();
      if (! Objects.equals(e1, e2))
      return false;
    }
    return !(it1.hasNext() || it2.hasNext());
  }
    ...
 }
```

Помимо изменения типа теста экземпляра и приведения, тип второго итератора изменился с `Iterator<E>` на `Iterator<?>`, а тип второго элемента – с `E` на `Object`. Этот код проходит проверку типов, потому что, хотя тип элемента во втором итераторе неизвестен, гарантируется, что он будет подтипом `Object`, а вложенный вызов `equals` требует только, чтобы его второй аргумент был объектом[1].

Возможны и другие способы исправления. Например, вместо джокерных типов `List<?>` и `Iterator<?>` можно было бы использовать простые типы `List` и `Iterator`, которые также являются сберегаемыми. Но мы рекомендуем использовать типы с неограниченными джокерами, а не простые типы, потому что они дают более сильные гарантии статической типизации; многие ляпы, которые помечаются как ошибки при использовании неограниченных джокеров, будут считаться лишь предупреждениями при использовании простых типов. Например, в разделе «Ограничения на джокеры» главы 2 мы видели объявление:

```
List<List<?>> lists = new ArrayList<List<?>>();
```

Типом `lists` является список списков, у каждого из которых свой тип. Попытка скопировать элемент из одного внутреннего списка в другой приведет к ошибке компиляции. Если бы типом был `<List<List>>`, то не было бы никаких ограничений на типы объектов во внутренних списках, и копирование привело бы только к предупреждению о невозможности проверки.

Еще одно исправление `AbstractList_1` – изменить объявление первого итератора на `Iterator<?>`, а первого элемента – на `Object`, так чтобы они совпадали со вторым итератором, а код по-прежнему проходил проверку типов. Но мы

[1] В отличие от предыдущей версии этот код удовлетворяет контракту о равенстве списков. Теперь `List<Integer>` будет равен `List<Object>`, если они содержат одинаковые значения в одном и том же порядке.

рекомендуем всегда использовать максимально специфичные объявления типов, это позволит компилятору выловить больше ошибок и сгенерировать более эффективный код.

Непроверенные приведения

Лишь очень редко компилятор способен определить, как в предыдущем примере, что успешное приведение к несберегаемому типу должно давать значение этого типа. Во всех остальных случаях приведение к несберегаемому типу выдает предупреждение о невозможности проверки. Это контрастирует со сравнениями типов, в которых участвуют несберегаемые типы; для них компилятор либо определяет, что они типобезопасны, либо сообщает об ошибке. Мы видели примеры обеих ситуаций.

В предыдущем разделе мы видели, что предупреждение о невозможности проверки означает лишь, что компилятор не может подтвердить безопасность кода. Но никакая система типов не идеальна: всегда будут существовать какие-то факты, которые программист может вывести, а система типов нет. Примеры неизбежных, но безопасных предупреждений о невозможности проверки см. в разделах «Как определить ArrayList» и «Создание массивов и varargs-аргументы» ниже в этой главе, «Приводите через простые типы, когда необходимо» и «Используйте параметризованные типы массивов с осторожностью» главы 7, а также в многочисленных местах главы 6 и приложения.

Мы опишем, что делать в таких ситуациях в разделе «Искореняйте предупреждения о невозможности проверки» главы 7. Короче говоря, включив в документацию аргументы, доказывающие, что ваш код действительно типобезопасен, вы должны использовать аннотацию `@SuppressWarnings("unchecked")`, чтобы подавить случайные предупреждения компилятора, которые могли бы замаскировать настоящие.

Непроверенные приведения в Java – это маневр для обхода проблемы, с которой мы в большей или меньшей степени сталкиваемся в любом статически типизированном языке. Для сравнения: C не проверяет приведения типов, а в его преемнике C++ непроверенные приведения куда опаснее, чем в Java. И в том, и в другом языке результат разыменования некорректного указателя не определен. Это означает, что программа может аварийно завершиться, молча вернуть неправильный результат или повредить совершенно посторонние данные. В отличие от C среда выполнения Java гарантирует, что ничего такого из-за непроверенных приведений не случится. Программа может возбудить исключение `ClassCastException` и не довести задачу до конца, но JVM останется в корректном состоянии, и повреждения посторонних данных не произойдет. Тем не менее непроверенные приведения в Java остаются обходным маневром, прибегать к которому следует с осторожностью.

Обработка исключений

В предложении `try` каждая фраза `catch` проверяет, соответствует ли возбужденное исключение (или несколько исключений) данному типу. Это та же проверка, что выполняется в тесте экземпляра, поэтому применимо то же ограничение:

тип должен быть сберегаемым. Кроме того, тип во фразе `catch` обязан быть подклассом `Throwable`. Поскольку вряд ли имеет смысл создавать подкласс `Throwable`, который никогда не будет встречаться во фразе `catch`, компилятор Java ругается, когда вы пытаетесь создать параметризованный подкласс `Throwable`.

Например, показанное ниже определение нового исключения, содержащего целое значение, допустимо:

org/jgcbook/chapter05/C_exception_handling/IntegerExceptionTest

```
class IntegerException extends Exception {
  private final int value;
  public IntegerException(int value) { this.value = value; }
  public int getValue() { return value; }
}
```

А вот простой пример использования этого исключения:

org/jgcbook/chapter05/C_exception_handling/IntegerExceptionTest

```
class IntegerExceptionTest {
  public static void main(String[] args) {
    try {
      throw new IntegerException(42);
    } catch (IntegerException e) {
      assert e.getValue() == 42;
    }
  }
}
```

В теле предложения `try` возбуждается исключение с заданным значением, которое перехватывается во фразе `catch`.

С другой стороны, следующее определение исключения запрещено, потому что его тип параметризован.

org/jgcbook/chapter05/C_exception_handling/ParametricException

```
class ParametricException<T> extends Exception { // ошибка компиляции
  private final T value;
  public ParametricException(T value) { this.value = value; }
  public T getValue() { return value; }
}
```

Попытка откомпилировать этот код рассматривается как ошибка:

```
% javac ParametricException.java
ParametricException.java:1: a generic class may not extend java.lang.Throw-
able
class ParametricException<T> extends Exception { // ошибка компиляции
                                 ^
1 error
```

Это ограничение разумно, потому что почти любая попытка перехватить такое исключение должна быть безуспешной, так как тип не сберегаемый. В полном соответствии с этим ограничением на объявление типов исключений тип во фразе `catch` тоже не может иметь параметров.

Хотя подклассы `Throwable` не могут быть параметризованными, переменную-тип можно использовать во фразе `throws` объявления метода, как в примере 5.1.

Пример 5.1. *Использование переменной-типа во фразе* `throws`

org/jgcbook/chapter05/C_exception_handling/TypeVariableInThrowsClause

```
public class TypeVariableInThrowsClause {
  @FunctionalInterface
  interface Testable<X extends Throwable> {
    void run() throws X;
  }

  static <X extends Throwable> void checkThrows(Testable<X> test, Class<X>
clazz) {
    try {
      test.run();
    } catch (Throwable t) {
      if (clazz.isInstance(t)) {
        return;
      } else {
        throw new AssertionError(t);
      }
    }
    throw new AssertionError("no exception was thrown");
  }

  public static void main(String[] args) {
    checkThrows(() -> List.of().iterator().next(), NoSuchElementException.
class);
    checkThrows(() -> { throw new Exception(); }, Exception.class);
  }
}
```

Класс `TypeVariableInThrowsClause` лежит в основе гипотетического каркаса для тестирования лямбда-выражений, которые возбуждают исключения некоторого типа. Метод `checkThrows` принимает лямбда-выражение функционального типа `Testable<X>`, где `X` – подтип `Throwable`. Он вычисляет лямбду, а затем использует свой второй параметр, объект класса `Class<X>`, чтобы с помощью отражения проверить, что исключение, фактически возбужденное лямбдой, является экземпляром этого класса.

Параметризованные типы и массивы

Как мы видели в разделе «Массивы» главы 2, тип элементов массива сберегаемый, т. е. массивы переносят на этап выполнения информацию о типе своих элементов, в отличие от параметризованных структур данных, для которых информация о типе после компиляции стирается. Из-за такого разнобоя стыковка – или, правильнее сказать, нестыковка – между массивами и параметризованными типами является самой неприятной проблемной областью языка.

Поэтому оставшаяся часть этой главы посвящена уяснению проблем, вызванных этой нестыковкой, и способам их обхода. На протяжении нескольких разделов нашим сквозным примером будет `ArrayList`, потому что изучение реализации параметризованной коллекции, в основе которой лежит массив, позволит исследовать основные болевые точки этой многострадальной связи и как их обойти. Как вы, наверное, понимаете, корень зла – преобразование между коллекциями и массивами, а точнее – поскольку массивы

нуждаются в сбереженной информации о типе, которой параметризованные коллекции не обладают, – преобразование *из* параметризованной коллекции *в* массив.

Методы, предоставляемые для этого преобразования каркасом коллекций, объявлены в интерфейсе `java.util.Collection`, который является супертипом всех коллекций, кроме отображений:

```
interface Collection<E> {
  ...
  Object[] toArray(); ❶
  <T> T[] toArray(IntFunction<T[]> generator) ❷
  <T> T[] toArray(T[] a) ❸
  ...
}
```

Взяв в качестве примера `ArrayList`, посмотрим на реализацию первых двух методов.

- Метод ❶: поскольку за `ArrayList` стоит `Object[]`, этот метод реализован просто копированием массива.
- Метод ❷: его можно определить в терминах метода ❸. Именно так он определен в интерфейсе `Collection`, где является методом по умолчанию:

  ```
  default <T> T[] toArray(IntFunction<T[]> generator) {
  return toArray(generator.apply(0));
  }
  ```

  `generator` – функция, которая принимает `int` и порождает массив `T[]` такой длины. Например, следующая лямбда является генератором для массивов строк:

  ```
  int n -> new String[n]
  ```

  В типичном вызове метода ❷ используется ссылка на метод, эквивалентная этой лямбде:

  ```
  Collection<String> cs = ...
  String[] sa = cs.toArray(String[]::new);
  ```

  Метод ❷ применяет эту функцию, чтобы создать массив того же размера, что у коллекции, а затем вызывает метод ❸, передавая ему результат.

- Метод ❸ и будет предметом обсуждения в трех следующих разделах. Он принимает массив некоторого сберегаемого типа `T`. Если этот массив достаточно длинный, то он копирует в него элементы коллекции, частично или полностью перезаписывая содержимое, и возвращает массив. В противном случае он создает новый массив того же типа, длина которого равна размеру коллекции, копирует в него элементы коллекции и возвращает.

Прежде чем переходить к реализации метода ❸, нужно ответить на два вопроса, которые поднимает его объявление и спецификация.

- Почему параметр-тип `T` не связан с типом коллекции `E` и какие следствия из этого вытекают? Краткий ответ – такое объявление дает возможность сделать тип элементов массива более узким, чем тип коллекции. Более полное обсуждение см. в разделе «Обеспечение доступности содержимого коллекции для дальнейшей обработки» главы 10.

- Что вызовы метод должны передавать в качестве параметра `T[]`? У этого метода две цели. Первая, которая удовлетворяется всегда, – передать сбереженный тип для возврата массива, ее мы будем изучать в последующих разделах. Вторая, удовлетворяемая, только если массив достаточно длинный, – играть роль получателя элементов коллекции. Когда вы пишете код, вызывающий этот метод, у вас есть выбор: передать заранее выделенный массив достаточной длины или передать меньший массив (возможно, нулевой длины) и позволить методу произвести выделение памяти. Этот выбор обсуждается в разделе «Обеспечение доступности содержимого коллекции для дальнейшей обработки» главы 10, где мы приходим к выводу, что с точки зрения эффективности лучшим обычно является вариант ❷.

Создание параметризованного массива

Здесь наша цель, которую мы достигнем в разделе «Как определить `ArrayList`», – представить миниатюрную реализацию класса `ArrayList`. Пока что сосредоточимся на методе из предыдущего раздела, а чтобы еще упростить задачу, предположим ненадолго, что спецификация всегда требует создания нового массива и никогда – повторного использования переданного массива. В первой попытке посмотрим, можно ли вообще избежать использования переданного массива:

org/jgcbook/chapter05/E_generic_array_creation/WrongMicroArrayList_1

```
class WrongMicroArrayList_1<E> {
  private int size;
  private Object[] data;
  ...
  public <T> T[] toArray(T[] a) {
    a = new T[size]; // ошибка компиляции
    System.arraycopy(data, 0, a, 0, size);
    return a;
  }
}
```

В массиве `data` типа `Object[]` хранится содержимое `ArrayList`, а в переменной `size` – размер коллекции, которая зачастую занимает только часть массива. Эта версия `toArray` принимает ссылку на массив, но просто пытается присвоить переменной-аргументу ссылку на вновь созданный массив параметризованного типа `T`. Если бы это удалось, то вызов `System::arraycopy` эффективно скопировал бы содержимое хранимого массива в новый.

Но попытка компиляции этой программы завершается безуспешно, потому что выделенный код создает новый массив несберегаемого типа. Естественно, компилятор сообщает об ошибке *создания параметризованного массива*:

```
% javac WrongMicroArrayList_1.java
WrongMicroArrayList_1.java:4: generic array creation
    a = new T[c.size()]; // ошибка компиляции
        ^
1 error
```

Явное создание параметризованного массива запрещено, потому что это привело бы к расхождению между объявленным типом и сбереженным типом массива. Мы обсудим эту проблему на примере в следующем разделе.

Создание параметризованного массива – один из типичных подводных камней: вы можете столкнуться с ним, когда используете параметризованный тип для создания массива. Рассмотрим, например, следующий (неправильный) код, который должен был бы вернуть массив, содержащий два списка:

org/jgcbook/chapter05/E_generic_array_creation/IndecentExposure

```
class IndecentExposure {
  public static List<Integer>[] twoLists() {
    List<Integer> a = List.of(1, 2, 3);
    List<Integer> b = List.of(4, 5, 6);
    return new List<Integer>[] {a, b}; // ошибка компиляции
  }
}
```

Это ошибка, потому что параметризованный тип не является сберегаемым. И на этот раз попытка откомпилировать код ведет к ошибке создания параметризованного массива:

```
% javac IndecentExposure.java
IndecentExposure.java:6: generic array creation
    return new List<Integer>[] {a, b}; // ошибка компиляции
                               ^
1 error
```

Мы обсудим эту проблему в разделе «Принцип непристойного обнажения» ниже в этой главе.

Неспособность создавать параметризованные массивы – одно из самых серьезных ограничений Java. Поскольку оно вызывает такое раздражение, не лишним будет еще раз повторить, почему оно возникает: обобщенные массивы вызывают проблемы, потому что обобщенные типы реализованы посредством стирания, но стирание необходимо, чтобы обеспечить преемственность и развитие кода.

Самый лучший обходной путь – использовать `ArrayList` или какой-то другой класс из каркаса коллекций вместо массива. Мы обсудили компромиссы между классами коллекций и массивами в разделе «Массивы» главы 2 и заметили, что во многих случаях коллекции предпочтительнее: для них компилятор отлавливает больше ошибок, они предлагают больше операций и большую гибкость представления. Часто лучшее решение проблем, связанных с массивами, – «просто сказать нет» и использовать коллекции.

Но иногда это не работает, потому что массив необходим из соображений совместимости или эффективности. Примеры встречаются и в каркасе коллекций: ради совместимости метод `toArray` преобразует коллекцию в массив, а ради эффективности класс `ArrayList` (наряду со многими другими) реализован посредством хранения элементов списка в массиве. В следующих разделах мы подробно обсудим оба этих случая, а также связанные с ними подводные камни и принципы, помогающие их избежать: принцип правдивости рекламы и принцип непристойного обнажения. Мы также рассмотрим проблемы, возникающие в связи с vararg-аргументами и созданием параметризованного массива.

Принцип правдивости рекламы

В предыдущем разделе мы видели, что наивное создание массива с помощью типа параметризованной коллекции не работает. Первая попытка исправления – создать массив `Object` и привести ссылку на него к типу `T[]`. Эта версия `toArray` компилируется, поэтому мы можем инициализировать класс какими-нибудь тривиальными тестовыми данными и добавить метод `main` для тестирования `toArray`.

org/jgcbook/chapter05/F_the_principle_of_truth_in_advertising/WrongMicroArrayList_2

```
class WrongMicroArrayList_2<E> {
  private int size = 2;               // тестовые данные
  private Object[] data = {"a", "b"}; // тестовые данные
  ...
  public <T> T[] toArray(T[] a) {
    a = (T[])new Object[size];        // непроверенное приведение
    System.arraycopy(data, 0, a, 0, size);
    return a;
  }
  public static void main(String[] args) {
    var wma = new WrongMicroArrayList_2<>();
    String[] a = wma.toArray(args); // исключение приведения класса
  }
}
```

Компилятор выдает предупреждение о непроверенном приведении:

```
% javac -Xlint WrongMicroArrayList_2.java
WrongMicroArrayList_2.java:4: warning: [unchecked] unchecked cast
    a = (T[])new Object[size]; // непроверенное приведение
             ^
  required: T[]
  found:    Object[]
  where T is a type-variable:
    T extends Object declared in method <T>toArray(Collection<T>)
1 warning
```

Бесшабашный разработчик, который проигнорирует это предупреждение, получит следующий результат при запуске программы:

```
% java WrongMicroArrayList_2
Exception in thread "main" java.lang.ClassCastException: class [Ljava.lang.
Object; \
cannot be cast to class [Ljava.lang.String; ([Ljava.lang.Object; and \
[Ljava.lang.String; are in module java.base of loader 'bootstrap')
        at WrongMicroArrayList_2.main(WrongMicroArrayList_2.java:10)
```

Фразой `[Ljava.lang.Object` виртуальная машина Java обозначает сбереженный тип массива, где `[L` говорит о том, что это массив ссылочного типа, а `java.lang.Object` – тип элементов массива. Сообщение в исключении приведения класса ссылается на строку, содержащую вызов `toArray`. Это сообщение может привести в недоумение, потому что указанная строка, на первый взгляд, не содержит никакого приведения!

Чтобы увидеть, что не так с этой программой, посмотрим, как она транслируется с использованием стирания. Стирание отбрасывает параметр типа в объявлении коллекции, заменяет вхождения переменной-типа `T` на `Object` и вставляет подходящее приведение в обращение к `toArray`, так что в итоге получается такой эквивалентный код:

```
org/jgcbook/chapter05/F_the_principle_of_truth_in_advertising/
WrongMicroArrayList_2_Erased
class WrongMicroArrayList_2_Erased {
  private int size = 2;                  // тестовые данные
  private Object[] data = {"a", "b"};    // тестовые данные
  ...
  public Object[] toArray(Object[] a) {
    a = (Object[])new Object[size];      // непроверенное приведение
    System.arraycopy(data, 0, a, 0, size);
    return a;
  }
  public static void main(String[] args) {
    var wma = new WrongMicroArrayList_2_Erased();
    String[] arr = (String[])wma.toArray(args); // исключение приведения
класса
  }
}
```

Во время выполнения первое из этих приведений завершается успешно. Но хотя данные коллекции неважно какого типа были скопированы в массив, возвращенный из `toArray`, сбереженным типом элементов этого массива является `Object`, поэтому второе приведение не проходит.

Чтобы избежать этой проблемы, необходимо придерживаться следующего принципа:

принцип правдивости рекламы: сбереженный тип массива должен быть подтипом результата стирания его статического типа.

Этот принцип соблюдается в теле самого метода `toArray`, где результатом стирания `T` является `Object`, но не в методе `main`, где `T` привязан к `String`, но сбереженным типом массива все равно является `Object`.

Прежде чем мы научимся создавать массивы в соответствии с этим принципом, необходимо подчеркнуть еще один момент. Напомним гарантию успешного приведения, которая прилагается к дженерикам в Java: никакая операция приведения, вставленная в результате стирания, никогда не завершается ошибкой, коль скоро не было выдано предупреждений о невозможности проверки. Сформулированный выше принцип иллюстрирует обратное: если имеются предупреждения о невозможности проверки, то приведение, вставленное при стирании, может завершиться ошибкой. Более того, *приведение, завершающееся ошибкой, может находиться совсем не в той части исходного кода, которая несет ответственность за предупреждение*! Поэтому код, порождающий предупреждения о невозможности проверки, следует писать с сугубой осторожностью.

Как создавать массивы

Во всех наших предыдущих попытках создавать массивы принимали участие выражения создания массивов, т. е. выражения вида `new Type[length]` или `new Type[]`

{...}. Эти попытки терпели неудачу, потому что выражения создания массивов требуют, чтобы на этапе компиляции был предоставлен сберегаемый тип, а единственный тип, который можно получить от параметризованной коллекции на этапе компиляции, сберегаемым не является. Вместо этого мы должны предоставить тип массива на этапе выполнения. Для этого обратимся к другому способу создания массивов: рефлексии.

Чтобы создать массив посредством рефлексии, необходимо знать только его длину и тип. Они предоставляются в виде `int` и *маркера типа* – объекта, представляющего сбереженный тип. В двух следующих подразделах мы исследуем различные возможные способы получения маркеров типа с целью создания массивов.

Массив рождает массив

«Деньги рождают деньги», – сказал Томас Фуллер (1732), заметив, что один из способов получить деньги – уже иметь их. Следуя авторитетному мнению Фуллера, мы можем создать новый массив из существующего, скопировав маркер типа последнего. Ниже показан наш следующий подход к реализации `ArrayList` с использованием этого технического приема.

org/jgcbook/chapter05/G_how_to_create_arrays/MicroArrayList_v1

```
class MicroArrayList_v1<E> {
  private int size = 2; // test data
  private Object[] data = {"a", "b"}; // тестовые данные
  ...
  @SuppressWarnings("unchecked")
  public <T> T[] toArray(T[] a) {
    a = (T[])Array.newInstance(a.getClass().getComponentType(), size);
    System.arraycopy(data, 0, a, 0, size);
    return a;
  }
  public static void main(String[] args) {
    MicroArrayList_v1<String> mal = new MicroArrayList_v1<>();
    String[] arr = mal.toArray(args);
    assert Arrays.equals(arr, new String[]{"a", "b"});
  }
}
```

Выделенное выражение, которое мы скоро рассмотрим во всех деталях, возвращает маркер типа, скопированный из массива, переданного `toArray`. Затем этот маркер вместе с размером коллекции подается на вход методу `java.lang.reflect.Array::newInstance`, который выделяет память для нового массива. Результат вызова `newInstance` имеет тип `Object`[1], поэтому для приведения к правильному типу `T[]` требуется непроверенное приведение. Тестовый код вызывает метод `toArray`, передавая ему ссылку на массив `args`; `toArray` игнорирует содержимое `args` и возвращает массив строк, содержащий элементы коллекции, которая в данном случае была передана во время инициализации.

[1] Почему `Array::newInstance` возвращает результат типа `Object`, а не `Object[]`? Потому что `newInstance` может возвращать массив примитивного типа, например `int[]`, который является подтипом `Object`, а не `Object[]`. Здесь, впрочем, такого не происходит, потому что переменная-тип `T` должна обозначать ссылочный тип.

Классная альтернатива

Бывает, что иметь деньги кажется единственным способом деньги получить. Для массивов это не вполне верно: альтернатива – использовать маркер типа из какого-то другого источника, например, экземпляра класса `java.lang.Class<T>`. Смысл этой параметризации интуитивно не очевиден: ссылки на объекты этого типа, если они получены определенным способом, содержат информацию этапа компиляции о представляемом ими классе этапа выполнения. Например, выражение `MyType.class`, в котором используется синтаксис литерала класса, имеет *статический* тип `Class<MyType>`, поэтому ссылка на него содержит информацию о типе, полезную на обоих этапах, компиляции и выполнения. Сколько именно информации о статическом типе содержит ссылка `Class`, зависит от того, как она была получена. Ниже перечислено три способа получить ссылки на объекты классов, при этом объем информации о статическом типе варьируется: от никакой до полного типа.

- В классе `java.lang.Class` объявлен метод `getComponentType`, который мы уже встречали в примере класса `MicroArrayList_v1`:

  ```
  public Class<?> getComponentType() {...}
  ```

  Если вызвать этот метод для класса массива, то он вернет ссылку, соответствующую типу элементов массива. (При вызове для любого другого класса он вернет `null`.) Тип возвращаемого значения, `Class<?>`, говорит, что этот метод не может предоставить никакой информации этапа компиляции о возвращенном им объекте класса.

- В классе `MicroArrayList_v1` также был использован метод `Object::getClass`, объявленный как

  ```
  public <T> Class<?> getClass() {...}
  ```

  но этот метод обрабатывается компилятором специальным образом: в спецификации языка Java сказано, что «фактическим типом результата является `Class<? extends |X|>`, где `|X|` – результат стирания статического типа выражения, от имени которого вызван `getClass`» (Gosling et al. 2023, § 4.3.2).

  Лучше всего проиллюстрировать это положение на примере, приведенном в официальной документации по классу `Object`. Этот код компилируется и выполняется без приведения:

  ```
  Number num = 0;
  Class<? extends Number> clazz = num.getClass();
  ```

  При компиляции второй строчки известно, что объект, на который ссылается `num`, принадлежит какому-то подтипу `Number`. Когда на этапе выполнения производится обращение к `getClass`, возвращается ссылка, соответствующая типу этого объекта, т. е. некоторому конкретному подтипу `Number`. Следовательно, эту информацию можно включить в статический тип `Class<? extends Number>` ссылки на объект класса.

- Выражение, в котором используется синтаксис литерала класса `MyType.class`, имеет точный статический тип `Class<MyType>`. Поэтому следующий код компилируется и исполняется без приведения:

```
Class<Number> clazz = Number.class;
```

Мы можем определить вариант метода `toArray`, который принимает в качестве маркера типа ссылку на тип `Class<T>`, а не на массив типа `T[]`:

org/jgcbook/chapter05/G_how_to_create_arrays/MicroArrayList_v2

```
class MicroArrayList_v2<E> {
  private int size = 2;                  // тестовые данные
  private Object[] data = {"a", "b"};    // тестовые данные
  ...
  @SuppressWarnings("unchecked")
  public <T> T[] toArray(Class<T> clazz) {
    T[] a = (T[])Array.newInstance(clazz, size);
    System.arraycopy(data, 0, a, 0, size);
    return a;
  }
  public static void main(String[] args) {
    MicroArrayList_v2<String> mal = new MicroArrayList_v2<>();
    String[] arr = mal.toArray(String.class);
    assert Arrays.equals(arr, new String[]{"a", "b"});
  }
}
```

Теперь метод `toArray` получает ссылку на `String.class`, а не на `String[]` и использует ее для создания массива элементов этого сбереженного типа, так что, в принципе, `Array::newInstance` мог бы вернуть точный тип массива. К сожалению, приведение по-прежнему необходимо, потому что, как было объяснено ранее, типом возвращаемого значения является `Object`, чтобы можно было работать с массивами примитивных типов.

Как определить ArrayList

Теперь у нас есть все, чтобы понять, как работает `ArrayList`. В примере 5.2 приведена сильно упрощенная – всего на 48 строчек – версия реализации `ArrayList` в OpenJDK, насчитывающей 1800 строчек. Этот пример поможет вам решить, что делать в ситуации, когда ради эффективности или совместимости необходимо использовать массивы, но при этом поддержать преобразование в коллекции и обратно. Такой код следует писать осторожно, поскольку он по необходимости включает непроверенные приведения и должен соблюдать принцип правдивости рекламы.

В примере 5.2 в конце этого раздела демонстрируется миниатюрная версия `ArrayList`, названная нами `MicroArrayList`, производная от `AbstractList`. Этот класс является «заготовкой реализации» для списков (более подробно мы будем обсуждать заготовки реализаций в главе 17). `AbstractList` объявляет или наследует абстрактные методы `get` и `size` и объявляет методы `set`, `add` и `remove`, которые возбуждают исключение `UnsupportedOperationException` и, следовательно, должны быть переопределены.

С другой стороны, первый и третий унаследованные перегруженные варианты `toArray`, с которыми мы встречались в разделе «Параметризованные типы и массивы» выше, поддерживаются, но перегружены в классе `ArrayList` (и `MicroArrayList`) в интересах эффективности. Кроме того, любой список на

основе массива, каковым, в частности, является `ArrayList`, должен реализовать маркерный интерфейс `RandomAccess`, сообщающий обобщенным алгоритмам, что обработка списка с помощью `get` быстрее, чем использование итератора по нему.

Класс представляет список элементов типа `E` двумя закрытыми полями: `size` типа `int`, в котором хранится длина списка, и `arr` типа `Object[]`, в котором хранятся элементы списка. Длина массива должна быть не меньше `size`, но в конце могут быть дополнительные неиспользуемые элементы.

Память для новых экземпляров базового массива выделяется в двух местах: в конструкторе класса и в методе `add`, если емкость массива недостаточна и должен быть создан новый массив с помощью вспомогательного метода `java.util.Arrays::copyOf`. У этого метода два перегруженных варианта (здесь они немного упрощены):

org/jgcbook/chapter05/H_how_to_define_arraylist/Arrays

```
public class Arrays {
  ...
  @SuppressWarnings("unchecked")
  public static <T> T[] copyOf(T[] original, int newLength) {
    return (T[]) copyOf(original, newLength, original.getClass());
  }
  public static <T,U> T[] copyOf(U[] original, int newLength,
      Class<? extends T[]> newType) {
    @SuppressWarnings("unchecked")
    T[] copy = (T[]) Array.newInstance(newType.getComponentType(), new-
Length);
    System.arraycopy(original, 0, copy, 0, Math.min(original.length, new-
Length));
    return copy;
  }
  ...
}
```

Первый из этих методов, вызываемый из `MicroArrayList::add`, принимает массив, выделяет маркер типа для массива (*не* для его элемента) и передает его второму методу (который также вызывается из `toArray`). Затем этот метод выделяет маркер типа для элемента массива (выделенный код) и передает его методу `Array::newInstance`, очень похоже на то, как было сделано в `MicroArrayList_v2`. Об этих служебных методах полезно знать всегда, когда вы хотите скопировать массив, а не только в этом случае; их использование устраняет необходимость использовать в своем коде рефлексию напрямую и в какой-то степени избавляет от сомнений по поводу непроверенных приведений и следующих за ними предупреждений компилятора.

Предыдущие разделы подготовили нас к реализации `toArray`. Она очень похожа на версию `MicroArrayList_v1`, только вместо вызова метода `Array.newInstance` непосредственно используется вспомогательный метод из `Arrays`. В этой версии наконец-то отброшено упрощающее предположение о том, что память для массива всегда выделяется заново, она способна повторно использовать переданный массив, если он достаточно длинный, чтобы вместить все элементы коллекции.

Обоснованием этой возможности служит желание разрешить повторное использование одного и того же массива в роли аргумента `toArray`, чтобы из-

бежать выделения новой памяти при каждом обращении. В этой ситуации если размер коллекции меньше длины массива, то код обработки элементов массива должен как-то обнаруживать, что данные закончились. Для этой цели метод `toArray` вставляет после данных, скопированных из коллекции, маркер конца `null`.

Пример 5.2. *ArrayList в миниатюре*

org/jgcbook/chapter05/H_how_to_define_arraylist/MicroArrayList

```
class MicroArrayList<E> extends AbstractList<E> implements RandomAccess {
  Object[] data;
  private int size;
  public MicroArrayList() { data = new Object[10]; }
  public int size() { return size; }
  public E get(int i) {
    Objects.checkIndex(i,size);
    return (E)data[i]; // непроверенное приведение
  }
  public E set(int i, E elt) {
    Objects.checkIndex(i,size);
    E old = (E)data[i]; // непроверенное приведение
    data[i] = elt;
    return old;
  }
  public void add(int i, E elt) {
    Objects.checkIndex(i,size+1);
    if (size + 1 > data.length) {
      // игнорировать возможность переполнения
      data = Arrays.copyOf(data, data.length + (data.length << 1));
    }
    System.arraycopy(data, i, data, i+1, size-i);
    data[i] = elt;
    size++;
  }
  public E remove(int i) {
    Objects.checkIndex(i,size);
    E old = (E)data[i]; // непроверенное приведение
    size--;
    System.arraycopy(data, i+1, data, i, size-i);
    data[size] = null; // освободить последний элемент для потенциальной
сборки
                       // мусора
    return old;
  }
  public <T> T[] toArray(T[] a) {
    if (a.length < size)
      return (T[])Arrays.copyOf(data, size, a.getClass()); // непроверенное
                                                           // приведение
    System.arraycopy(data, 0, a, 0, size);
    if (a.length > size) a[size] = null; // маркер конца
    return a;
  }
  public Object[] toArray() { return Arrays.copyOf(data, size); }
}
```

Принцип непристойного обнажения

Хотя *создавать* массив с несберегаемым типом элементов считается ошибкой, *объявлять* массив с таким типом или выполнять непроверенное приведение к такому типу разрешено. Этими возможностями следует пользоваться очень осторожно, и полезно понимать, что плохого может случиться при ненадлежащем использовании. В частности, библиотека *никогда* не должна раскрывать наружу массив элементов несберегаемого типа.

В разделе «Массивы» главы 2 приведен пример, показывающий, почему сбережение необходимо.

org/jgcbook/chapter05/I_the_principle_of_indecent_exposure/Program_1

```
Integer[] ints = {0};
Number[] nums = ints;
nums[0] = 3.14; // исключение сохранения в массиве
int n = ints[0];
```

Здесь массиву целых присваивается массив чисел, а затем предпринимается попытка сохранить в этом массиве число с двойной точностью. Эта попытка приводит к исключению сохранения в массиве из-за проверки сбереженного типа. И это правильно, потому что иначе в последней строчке мы попытались бы сохранить число с двойной точностью в целой переменной.

Вот еще похожий пример, в котором массивы чисел заменены массивами *списков* чисел:

org/jgcbook/chapter05/I_the_principle_of_indecent_exposure/Program_2

```
List[] intListArray = { List.of(0) };
List<Integer>[] intLists = (List<Integer>[])intListArray; // непроверенное
приведение
List<? extends Number>[] numLists = intLists;
numLists[0] = List.of(3.14);
int n = intLists[0].get(0); // исключение приведения класса!
```

Здесь массив списков целых присваивается массиву списков чисел, а затем производится попытка сохранить список чисел с двойной точностью в массиве списков чисел. Хотя и эта попытка должна бы завершиться неудачей, этого не происходит, потому что проверка сбереженного типа неадекватна: сбереженная информация содержит только результат стирания типа, показывающий, что это массив списков `List`, а не массив списков `List<Integer>`. Поэтому сохранение завершается успешно, и программа неожиданно «падает» где-то в другом месте.

В примере 5.3 показан аналогичный код, разбитый на два класса, чтобы продемонстрировать, как плохо спроектированная библиотека может создавать проблемы ни в чем не повинному клиенту.

Пример 5.3. *Избегайте массивов несберегаемого типа*

org/jgcbook/chapter05/I_the_principle_of_indecent_exposure/DeceptiveLibrary

```
// DeceptiveLibrary.java:
public class DeceptiveLibrary {
  public static List<Integer>[] createIntLists(int size) {
    List<Integer>[] intLists =
      (List<Integer>[]) new List[size]; // непроверенное приведение
```

```
    for (int i = 0; i < size; i++)
      intLists[i] = List.of(i+1);
    return intLists;
  }
}
// InnocentClient.java:
class InnocentClient {
  public static void main(String[] args) {
    List<Integer>[] intLists = DeceptiveLibrary.createIntLists(1);
    List<? extends Number>[] numLists = intLists;
    numLists[0] = List.of(1.01);
    int i = intLists[0].get(0); // исключение приведения класса!
  }
}
```

В первом классе, названном `DeceptiveLibrary`, определен статический метод, который возвращает массив списков целых заданного размера. Поскольку создание параметризованных массивов запрещено, создается массив элементов простого типа `List`, а чтобы придать элементам параметризованный тип `List<Integer>`, выполняется непроверенное приведение. Выделенный код приводит к предупреждениям по поводу использования простого типа и по поводу непроверенного приведения:

```
% javac -Xlint:unchecked DeceptiveLibrary.java
DeceptiveLibrary.java:9: warning: [rawtypes] found raw type: List
        (List<Integer>[]) new List[size]; // unchecked cast
                          ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
DeceptiveLibrary.java:9: warning: [unchecked] unchecked cast
        (List<Integer>[]) new List[size]; // unchecked cast
                          ^
  required: List<Integer>[]
  found:    List[]
2 warnings
```

Поскольку массив в действительности является массивом списков целых, приведения выглядят разумно, и вы могли бы решить, что игнорировать это предупреждение безопасно. Но, как мы увидим, вы делаете это на свой страх и риск!

Второй класс, `InnocentClient`, содержит метод `main`, похожий на метод из предыдущего примера. Поскольку непроверенное приведение находится в библиотеке, при компиляции не выдается никакого предупреждения о невозможности проверки. Однако при выполнении этого кода список целых перезаписывается списком чисел с двойной точностью. А попытка прочитать целое из массива списков целых приводит к ошибке приведения, неявно вставленного в процессе стирания типа:

```
%java InnocentClient
Exception in thread "main" java.lang.ClassCastException: class java.lang.Dou-
ble \
cannot be cast to class java.lang.Integer \
(java.lang.Double and java.lang.Integer are in module java.base of loader \
'bootstrap')
    at InnocentClient.main(InnocentClient.java:8)
```

Как и в предыдущем разделе, это сообщение об ошибке может вызвать ступор, потому что эта строка, на первый взгляд, не содержит никакого приведения!

Во избежание этой проблемы непременно соблюдайте следующий принцип:

> *принцип непристойного обнажения*: никогда не раскрывайте публично массив, элементы которого имеют несберегаемый тип.

Это еще один случай, когда непроверенное приведение в одной части программы может привести к ошибке приведения класса совсем в другой части, где приведение находится не в исходном коде, а сгенерировано в процессе стирания. Поскольку такие ошибки очень трудно отлаживать, непроверенные приведения следует использовать с сугубой осторожностью.

Принципы правдивости рекламы и непристойного обнажения тесно связаны. Первый требует, чтобы тип массива на этапе выполнения был надлежащим образом сбережен, а второй – чтобы тип массива на этапе компиляции был сберегаемым.

Интересный пример нарушения принципа непристойного обнажения имеется в библиотеке рефлексии:

```
TypeVariable<Class<T>>[] java.lang.Class.getTypeParameters()
TypeVariable<Method>[] java.lang.reflect.Method.getTypeParameters()
```

Следуя описанному выше образцу, нетрудно создать свою собственную версию класса `InnocentClient`, которая возбуждает исключение приведения класса в точке, где никакого приведения нет и в помине, а роль `DeceptiveLibrary` в этом случае будет играть официальная библиотека Java. Мы обсуждали компромиссы этого дизайна в разделе «Массивы» главы 24.

Создание массива и vararg-аргументы

Удобная нотация vararg-аргументов дает методам возможность принимать переменное число аргументов, упакованных в массив, как обсуждалось в разделе «Параметризованные методы и vararg-аргументы» главы 1. Однако эта нотация не так удобна, как хотелось бы, потому что если vararg-аргумент параметризованный, то в результате должен быть создан параметризованный массив со всеми вытекающими проблемами.

Например, рассмотрим такой код:

org/jgcbook/chapter05/J_array_creation_and_varargs/Varargs

```
class Varargs {
  static <T> T[] createGenericArray(T... args) {
   return args;
  }
}
```

Как следует из названия, метод `createGenericArray` ни во что не ставит запрет, который мы обсуждали в разделе «Создание параметризованного массива» выше. Обращения к этому методу могут привести к плачевным последствиям. Например, легко использовать его для воспроизведения проблемы, с которой призван бороться принцип непристойного обнажения, обсуждавшийся в предыдущем разделе.

org/jgcbook/chapter05/J_array_creation_and_varargs/Varargs

```
List<Integer>[] intLists = Varargs.createGenericArray(List.of(1));
List<? extends Number>[] numLists = intLists;
numLists[0] = List.of(3.14);    // загрязнение кучи
int n = intLists[0].get(0);     // исключение приведения класса
```

Термином «загрязнение кучи» описывают ситуацию, когда переменная параметризованного типа ссылается на объект другого типа. В данном случае переменной `numLists[0]` – той же, что `intLists[0]`, – присвоено значение типа `List<Double>`, хотя объявлена она была как `List<Integer>`. Невидимое приведение, вставленное компилятором в последнюю строчку, в этом случае приводит к ошибке.

По этой причине компиляция класса `Varargs` порождает два предупреждения: о создании параметризованного массива и о загрязнении кучи:

```
% javac -Xlint Varargs.java
Varargs.java:2: warning: [unchecked] Possible heap pollution from parameter-
ized \
vararg type T
    static <T> T[] createGenericArray(T... args) {
                                      ^
  where T is a type-variable:
    T extends Object declared in method <T>createGenericArray(T...)
```

Попытка явно создать параметризованный массив сразу приводит к ошибке, но такое неявное создание разрешено, хотя и с предупреждением, потому что методы с vararg-аргументами очень полезны на практике (`Arrays::asList` – только один пример; другие мы встретим в главе 16). Как и другие предупреждения о невозможности проверки, предупреждение о создании параметризованного массива нарушает гарантию успешного приведения, сопровождающую дженерики. Нетрудно на основе приведенных выше примеров сконструировать катастрофу в результате игнорирования предупреждения о невозможности проверки, а также создать аналогичный пример, в котором вместо этого предупреждения выдается предупреждение о создании параметризованного массива.

Предупреждение компилятора говорит только о *возможном* загрязнении кучи, потому что методы с параметризованными vararg-аргументами необязательно несут опасность. Например, если бы метод `createGenericArray` просто печатал переданные ему аргументы, а не возвращал созданный массив, то его вызов не мог бы привести к загрязнению кучи:

```
static <T> void createGenericArray(T... args) {
  for (T elem : args) System.out.println(elem);
}
```

Тем не менее компилятор все же выдает предупреждение о загрязнении кучи в месте объявления метода и предупреждения о создании параметризованного массива во всех местах, где метод вызывается. Если в действительности метод типобезопасен – как, например, `Arrays::asList`, – то предупреждения, выдаваемые при его вызовах, могут сильно раздражать. Начиная с Java 7, эта проблема была решена введением аннотации метода `@SafeVarargs`, которая подавляет предупреждения компилятора как в месте объявления метода, так и в местах его вызова. Но `@SafeVarargs` никак не гарантирует типобезопасность, это

лишь заявление о типобезопасности метода, на которое вызывающая сторона может положиться. А как мы можем быть уверены, применяя эту аннотацию к методу с параметризованными vararg-аргументами, в том, что он не может вызвать загрязнение кучи? Есть две потенциальные опасности.

- Одну мы уже видели: первая версия `createGenericArray` возвращала ссылку на параметризованный массив, который компилятор создает для упаковки vararg-аргументов. Это было прямое нарушение принципа непристойного обнажения, и оно относится к любому методу, который выставляет параметризованный массив на обозрение недоверенному коду. (Раскрытие массива доверенному коду – например, передача другому методу с аннотацией `@SafeVarargs` – безопасно).
- Пример второго способа конструирования метода с vararg-аргументами дает включение кода, который мы использовали для вызова `createGenericArray`, в сам метод:

```
static <T> void createGenericArray(List<Integer>... intLists) {
  List<? extends Number>[] numLists = intLists;
  numLists[0] = List.of(3.14);   // загрязнение кучи
  int n = intLists[0].get(0);    // исключение приведения класса
}
```

  В этом случае методу даже не нужно раскрывать массив `List<Integer>` недоверенному коду; он сам создал опасность, сохранив в нем значение типа `List<Double>`. Вообще, сохранение значений в массиве параметризованных vararg-аргументов небезопасно относительно типов.

Используя `@SafeVarargs`, мы можем безопасно писать методы с параметризованными vararg-аргументами, не создавая неудобств вызывающей стороне, но только при условии, что избежим этих двух опасностей. Заметим, что эта аннотация применима только к конструкторам и методам, не допускающим переопределения, – иными словами, закрытым, статическим или финальным.

Вместо неявного массива vararg-аргументов можно воспользоваться явным параметром `List`:

```
static <T> void useListInstead(List<? extends T> args) {
  for (T elem : args) System.out.println(elem);
}
```

Эта альтернатива стала более привлекательной с тех пор, как в версии Java 9 появился фабричный метод `List::of`, который позволяет вызывающей стороне удобно обернуть аргументы метода списком. Например:

```
useListInstead(List.of(1, 3.14, "a")));
```

Для любого числа аргументов, не большего 10, `List::of` имеет перегруженный вариант, обрабатывающий точное число аргументов. На случай, когда аргументов больше 10, имеется перегруженный вариант с vararg-аргументом. Отметим, что эта идиома удобна только потому, что стало возможно снабдить этот перегруженный вариант аннотацией `@SafeVarargs`. В первом издании этой книги, вышедшем, когда текущей была версия Java 6, мы рекомендовали никогда не использовать vararg-аргументы с несберегаемым типом. Нынешняя ситуация – пример неуклонного улучшения языка Java: `SafeVarargs` была добавлена в Java 7, а семейство фабричных методов `List::of` – в Java 9.

Где требуются сберегаемые типы

Ниже приведен перечень мест, где сберегаемые типы требуются или рекомендуются.

- Выполнять приведение обычно следует к сберегаемому типу. (Приведение к несберегаемому типу, как правило, влечет за собой предупреждение о невозможности проверки).
- Класс, расширяющий `Throwable`, не должен быть параметризованным.
- При явном создании экземпляра массива тип элементов должен быть сберегаемым.
- Сбереженный тип массив должен быть подтипом результата стирания его статического типа (см. принцип правдивости рекламы), а публично раскрываемый массив должен иметь сберегаемый тип (см. принцип непристойного обнажения).
- Метод с vararg-аргументами не должен ни раскрывать параметризованный массив недоверенному коду, ни сохранять в нем значения. Все такие методы должны быть снабжены аннотацией `@SafeVarargs`.

Эти ограничения вытекают из того факта, что параметризованные типы реализованы посредством стирания и рассматривать их нужно как цену, которую необходимо заплатить за легкость развития кода (см. приложение).

Для полноты картины мы добавим в этот перечень еще одно ограничение, связанное с отражением.

- Маркеры типов соответствуют сберегаемым типам, и параметр-тип в `Class<T>` должен быть сберегаемым. Эти ограничения обсуждаются в разделе «Рефлектированные типы являются сберегаемыми» главы 6.

О дизайне дженериков в Java

Простор для проектирования дженериков был огромным, поэтому решения проектировщиков породили многочисленные споры – некоторые не стихают по сей день. Так как многие (хотя и не все) вращаются вокруг вопроса о стирании, мы завершим эту главу обсуждением некоторых вопросов, все еще возникающих в связи с дизайном дженериков, и рассмотрим кое-какие потенциальные альтернативы.

Стирание

Стирание, безусловно, является самой спорной особенностью дизайна параметризованных типов. Так повелось с их появления в версии Java 5 и продолжается по сей день. Как следует из этой главы, вызванные этим трудности никуда не делись, хотя первоначальное обоснование – простота миграции старого кода (см. приложение) – уже не актуально. Поэтому так как возможность изменить дизайн, перейдя на сбережение типов, технически остается, имеет смысл обсудить плюсы и минусы стирания.

В разделе «Параметризованные типы и шаблоны» главы 1 мы сравнили расширение шаблонов в C++ со стиранием типов в Java. Стратегия расширения, которая отображает параметризованный класс в новый класс при каждой конкретизации, обеспечивает большую типобезопасность (так как

каждую конкретизацию можно подвергнуть проверке типов отдельно после расширения) и открывает возможность для специализированных оптимизаций. Но за эти преимущества приходится расплачиваться разбуханием кода и невозможностью абстрагировать семейства параметрических типов – в отличие, например, от джокеров в Java. Ранний эксперимент по сравнению расширения шаблонов со стиранием типов в прототипе компилятора Java (Odersky et al. 2000) стал серьезным аргументом против дальнейших исследований в этом направлении.

Другим способом избежать стирания могло бы стать сбережение типов. Это устранило бы некоторые трудности, рассмотренные в настоящей главе, например, разрешив тесты экземпляров, тесты загрязнения кучи и такие оптимизации кода, как специализированное размещение в памяти, производимые на этапе выполнения. Поставленный в работе Goetz (2020) вопрос звучит так: если бы внедрить схему сбережения сейчас, то перевесил бы выигрыш затраты? При этом учитываются:

- затраты на этапе выполнения в форме увеличенного статического и динамического потребления памяти, увеличенных затрат на загрузку классов, увеличенных затрат на своевременную (JIT) компиляцию и повышенной нагрузки на кеш кода;
- потенциальные проблемы в JVM и языковых экосистемах, которые зависят от соглашений между несколькими поставщиками JVM и разработчиками языков, работающих на JVM;
- необходимость обширных модификаций JVM (хотя их влияние на исходный код было бы ограничено), чтобы можно было обрабатывать код со сбережением и без сбережения типов и избежать проблем двоичной совместимости.

В работе Goetz (2020) эти вопросы рассмотрены весьма детально и сделан вывод, что стирание типов было наилучшим инженерным компромиссом в момент включения параметризованных типов в язык и остается таковым и сегодня.

Неограниченные джокеры

Элементы создаваемых массивов должны иметь сберегаемый тип, поэтому проектировщики постарались сделать понятие сберегаемого типа максимально общим, чтобы свести неудобства этого ограничения к минимуму. Если бы они были готовы ограничить понятие сбереженного типа, то могли бы упростить себе задачу, исключив типы с неограниченными джокерами, например `List<?>`. Но если бы они так поступили, то сберегаемый тип стал бы синонимом непараметризованного типа (т. е. примитивные типы, простые типы и типы, объявленные без параметра-типа). Это было бы совместимо с синтаксисом литералов классов, поскольку `List.class` разрешен, а `List<?>.class` запрещен.

Массивы

Одна из причин сложностей дизайна дженериков Java – необходимость обеспечить хорошую поддержку массивов. Альтернативный дизайн был бы проще, хотя использовать массивы было бы не так удобно. Посмотрим, как он мог бы выглядеть.

- Хотя создавать разрешено только массивы с элементами сберегаемого типа, допускается объявлять массив несберегаемого типа или приводить к несберегаемому типу массива, правда, ценой появления предупреждения о невозможности проверки в каком-то месте кода. Как мы видели, такие предупреждения нарушают гарантию успешного приведения, прилагаемую к параметризованным типам, и могут стать причиной ошибок приведения классов даже тогда, когда исходный код не содержит никаких приведений.

  Проще и, пожалуй, безопаснее было бы поставить ссылки на массив несберегаемого типа вне закона. Это означало бы, что объявить массив типа `E[]`, где `E` – переменная-тип, было бы невозможно.

  Однако это означало бы также, что нельзя было бы назначить параметризованный тип методу `toArray` (и аналогичным) для коллекций. Вместо

  ```
  public <T> T[] toArray(T[] arr)
  ```

  мы должны были бы писать

  ```
  public Object[] toArray(Object[] arr)
  ```

  и во многих случаях употребления этого метода пришлось бы явно приводить тип результата, что усложнило бы жизнь пользователям.

- Этот альтернативный дизайн потребовал бы также другой реализации vararg-аргументов, с использованием списков, а не массивов. До введения аннотации `@SafeVarargs` (см. раздел «Создание массива и vararg-аргументы» выше) реализация vararg-аргументов с помощью массива означала, что методы с несберегаемыми vararg-аргументами можно использовать с несберегаемым типом только ценой подавления предупреждений о невозможности проверки в каждом месте вызова. Но теперь, когда вызывать такие методы стало несравненно удобнее, преимущества реализации с помощью массивов проявились более наглядно. Например, компилятор может вывести наименее общий тип последовательности параметров и сделать этот тип доступным на этапе выполнения, как в следующем коде:

  ```
  class Varargs {
    @SafeVarargs
    static <T> Class<T> getVarargsType(T... args) {
      @SuppressWarnings("unchecked")
      Class<T> componentType = (Class<T>) args.getClass().getComponent-
  Type();
      return componentType;
    }
    public static void main(String[] args) {
      assert getVarargsType(3, 4, 5.5).equals(java.lang.Number.class);
      assert getVarargsType(3, 4, 5).equals(java.lang.Integer.class);
    }
  }
  ```

  В случае реализации vararg-аргументов с помощью параметризованных списков это было бы невозможно.

- То же изменение устранило бы необходимость в принципе непристойного обнажения. В конце раздела «Принцип непристойного обнажения» выше мы видели два метода в библиотеке рефлексии, которые нарушают этот принцип, чем вносят небезопасность в клиентский код:

  ```
  TypeVariable<Class<T>>[] java.lang.Class.getTypeParameters()
  TypeVariable<Method>[] java.lang.reflect.Method.getTypeParameters()
  ```

  Существуют аргументы как за, так и против такого проектного решения. Аргумент против понятен – мы, конечно, хотели бы избежать небезопасности относительно типов. Аргумент за заключается в том, что библиотека рефлексии, по историческим причинам, целиком основана на массивах, и класть в основу именно этих методов коллекции значило бы вводить в API специальный случай, который только всех запутывал бы.

  Другой способ взглянуть на этот вопрос примыкает к обсуждению системы типов коллекций в разделе «Не только статическая типизация» главы 18. Там мы рассматриваем систему типов как лишь один из инструментов, доступных проектировщику для объяснения смысла и контроля семантики API. Применив то же рассуждение здесь, мы могли бы заметить, что назначение API заключается в том, чтобы передать информацию о параметризованном типе на этап выполнения. Возврат массивов предоставляет эту информацию читателю. Мы не знаем о разумном сценарии, когда клиенту нужно было бы записывать в эти массивы, поэтому потенциально возможное загрязнение кучи вряд ли может стать проблемой на практике.

  Совершенной системы не бывает, и для каркаса коллекций мы убедимся в этом в части II (и подробно обсудим в разделе «Фундаментальные проблемы дизайна каркаса коллекций» главы 18. Это рассуждение наталкивает на мысль рассматривать корректность системы типов как лишь одну из движущих сил, пусть даже самую важную, определяющих дизайн API.

Arrays::copyOf

Проблемы, возникающие при выборе ограничений на параметры-типы этого метода, роднят его с прагматическим стилем анализа, выполненного в предыдущем разделе. Там мы видели, как стремление соблюсти баланс между безопасностью на этапе компиляции и удобством использования подтолкнуло проектировщиков к выбору небезопасного API. Вот еще один пример. Метод `Arrays::copyOf` объявлен так:

```
static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]>
newType)
```

Часто возникает вопрос, почему переменные-типы `U` и `T`, представляющие типы элементов исходного и конечного массива, независимы, т. е. ни один не ограничивает другой. В результате типы массивов могут оказаться несогласованными, что приведет к исключению `ArrayStoreException`.

Очевидное исправление – объявить `U` с границей `U extends T`. Это, конечно, предотвратило бы некоторые ошибки, но метод стал бы менее гибким. Если добавить такую границу, то метод можно было бы использовать, только чтобы скопировать

элементы массива в массив, для которого тип элементов является супертипом исходного. Например, можно было бы скопировать все элементы `Number[]` в новый массив `Object[]`.

Это действительно повысило бы типобезопасность, но ценой исключения другого сценария: копирования элементов массива в другой массив, для которого тип элементов является *подтипом* исходного. (Конечно, такое возможно, только если вызывающая сторона позаботилась о том, чтобы все элементы массива были согласованы с конечным типом.) Столкнувшись с такой ситуацией, вы будете рады, что метод копирования массивов поддерживает ее. Вот тривиальный пример, демонстрирующий этот сценарий:

```
Number[] orig = { 1, 2, 3 };
assert Arrays.stream(array).allMatch(e -> e instanceof Integer);
Integer[] copy = Arrays.copyOf(orig, 3, Integer[].class);
```

Это аналогично безопасному понижающему приведению индивидуального объекта после проверки типов с помощью `instanceof`, точно так же безопасность проистекает из проверки типов времени выполнения, а не статических. Рассуждение из предыдущего раздела применимо и здесь: статическая безопасность необязательно должна быть непререкаемым требованием при проектировании API.

Похожие соображения объясняют сигнатуру перегруженных вариантов `Arrays::sort`, в которых используется естественный порядок. В документации этих методов сказано, что все элементы должны реализовывать интерфейс `Comparable`, но объявления методов принимают `Object[]` в качестве аргумента:

```
static void sort(Object[] arr)
static void sort(Object[] arr, int fromIndex, int toIndex)
```

Массив `arr` мог бы содержать только элементы, реализующие `Comparable`, несмотря на то что типом элемента является `Object`. Тогда, как и в случае `Arrays::copyOf`, более либеральная схема типов, принятая проектировщиками, позволяет сортировать массив непосредственно, не копируя его предварительно в новый массив типа `Comparable[]`.

Заключение

В этой главе мы рассмотрели некоторые малоприятные последствия стратегии стирания, а именно сложность создания массива и принципы правдивости рекламы и непристойного обнажения, а также оценили наиболее важные проектные решения, сформировавшие облик дженериков в Java.

В следующей главе мы будем изучать рефлексию, что означает быть параметризованным для типа класса `Class<T>` и каким образом информация о параметризованном типе, будучи характеристикой времени компиляции, все же оказывается доступна на этапе выполнения с помощью рефлексии.

Глава 6

Рефлексия

Термином *рефлексия* называется набор средств, позволяющих программе исследовать свое собственное определение. В Java рефлексия используется в обозревателях классов, инспекторах объектов, отладчиках, интерпретаторах, таких службах, как JavaBeans™ и сериализация объектов, да и вообще в любом инструменте, который динамически создает, инспектирует или манипулирует произвольными объектами Java.

Рефлексия присутствовала в Java с самого начала, но появление параметризованных типов изменило ее в двух важных отношениях: привлекла параметризованные типы к рефлексии и рефлексию к параметризованным типам.

Под привлечением *параметризованных типов к рефлексии* мы понимаем, что некоторые типы, используемые для рефлексии, стали параметризованными. В частности, класс `Class` превратился в параметризованный класс `Class<T>`. Ранее мы встречались с параметризованными типами `Class` в разделе «Как создавать массивы» главы 5, где видели специальные технические приемы, которыми литералы классов и метод `getClass` класса `Object` пользуются для возврата более точной информации о типе. В этой главе мы увидим, как параметризованные типы используются с особенно большим успехом в аннотациях, и изучим следствия, вытекающие из того факта, что параметр-тип `T` в `Class<T>` может быть привязан только к сберегаемому типу. Кроме того, мы представим небольшую библиотеку, которая может помочь избежать многих типичных случаев непроверенных приведений.

Под привлечением *рефлексии к параметризованным типам* мы понимаем, что рефлексия возвращает информацию о параметризованных типах. Существуют интерфейсы для представления параметризованных типов, в том числе переменных-типов, параметров-типов и типов с джокерами, а также методы для получения параметризованных типов полей, конструкторов и методов.

В следующих разделах мы расскажем обо всем этом по очереди.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter06*.

Параметризованные типы для рефлексии

Центральным классом в API рефлексии является `Class`, который представляет информацию о типе объекта на этапе выполнения. В разделе «Как создавать массивы» главы 5 мы видели разные способы получить объект класса. Здесь нас будут интересовать два:

- можно вызвать метод `Object::getClass`, который возвращает ссылку типа `Class<? extends |X|>`, где `|X|` – результат стирания статического типа объекта, от имени которого вызван метод `getClass` получателя;
- можно написать литерал класса вида `MyType.class`, который возвращает ссылку типа `Class<MyType>`.

Ссылка типа `Class<T>` содержит как статическую информацию о типе, так и информацию времени выполнения: сам объект класса представляет тип времени выполнения, а `T` – статический тип.

Приведем пример:

org/jgcbook/chapter06/A_generics_for_reflection/Program_1

```
Class<Integer> ki = Integer.class;
Number n = Integer.valueOf(42);
Class<? extends Number> kn = n.getClass();
assert ki == kn;
```

Для данного загрузчика класса один и тот же тип всегда представляется одним и тем же объектом класса. Чтобы лучше подчеркнуть этот момент, мы здесь сравниваем объекты класса на тождественность (с помощью оператора `==`). Фактически это проверка на равенство `Class`: класс `Class` не переопределяет метод `equals`, унаследованный от `Object`, который реализован как сравнение на тождественность. Отказ от переопределения `Object::equals` имеет смысл, когда будет существовать всего один экземпляр класса (и еще в некоторых обстоятельствах; см. Bloch 2017, совет 10).

В этом примере показано, как компилятор обращается с литералами классов и методом `getClass`: если `T` – тип без параметров, то `T.class` имеет тип `Class<T>`, а если `e` – выражение типа `T`, то `e.getClass()` имеет тип `Class<? extends T>`. (Мы увидим, что происходит, когда `T` имеет параметры-типы, в следующем разделе.) Джокер необходим, потому что тип объекта, на который ссылается переменная, может быть подтипом типа переменной – как в этом случае, где переменная типа `Number` содержит объект типа `Integer`.

Часто при использовании рефлексии статический тип объекта класса неизвестен. В таких случаях мы должны писать `Class<?>` для типа, используя неограниченный джокер. Однако если информация, предоставляемая параметром-типом, доступна, то она может быть весьма ценной, как в методе `toArray` класса `MicroArrayList_v2` (см. раздел «Классная альтернатива» главы 5).

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(Class<T> clazz) {
  T[] a = (T[])Array.newInstance(clazz, size); // непроверенное приведение
  System.arraycopy(data, 0, a, 0, size);
  return a;
}
```

Здесь параметр-тип гарантирует, что непроверенное приведение безопасно. Класс `Class<T>` содержит методы, которые используют параметр-тип интересным способом:

```
class Class<T> {
  public T newInstance(); ❶
  public boolean isInstance(Object obj); ❷
  public T cast(Object o); ❸
  public Class<? super T> getSuperclass(); ❹
  public boolean isAssignableFrom(Class<?> cls); ❺
  public <U> Class<? extends U> asSubclass(Class<U> k); ❻
  public <A extends Annotation> A getAnnotation(Class<A> annotationClass);
❼
  public boolean isAnnotationPresent(Class<? extends Annotation>
                                     annotationClass); ❽
  ...
}
```

Метод ❶ возвращает новый экземпляр класса, который, конечно же, будет иметь тип `T`. Метод ❸ приводит произвольный объект к классу получателя и, следовательно, либо возбуждает исключение приведения класса, либо возвращает результат типа `T`. Чтобы избежать исключения, вы можете использовать метод ❷, который проверяет, является ли объект экземпляром этого класса. Метод ❹ возвращает суперкласс, который должен иметь заданный тип. Метод ❻ проверяет, что класс получателя является подклассом класса аргумента и либо возбуждает исключение приведения класса, либо возвращает получателя, тип которого подходящим образом изменен. Опять-таки исключения можно избежать, если предварительно вызвать метод ❺, проверяющий, что данный класс действительно является подклассом аргумента.

Методы ❼ и ❽ – часть механизма аннотаций. Они интересны тем, что показывают, как параметр-тип для классов можно употребить с пользой. Например, `Retention` является подклассом `Annotation`, поэтому мы можем извлечь мета-аннотацию `Retention` из класса аннотации `ac` следующим образом:

```
Retention r = ac.getAnnotation(Retention.class);
```

Здесь параметризованный тип имеет два преимущества. Во-первых, результат вызова не нужно ни к чему приводить, потому что система параметризованных типов может назначить ему совершенно точный тип. Во-вторых, если вы случайно вызовете метод, который передает объект класса для класса, не являющегося подклассом `Annotation`, то это будет обнаружено на этапе компиляции, а не выполнения.

Вот еще интересное использование объектов классов: служебный класс `Collections` содержит метод, который строит обертку, проверяющую, действительно ли каждый элемент, добавляемый в данный список или извлекаемый из него, принадлежит данному классу. (Подобные методы существуют и для других классов коллекций, например, множеств и отображений.) Метод имеет следующую сигнатуру:

```
public static <T> List<T> checkedList(List<T> l, Class<T> k)
```

Эта обертка дополняет статическую проверку на этапе компиляции динамической проверкой на этапе выполнения, что может быть полезно для повышения безопасности или взаимодействия с унаследованным кодом (см. раздел

«Проверяемые коллекции» главы 16). Реализация вызывает метод `Class::cast`, описанный ранее, где получателем является маркер класса, переданный методу, а приведение применяется ко всем элементам, которые читаются из списка или записываются в него с помощью методов `get`, `set` или `add`. Снова отметим, что параметр-тип `Class<T>` означает, что код `checkedList` не требует дополнительных приведений сверх обращений к `Class::cast` и что компилятор может проверить, что метод вызывается с подходящим объектом класса.

Рефлектированные типы являются сберегаемыми

Рефлексия делает доступной программе только сбереженную информацию о типе. Поэтому необходимо, чтобы каждый маркер класса соответствовал сберегаемому типу. Если вы попытаетесь применить рефлексию к параметризованному типу, то получите сбереженную информацию для соответствующего простого типа:

org/jgcbook/chapter06/B_reflected_types_are_reifiable_types/Program_1

```
List<Integer> ints = new ArrayList<Integer>();
List<String> strs = new ArrayList<String>();
assert ints.getClass() == strs.getClass();
assert ints.getClass() == ArrayList.class;
```

Здесь тип списка целых и тип списка строк представлены одним и тем же объектом класса, для которого литералом класса является `ArrayList.class`.

Поскольку класс всегда представляет сберегаемый тип, нет смысла передавать несберегаемый параметр-тип классу `Class`. Поэтому оба основных способа порождения класса с параметром-типом, а именно метод `getClass` и литералы классов, спроектированы так, что отдают сберегаемый тип для параметра-типа во всех случаях.

Напомним, что метод `getClass` обрабатывается компилятором специальным образом. Если выражение `e` имеет тип `T`, то выражение `e.getClass()` имеет тип `Class<? extends |T|>`, где `|T|` – результат стирания типа `T`. Приведем пример:

org/jgcbook/chapter06/B_reflected_types_are_reifiable_types/Program_2

```
List<Integer> ints = new ArrayList<Integer>();
Class<? extends List> k = ints.getClass();
assert k == ArrayList.class;
```

Здесь выражение `ints` имеет тип `List<Integer>`, поэтому выражение `int.getClass()` имеет тип `Class<? extends List>`; это так, потому что стирание `List<Integer>` дает простой тип `List`. Фактическим значением `k` является `ArrayList.class`, который имеет тип `Class<ArrayList>`, действительно являющийся подтипом `Class<? extends List>`.

Литералы классов также имеют ограничения; даже синтаксически недопустимо передавать параметр-тип типу в литерале класса. Поэтому следующий фрагмент некорректен:

org/jgcbook/chapter06/B_reflected_types_are_reifiable_types/ClassLiteral

```
class ClassLiteral {
  public Class<?> k = List<Integer>.class; // синтаксическая ошибка
}
```

На самом деле грамматика Java такова, что подобную фразу трудно разобрать, что приводит к целой серии сообщений о синтаксических ошибках:

```
% javac ClassLiteral.java
ClassLiteral.java:6: error: <identifier> expected
  public Class<?> k = List<Integer>.class; // синтаксическая ошибка
                                    ^
ClassLiteral.java:6: error: <identifier> expected
  public Class<?> k = List<Integer>.class; // синтаксическая ошибка
                                          ^
2 errors
```

Эта синтаксическая ошибка влечет за собой нерегулярность. Во всех остальных местах, где требуется сберегаемый тип, мы можем передать как простой тип (например, `List`) так и параметризованный тип с неограниченными джокерами (например, `List<?>`). Однако для литералов классов необходимо передавать только простой тип. Никакие джокеры, даже неограниченные, недопустимы; замена `List<Integer>` на `List<?>` в показанном выше коде приведет к аналогичному каскаду ошибок.

Заметим, что литералы классов представляют объекты классов, созданные JVM во время выполнения. Поэтому в типе вида `Class<T>` параметр `T` может представлять только тип, известный во время выполнения, т. е. сберегаемый тип. Это согласуется с синтаксическим ограничением, разрешающим использовать только простые типы для создания литералов классов. Хотя переменную или параметр метода можно объявить с ограниченным типом-параметром, например `Class<? Extends T>`, объект, на который она ссылается, всегда будет иметь какой-то конкретный сберегаемый тип, в данном случае подтип `T`. То же самое верно для типов массивов: как мы видели в главе 5, можно объявить переменную с типом массива несберегаемого типа, например `List<? Extends Number>[]`, но при создании массива, на который она ссылается, элементы обязательно должны иметь сберегаемый тип.

Рефлексия для примитивных типов

Любой тип в Java, в том числе примитивный и тип массива, имеет литерал класса и соответствующий объект класса.

Например, `int.class` обозначает объект класса для примитивного типа целых (этот маркер также является значением статического поля `Integer.TYPE`). Типом этого маркера не может быть `Class<int>`, потому что `int` не является ссылочным типом, поэтому принимается тип `Class<Integer>`. Выбор, пожалуй, странный, так как при таком типе можно было бы ожидать, что вызовы `int.class.cast(o)` и `int.class.newInstance()` возвращают значение типа `Integer`, тогда как на самом деле они возбуждают исключения. Аналогично можно было бы ожидать, что вызов

```
java.lang.reflect.Array.newInstance(int.class, size)
```

возвращает значение типа `Integer[]`, тогда как на самом деле он возвращает значение типа `Object` (как мы видели в разделе «Как создавать массивы» главы 5), которое только потом можно привести к типу `int[]`. Эти примеры наводят на мысль, что, возможно, имело бы больше смысла приписать литералу `int.class` тип `Class<?>`.

С другой стороны, `int[].class` обозначает объект класса для массивов с элементами примитивного типа `int`, и его типом является `Class<int[]>`, что разрешено, поскольку `int[]` – ссылочный тип.

Обобщенная библиотека рефлексии

Как мы видели, беспечное использование непроверенных приведений может приводить к проблемам, в том числе нарушениям принципов правдивости рекламы и непристойного обнажения. Один из способов минимизировать применение непроверенных приведений – инкапсулировать их в библиотеке. Библиотеку можно тщательно проанализировать, дабы убедиться, что непроверенные приведения в ней используются безопасно, тогда как код, обращающийся к библиотеке, может быть свободен от таких приведений.

В примере 6.1 предлагается библиотека обобщенных функций, в которых рефлексия используется типобезопасным образом. В ней определен служебный класс `GenericReflection`, содержащий следующие методы:

```
public static <T> T newInstance(T object)
public static <T> Class<? extends T> getComponentType(T[] a)
public static <T> T[] newArray(Class<? extends T> k, int size)
public static <T> T[] newArray(T[] a, int size)
```

Первый метод принимает объект, находит класс этого объекта и возвращает новый экземпляр этого класса; он должен иметь такой же тип, как оригинальный объект. Второй метод принимает массив и возвращает маркер класса для типа его элементов, который хранится в составе информации о типе массива времени выполнения. Третий метод выделяет память для нового массива заданного размера, тип элементов которого определен заданным маркером класса. Четвертый метод принимает массив и размер и выделяет память для нового массива заданного размера, элементы которого имеют такой же тип, как у заданного массива; это просто композиция двух предыдущих методов. Код каждого из первых трех методов состоит из вызова одного или двух методов из библиотеки рефлексии Java и непроверенного приведения к подходящему типу возвращаемого значения.

Непроверенные приведения необходимы, потому что методы из библиотеки рефлексии Java по различным причинам не могут вернуть достаточно точные типы. Метод `getComponentType` находится в классе `Class<T>`, и Java не предлагает никакой возможности ограничить тип получателя классом `Class<T[]>` в сигнатуре метода (хотя метод возбуждает исключение, если получатель не является маркером класса для типа массива). Метод `newInstance` в классе `java.lang.reflect.Array` должен возвращать значение типа `Object`, а не `T[]`, потому что может вернуть массив примитивного типа. Метод `getClass` при вызове от имени получателя типа `T` возвращает маркер не типа `Class<? Extends T>`, а типа `Class<?>` из-за стирания, которое необходимо, чтобы маркеры класса всегда имели сберегаемый тип. Однако во всех этих случаях непроверенное приведение безопасно, и пользователи могут вызывать все четыре определенных здесь библиотечных метода, не нарушая гарантии успешного приведения.

Пример 6.1. *Типобезопасная библиотека обобщенной рефлексии*

org/jgcbook/chapter06/D_a_generic_reflection_library/GenericReflection

```
class GenericReflection {
  public static <T> T newInstance(T obj) throws ReflectiveOperationException
{
    Object newobj = obj.getClass().getConstructor().newInstance();
    return (T)newobj; // непроверенное приведение
  }
  public static <T> Class<? extends T> getComponentType(T[] a) {
    Class<?> k = a.getClass().getComponentType();
    return (Class<? extends T>)k; // непроверенное приведение
  }
  public static <T> T[] newArray(Class<? extends T> k, int size) {
    if (k.isPrimitive()) {
      throw new IllegalArgumentException ("Argument cannot be primitive:
"+k);
    }
    Object a = java.lang.reflect.Array.newInstance(k, size);
    return (T[])a; // непроверенное приведение
  }
  public static <T> T[] newArray(T[] a, int size) {
    return newArray(getComponentType(a), size);
  }
}
```

В первом методе используется метод `Constructor::newInstance` (из `java.lang.reflect`), чтобы избежать обращения к методу `Class.newInstance`, который уже давно объявлен нерекомендуемым из-за того, что распространяемые им исключения, возбужденные конструктором без аргументов, обходят проверку исключений на этапе компиляции. Метод `Constructor::newInstance` не страдает от этой проблемы, потому что обертывает исключения, возбужденные конструктором, (проверяемым) классом `InvocationTargetException`. Однако вызовы `getConstructor` и `newInstance` могут возбуждать другие исключения, их общим суперклассом является `ReflectiveOperationException`.

Второй метод гарантированно будет правильно типизирован в любой программе, соблюдающей принципы непристойного обнажения и правдивости рекламы. Первый принцип гарантирует, что тип элемента на этапе компиляции будет сберегаемым, а второй – что сбереженный тип элемента, возвращенный на этапе выполнения, является подтипом сберегаемого типа элемента, объявленного на этапе компиляции.

Третий метод возбуждает исключение `IllegalArgumentException`, если его аргумент имеет примитивный тип. Тем самым мы обрабатываем следующий коварный случай: если первым аргументом является, скажем, `int.class`, то его тип равен `Class<Integer>`, но новый массив будет иметь тип `int[]`, не являющийся подтипом `Integer[]`. Эта проблема не возникла бы, если бы `int.class` имел тип `Class<?>`, а не `Class<Integer>`, как обсуждалось в предыдущем разделе.

В качестве примера использования первого метода ниже приведен метод, который копирует коллекцию в новую коллекцию того же вида с сохранением типа аргумента:

org/jgcbook/chapter06/D_a_generic_reflection_library/Program_1

```
public static <T, C extends Collection<T>> C copy(C coll) {
  try {
    C copy = GenericReflection.newInstance(coll);
    copy.addAll(coll);
    return copy;
  } catch (ReflectiveOperationException e) {
    throw new RuntimeException(e);
  }
}
```

Вызов `copy` для `ArrayList<Integer>` возвращает новый экземпляр `ArrayList<Integer>`:

org/jgcbook/chapter06/D_a_generic_reflection_library/Program_1

```
List<Integer> coll = new ArrayList<>(List.of(1, 2, 3));
assert copy(coll).equals(coll);
assert copy(coll).getClass().equals(coll.getClass());
```

Аналогично вызов `copy` для `HashSet<String>` возвращает новый экземпляр `HashSet<String>` и т. д.

В качестве примера использования последнего метода ниже приведен метод `toArray` из раздела «Принцип правдивости рекламы» главы 5, в котором непроверенные приведения заменены обращением к обобщенной библиотеке рефлексии:

org/jgcbook/chapter06/D_a_generic_reflection_library/Program_2

```
public static <T> T[] toArray(Collection<T> c, T[] a) {
  if (a.length < c.size()) a = GenericReflection.newArray(a, c.size());
  int i=0; for (T x : c) a[i++] = x;
  if (i < a.length) a[i] = null;
  return a;
}
```

В общем случае, если вам позарез необходимо использовать непроверенные приведения, то мы рекомендуем инкапсулировать их небольшим числом библиотечных методов, как мы и проделали. Не позволяйте непроверенному коду расползаться по всей программе!

Рефлексия для параметризованных типов

Параметризованные типы изменяют библиотеку рефлексии в двух отношениях. Мы уже обсудили привлечение параметризованных типов к рефлексии, а точнее добавление параметра-типа в класс `Class<T>`. В этом разделе мы обсудим привлечение рефлексии к параметризованным типам, а точнее добавление методов и классов, поддерживающих доступ к параметризованным типам.

В примере 6.2 показана простая демонстрация использования рефлексии для параметризованных типов. В ней рефлексия применяется для нахождения класса с заданным именем, а затем печатается класс вместе с его полями, конструкторами и методами, для чего используются классы `Field`, `Constructor`

и `Method`. Два метода для преобразования класса, поля, конструктора или метода в строку для печати – `toString` и `toGenericString`. Менее информативная реализация `toString`, написанная еще до появления параметризованных типов, поддерживается в основном ради обратной совместимости. Небольшой демонстрационный класс показан в примере 6.3, а пример выполнения этого класса – в примере 6.4.

Пример 6.2. *Использование рефлексии для параметризованных типов*

org/jgcbook/chapter06/E_reflection_for_generics/ReflectionForGenerics

```
class ReflectionForGenerics {
  public static void toString(Class<?> k) {
    System.out.println(k + " (toString)");
    System.out.println(k);
    for (Field f : k.getDeclaredFields())
      System.out.println(f.toString());
    for (Constructor<?> c : k.getDeclaredConstructors())
      System.out.println(c.toString());
    for (Method m : k.getDeclaredMethods())
      System.out.println(m.toString());
    System.out.println();
  }
  public static void toGenericString(Class<?> k) {
    System.out.println(k + " (toGenericString)");
    System.out.println(k.toGenericString());
    for (Field f : k.getDeclaredFields())
      System.out.println(f.toGenericString());
    for (Constructor<?> c : k.getDeclaredConstructors())
      System.out.println(c.toGenericString());
    for (Method m : k.getDeclaredMethods())
      System.out.println(m.toGenericString());
    System.out.println();
  }
  public static void main (String[] args) throws ClassNotFoundException {
    for (String name : args) {
      Class<?> k = Class.forName(name);
      toString(k);
      toGenericString(k);
    }
  }
}
```

Пример *6.3. Демонстрационный класс*

org/jgcbook/chapter06/E_reflection_for_generics/Cell

```
class Cell<E> {
  private E value;
  public Cell(E value) { this.value=value; }
  public E getValue() { return value; }
  public void setValue(E value) { this.value=value; }
  public static <T> Cell<T> copy(Cell<T> cell) {
    return new Cell<T>(cell.getValue());
  }
}
```

Пример 6.4. *Пример выполнения*

```
% java ReflectionForGenerics Cell
class Cell (toString)
class Cell
private java.lang.Object Cell.value
public Cell(java.lang.Object)
public java.lang.Object Cell.getValue()
public void Cell.setValue(java.lang.Object)
public static Cell Cell.copy(Cell)

class Cell (toGenericString)
class Cell<E>
private E Cell.value
public Cell(E)
public E Cell.getValue()
public void Cell.setValue(E)
public static <T> Cell<T> Cell.copy(Cell<T>)
```

Пример выполнения показывает, что хотя информация о сбереженном типе для объектов и маркеров класса не содержит никаких сведений о параметризованных типах, в фактическом байт-коде класса все же закодирована информация о параметризованных, равно как и о стертых типах. Информация о параметризованных типах, по существу, является комментарием. При выполнении кода она игнорируется, а сохраняется только для целей рефлексии.

Рефлексия параметризованных типов

Библиотека рефлексии предоставляет интерфейс `Type` для описания параметризованного типа. Существует единственный класс, который реализует этот интерфейс и четыре расширяющих его интерфейса, что соответствует пяти разновидностям типов:

- класс `Class`, представляющий примитивный или простой тип;
- интерфейс `ParameterizedType`, представляющий параметризованный класс или интерфейс, от которого можно получить массив типов-параметров;
- интерфейс `TypeVariable`, представляющий переменную-тип, от которого можно получить границы этого типа;
- интерфейс `GenericArrayType`, представляющий массив, от которого можно получить тип элементов массива;
- интерфейс `WildcardType`, представляющий джокер, от которого можно получить нижнюю и верхнюю границу джокера.

Выполнив серию тестов экземпляра для каждого из этих интерфейсов, вы сможете определить, с типом какого вида работаете. Затем этот тип можно напечатать или обработать, пример мы вскоре увидим.

Имеются методы для возврата суперкласса или суперинтерфейсов класса в виде типов и для доступа к параметризованному типу поля, типам аргументов конструктора и типам аргументов и результата метода.

Можно также получить переменные-типы, обозначающие формальные параметры в объявлении класса или интерфейса, параметризованного метода или конструктора. Тип переменной-типа принимает параметр и запи-

сывается в виде `TypeVariable<D>`, где `D` представляет тип объекта, объявившего переменную-тип. Таким образом, переменные-типы класса имеют тип `TypeVariable<Class<?>>`, тогда как переменные-типы обобщенного метода имеют тип `TypeVariable<Method>`. Можно возразить, что этот параметр-тип – спорное проектное решение: он не особенно полезен, а его наличие – причина проблемы, описанной в конце раздела «Принцип непристойного обнажения» главы 5.

В примере 6.5 эти методы используются для печати всей информации из заголовка, ассоциированного с классом. Вот два примера их использования:

```
% java ReflectionDemo java.util.AbstractList
class java.util.AbstractList<E>
extends java.util.AbstractCollection<E>
implements java.util.List<E>

% java ReflectionDemo java.lang.Enum
class java.lang.Enum<E extends java.lang.Enum<E>>
implements java.lang.constant.Constable,java.lang.Comparable<E>,java.
io.Serializable
```

Код в примере 6.5 содержит методы для печати объявления класса, его суперкласса и реализуемых им интерфейсов. Основной частью кода является метод `printType`, сопоставление с образцом в предложении `switch` используется для классификации типа в соответствии с пятью ранее перечисленными случаями.

Пример 6.5. *Манипулирование типом* `Type`

org/jgcbook/chapter06/F_reflecting_generic_types/ReflectionDemo

```
class ReflectionDemo {
  public static String printSuperclass(Type sup) {
    if (!sup.equals(Object.class)) {
      return "extends " + printType(sup) + "\n";
    }
    return "";
  }
  public static String printInterfaces(Type[] impls) {
    if (impls.length > 0) {
      return "implements " + Arrays.stream(impls)
          .map(ReflectionDemo::printType)
          .collect(Collectors.joining(",")) + "\n";
    }
    return "";
  }
  public static String printTypeParameters(TypeVariable<?>[] vars) {
    if (vars.length > 0) {
      return Arrays.stream(vars)
          .map(b -> b.getName() + printBounds(b.getBounds()))
          .collect(Collectors.joining(",","<",">\n"));
    }
    return "";
  }
  public static String printBounds(Type[] bounds) {
    if (bounds.length > 0 && ! Arrays.equals(bounds,new Type[]{ Object.class
})) {
```

```
        return " extends " +
        Arrays.stream(bounds)
            .map(ReflectionDemo::printType)
            .collect(Collectors.joining(" & "));
      }
      return "";
    }
    public static String printType(Type type) {
      return switch(type) {
        case Class<?> cls -> cls.getName();
        case ParameterizedType p -> printParameterizedType(p);
        case TypeVariable<?> tv -> tv.getName();
        case GenericArrayType gat -> gat.getGenericComponentType() + "[]";
        case WildcardType wt -> printWildcard(wt);
        default -> throw new IllegalStateException("Unexpected value: " +
type);
      };
    }
    private static String printParameterizedType(ParameterizedType p) {
      Class<?> c = (Class<?>)p.getRawType();
      Type o = p.getOwnerType();
      String ptString = o != null ? printType(o) + "." : "";
      ptString = ptString + c.getName();
      return ptString + Arrays.stream(p.getActualTypeArguments())
          .map(ReflectionDemo::printType)
          .collect(Collectors.joining(",","<",">"));
    }
    private static String printWildcard(WildcardType wt) {
      Type[] upper = wt.getUpperBounds();
      Type[] lower = wt.getLowerBounds();
      if (lower.length == 1) {
        return "? super " + printType(lower[0]);
      } else if (upper.length == 1 && ! upper[0].equals(Object.class)) {
        return "? extends" + printType(upper[0]);
      }
      return "";
    }
    public static String printClass(Class<?> c) {
      return "class " +
        printType(c) +
        printTypeParameters(c.getTypeParameters()) +
        printSuperclass(c.getGenericSuperclass()) +
        printInterfaces(c.getGenericInterfaces());
      }
    public static void main(String[] args) throws ClassNotFoundException {
      for (String name : args) {
        Class<?> c = Class.forName(name);
        System.out.println(printClass(c));
      }
    }
}
```

Заключение

В этой главе мы изучили связь между рефлексией и параметризованными типами в форме класса `Class<T>`, который объединяет информацию о типе на этапе компиляции и сбереженную, а также методы рефлексии, открывающие доступ к информации о параметризованном типе, сохраненной на этапе выполнения.

В следующей главе, последней в части I, мы приведем ряд мыслей на тему того, как использовать параметризованные типы на практике и как обойти проблемы, которые иногда при этом возникают.

Глава 7

Эффективные дженерики Java

В этой главе содержатся рекомендации по эффективному использованию дженериков на практике. Советы разнятся по применимости от разумной практики, которой нужно по возможности следовать всегда, до приемов, которые следует рассматривать, только когда ничего другого не остается. Каждому совету предшествует указание его места на этой шкале.

Название этой главы – дань уважения книге Джошуа Блоха «Effective Java» (2017).

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter07*.

Искореняйте предупреждения о невозможности проверки

Предупреждения о невозможности проверки редко бывают неизбежными. В этом совете описываются необычные ситуации, когда избежать такого предупреждения невозможно, и показано, что делать в таких случаях.

В разделе «Непроверенные приведения» главы 5 мы упоминали некоторые ситуации, когда компилятор не может проверить, что предупреждение о невозможности проверки не является необходимым. Ниже приведен еще один пример. Метод `promote` в этом коде приводит список объектов к типу списка строк, если он содержит только строки, и возбуждает исключение приведения класса в противном случае. Метод `main` – примитивный тест `promote`, в котором предложения `assert` используются, чтобы показать ожидаемый путь выполнения программы.

Если выполнить эту программу с включенными утверждениями, то утверждение не сработает.

org/jgcbook/chapter07/A_eliminate_unchecked_warnings/Promote_1

```
class Promote_1 {
  public static List<String> promote(List<Object> objs) {
    for (Object o : objs)
      if (!(o instanceof String))
        throw new ClassCastException();
    return (List<String>)(List<?>)objs; // непроверенное приведение
  }
```

```
  public static void main(String[] args) {
    List<Object> objs1 = List.of("one","two");
    List<Object> objs2 = List.of(1,"two");
    List<String> strs1 = promote(objs1);
    assert strs1 == (List<?>)objs1;
    try {
      List<String> strs2 = promote(objs2);
      assert false;
    } catch (ClassCastException e) {
      // всегда должен приходить в это место
    }
  }
}
```

Метод `promote` в цикле обходит список объектов и возбуждает исключение приведения класса, если объект не является строкой. Поэтому при достижении последней строчки метода безопасно привести список объектов к списку строк.

Но компилятор этого понять не может, поэтому программист вынужден использовать непроверенное приведение. Приводить список объектов к списку строк запрещено, поэтому приведение нужно выполнять в два шага. Сначала мы приводим список объектов к списку джокерного типа; это приведение безопасно. Затем мы приводим список джокерного типа к списку строк; это приведение разрешено, но генерирует предупреждение о невозможности проверки:

```
% javac -Xlint:unchecked Promote.java
Promote.java:7: warning: [unchecked] unchecked cast
    return (List<String>)(List<?>)objs; // непроверенное приведение
                          ^
  required: List<String>
  found:    List<CAP#1>
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 warning
```

Поскольку мы знаем, что это приведение действительно типобезопасно, предупреждение можно подавить. На самом мы рекомендуем так и делать, потому что иначе многочисленные ложные тревоги в диагностических сообщениях компилятора могут побудить вас проигнорировать все скопом – и при этом, возможно, пропустить заслуживающие внимания. Именно в этом заключается цель аннотации `@SuppressWarnings`. Из обоснования подавления предупреждений следует, что вам надлежит:

- максимально ограничить область видимости аннотации, чтобы не подавить полезные предупреждения в этой области;
- представить обоснование подавления предупреждения, чтобы с ним мог ознакомиться человек, который в дальнейшем будет сопровождать код.

Реализовав эти рекомендации (вытекающие из совета 27 в книге Bloch 2017) в приведенном выше примере, получим:

org/jgcbook/chapter07/A_eliminate_unchecked_warnings/Promote_2

```
class Promote_2 {
  public static List<String> promote(List<Object> objs) {
    for (Object o : objs)
      if (!(o instanceof String))
        throw new ClassCastException();
    @SuppressWarnings("unchecked") // это приведение типобезопасно, потому что
                                   // попасть сюда можно, только если все
элементы
                                   // списка являются строками
    List<String> strings = (List<String>) (List<?>) objs;
    return strings;
  }
}
```

Заметим, что в новом локальном объявлении мы не аннотируем весь метод целиком, а применяем аннотацию `@SuppressWarnings` только к следующей за ней строчке.

Как и любую практическую рекомендацию, эту нужно применять прагматически. Если в одном методе много предупреждений о невозможности проверки, то буквальное следование ей может замусорить код, так что его будет трудно читать. В таких ситуациях может быть оправдано расширение области видимости аннотации на метод или даже весь класс. Именно по этой причине мы иногда отклоняемся от нее в этой книге, особенно если появление предупреждений о невозможности проверки подробно объяснено в тексте.

Наконец, повторим, что прежде чем подавлять предупреждение о невозможности проверки, вы должны быть абсолютно уверены, что понимаете причину предупреждения, и убеждены в его безвредности. В противном случае, подавив предупреждение в одном месте, вы получите исключение в другом!

Обеспечивайте типобезопасность при вызове недоверенного кода

Во многих системах ваш код должен взаимодействовать с чужим кодом, которому нельзя доверять.

Важно понимать, что гарантии, предлагаемые параметризованными типами, действуют, только если нет предупреждений о невозможности проверки. Это означает, что дженерики гораздо менее полезны для обеспечения безопасности в коде, написанном кем-то другим, поскольку вы не можете знать, возникали ли предупреждения о невозможности проверки во время его компиляции.

Допустим, имеется класс, который определяет заказ, и его подкласс, определяющий аутентифицированный заказ:

org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/Order

```
class Order { ... }
class AuthenticatedOrder extends Order { ... }
```

Интерфейсы определяют заказчиков и исполнителей заказов. В данном случае заказчик должен размещать только аутентифицированные заказы, а исполнитель обрабатывает все виды заказов.

org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/Order

```
interface OrderSupplier {
  public void addOrders(List<AuthenticatedOrder> orders);
  public List<AuthenticatedOrder> getOrders();
}
interface OrderProcessor {
  public void processOrders(List<? extends Order> orders);
}
```

Глядя на эти типы, вы можете подумать, что показанный ниже посредник гарантирует, что от заказчика исполнителю могут поступать только аутентифицированные заказы.

org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/NaiveBroker

```
class NaiveBroker {
  public void connect(OrderSupplier supplier, OrderProcessor processor) {
    List<AuthenticatedOrder> orders = new ArrayList<>();
    supplier.addOrders(orders);
    processor.processOrders(orders);
  }
}
```

Но злонамеренный заказчик все-таки может делать неаутентифицированные заказы, причем двумя способами. Вот первый из них:

org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/DeviousSupplier_1

```
class DeviousSupplier_1 implements OrderSupplier {
  public void addOrders(List<AuthenticatedOrder> orders) {
    List raw = orders;
    Order order = new Order(); // не аутентифицирован
    raw.add(order);            // непроверенный вызов
  }
  public List<AuthenticatedOrder> getOrders() {
    ...
  }
}
```

При компиляции злонамеренного заказчика выдается предупреждение о невозможности проверки, но посредник об этом знать не может.

Некомпетентность может стать причиной такого же количества проблем, как и злонамеренность. Любой код, порождающий предупреждения о невозможности проверки при компиляции, мог бы привести к таким проблемам, быть может, просто потому что автор допустил ошибку. А код, написанный до появления дженериков, может создать ту же проблему не из-за злонамеренности или некомпетентности, а просто потому что работает с простыми типами.

Правильное решение – возложить на посредника обязанность передавать заказчику проверенный список.

org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/WaryBroker_1

```
class WaryBroker_1 {
  public void connect(OrderSupplier supplier, OrderProcessor processor) {
    List<AuthenticatedOrder> orders = new ArrayList<AuthenticatedOrder>();
    supplier.addOrders(Collections.checkedList(orders, AuthenticatedOrder.
```

```
class));
    processor.processOrders(orders);
  }
}
```

Теперь, если заказчик попытается добавить в список что-то, кроме аутентифицированного заказа, то будет возбуждено исключение

Использовать проверенные коллекции таким образом невозможно, если заказчик возвращает коллекцию, которую сам же и создал. И это второй способ, каким `DeviousSupplier` может разместить неаутентифицированные заказы.

org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/DeviousSupplier_2

```
class DeviousSupplier_2 implements OrderSupplier {
  public void addOrders(List<AuthenticatedOrder> orders) {
  ...
  }
  public List<AuthenticatedOrder> getOrders() {
    List<Order> orders = new ArrayList<>();
    orders.add(new Order());
    return (List<AuthenticatedOrder>)(List)orders; // непроверенное приведе-
ние
  }
}
```

В этом случае клиентский код должен проверить, что возвращенная коллекция не содержит объектов недопустимого типа. Чтобы избежать возбуждения исключения приведения класса где-то совсем в другом месте, что сильно затруднило бы диагностику ошибки, недоверчивый клиент проверит возвращенную коллекцию немедленно, чтобы обнаружить нарушение типов как можно ближе к месту его возникновения. В этом примере выделенный код возбудит исключение, если список содержит объект не того типа.

org/jgcbook/chapter07/B_enforce_type_safety_when_calling_untrusted_code/WaryBroker_2

```
class WaryBroker_2 {
  public void connect(OrderSupplier supplier, OrderProcessor processor) {
    List<AuthenticatedOrder> orders = supplier.getOrders();
    for (AuthenticatedOrder order : orders) {}
  }
}
```

Третий способ избежать нарушений системы типов – специализация, т. е. создание специального типа списка, который может содержать только аутентифицированные заказы. Мы обсудим это ниже.

Специализируйте для создания сберегаемых типов

Описываемая в этом совете техника полезна и разумна, но редко бывает востребована.

Параметризованные типы не являются сберегаемыми, поэтому к ним неприменимы такие операции, как тесты экземпляров, приведение и создание массива, применимые только к сберегаемым типам. В подобных случаях помогает обходной маневр – создать специализированную версию параметризованного типа. Мы будем использовать наследование абстрактной коллекции для создания конкретной (см. более подробное объяснение в разделе «Настра-

ивайте коллекции с помощью абстрактных классов» главы 17). В примере 7.1 показано, как создать специализированный список строк; специализация для других типов аналогична.

Пример 7.1. *Специализация для создания сберегаемого типа*

org/jgcbook/chapter07/C_specialize_to_create_reifiable_types/ListString

```
class ListString extends AbstractList<String> implements List<String> {
  private final List<String> list;
  public ListString() { this.list = new ArrayList<>(); }
  public ListString(Collection<? extends String> c) {
    this.list = new ArrayList<>(c);
  }
  public ListString(int capacity) { this.list = new ArrayList<>(capacity); }
  public int size() { return list.size(); }
  public String get(int i) { return list.get(i); }
  public String set(int i, String s) { return list.set(i,s); }
  public String remove(int i) { return list.remove(i); }
  public void add(int i, String s) { list.add(i,s); }
}
```

Здесь объявляется `ListString` (не параметризованный, а значит, сберегаемый тип), являющийся подтипом `List<String>` (параметризованного, а значит, не сберегаемого). Таким образом, каждое значение первого типа принадлежит второму, но обратное неверно. Вот пример использования:

org/jgcbook/chapter07/C_specialize_to_create_reifiable_types/Program_1

```
ListString listString1 = new ListString(List.of("one", "two"));
ListString listString2 = new ListString(List.of("seven", "eight"));
List<? extends List<?>> lists =
  Arrays.asList(listString1, List.of(3, 4), List.of("five", "six"), list-
String2);
ListString[] listStringArray = lists.stream()
    .filter(list -> list instanceof ListString)
    .toArray(ListString[]::new);
assert Arrays.toString(listStringArray).equals("[[one, two], [seven,
eight]]");
```

Здесь мы создаем список списков, а затем ищем среди них те, что реализуют `ListString`, и помещаем их в массив. Теперь создание массива, тесты экземпляров и приведения не составляют никакой проблемы, потому что применяются к сберегаемому типу `ListString`, а не к несберегаемому `List<String>`. Заметим, что `List<String>`, который не был обернут, не распознается как экземпляр `ListString`, поэтому третий элемент списка списков не копируется в массив.

Реализация класса `ListString` – прямолинейное применение техники, описанной в разделе «Настраивайте коллекции с помощью абстрактных классов» главы 17. Экземпляр `ListString` – представление стоящего за ним `ArrayList`, которое приводит к ошибке компиляции при любой попытке вставить в список что-то, кроме строки.

Этот код сделан простым, чтобы не удлинять объяснение. Но возможны и другие варианты: в более эффективной версии можно было бы отказаться от использования `AbstractList`, а вместо этого реализовать непосредственно

все 25 методов интерфейса `List` вместе с методом `toString`. Можно было бы выбрать реализацию, которая вместо создания внутреннего списка оборачивает переданный список, так чтобы типобезопасность гарантировалась на этапе выполнения посредством вставленных компиляторов операций приведения. Можно было бы также добавить дополнительные методы в интерфейс `ListString`, например метод `unwrap`, который возвращает базовый `List<String>`, или вариант `subList` который возвращает `ListString` вместо `List<String>` путем рекурсивного применения `wrap` к делегированному вызову.

Избегайте бессмысленных переменных-типов

Это универсальный совет.

Параметры-типы обобщенных методов обычно нужны, чтобы выразить некоторые ограничения на тип или типы аргументов метода и (или) возвращаемое значение. Например, метод суммирования элементов коллекции мог бы наложить на свой единственный параметр ограничение – быть коллекцией экземпляров `Number`:

```
public static <T extends Number> T sum(Collection<T> c);
```

Но может быть и так, что любой тип приемлем, а параметр-тип служит для выражения связи между различными параметрами метода или между параметром и возвращаемым значением. Например, объявление `java.util.Collections::synchronizedList` позволяет принять любой тип списка, но требует, чтобы возвращаемое значение было того же типа:

```
public static <T> List<T> synchronizedList(List<T> list);
```

Однако иногда у обобщенного метода нет таких ограничений. Например, метод `shuffle` класса `Collections`, который случайным образом переставляет элементы списка, может принимать список любого типа. Возникает искушение объявить его так же, как в предыдущих примерах:

```
public static <T> void shuffle(List<T> list);
```

и такое объявление вполне допустимо. Но это неясный и сбивающий с толку способ сказать, что тип списка является неограниченным; в такой ситуации джокер выражает тот же смысл более кратко и явно:

```
public static void shuffle(List<?> list);
```

Основное преимущество джокера – уменьшение когнитивной нагрузки на разработчиков, использующих метод API. Читатель, который видит параметр-тип, естественно, ищет ограничения, которые он представляет. Напротив, джокер несет в себе сообщение о том, что никаких ограничений на этот тип не наложено.

Используйте обобщенные вспомогательные методы для запоминания джокера

Техника, описанная в этом совете, полезна, когда нужно определить метод с джокерными параметрами.

В разделе «Запоминание джокера» главы 2 мы видели, что наивная реализация метода `Collections::reverse` не годится, потому что компилятор не позволит записать в элементы `List<?>` ссылки на объекты типа `Object`.

org/jgcbook/chapter07/E_use_generic_helper_methods_to_capture_wildcard/Program_1

```
public static void reverse(List<?> list) {
  List<Object> tmp = new ArrayList<>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1)); // ошибка компиляции
  }
}
```

Ошибка возникает из-за невозможности записать из копии обратно в оригинал; мы пытаемся писать из списка объектов в список неизвестного типа:

```
Capture.java:8: error: incompatible types: Object cannot be converted to CAP#1
            list.set(i, tmp.get(list.size()-i-1)); // ошибка компиляции
                                ^
  where CAP#1 is a fresh type-variable: CAP#1 extends Object from capture of ?
```

Однако если временному списку назначить тот же тип, что у параметра метода, то метод компилируется без ошибок.

org/jgcbook/chapter07/E_use_generic_helper_methods_to_capture_wildcard/Program_2

```
public static <T> void reverse(List<T> list) {
  List <T> tmp = new ArrayList<>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1));
  }
}
```

Теперь мы имеем проблему, описанную в предыдущем разделе: открытый API `reverse` определяет переменную-тип без какой-либо семантики.

Решение заключается в том, что соединить обе реализации, поместив джокерное объявление в состав открытого API и реализовав его путем вызова обобщенного закрытого метода:

org/jgcbook/chapter07/E_use_generic_helper_methods_to_capture_wildcard/Program_3

```
public static void reverse(List<?> list) { rev(list); }

private static <T> void rev(List<T> list) {
  List<T> tmp = new ArrayList<>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1));
  }
}
```

Эту технику часто применяют для определения методов, объявления которых содержат джокеры.

Приводите через простые типы, когда необходимо

К этой технике мы прилагаем строгое предупреждение: по возможности избегайте ее!

Приведение через простой тип – способ обойти ограничения дженериков. Эти ограничения существуют не без причины, и, стало быть, у этой техники есть цена: обратное приведение простого типа к параметризованному порождает предупреждение о невозможности проверки, и программист несет полную ответственность за предотвращение исключений приведения класса на этапе выполнения. Но подход может оказаться полезным, когда нужно обойти ограничения дженериков Java.

В качестве примера рассмотрим, как можно было бы модифицировать технику запоминания джокера. Показанный в предыдущем разделе вспомогательный закрытый метод сопровождается небольшой потерей эффективности в форме увеличенного размера файла класса и кеша кода, а также накладных расходов на вызов дополнительного метода. Обычно такие мелочи несущественны, но иногда для критического кода, входящего в состав интенсивно используемой библиотеки, ценна любая возможная оптимизация. Поэтому в методе `Collections::reverse` используется простой тип, как показано ниже.

org/jgcbook/chapter07/F_cast_through_raw_types/Program_1

```
@SuppressWarnings({"rawtypes", "unchecked"})
public static void reverse(List<?> list) {
  List tmp = new ArrayList(list); // непроверенный вызов ArrayList::new
  for (int i = 0; i < list.size(); i++) {
    ((List) list).set(i, tmp.get(list.size()-i-1)); // непроверенный вызов set
  }
}
```

При таком использовании простых типов удается избежать создания и вызова дополнительного метода просто ради того, чтобы заставить дженерики работать. Платить за это приходится предупреждениями о невозможности проверки, которые необходимо подавить, обосновав это подавление: в данном случае типобезопасность гарантирована, потому что элементы, которые мы помещаем в список, из этого же списка и были получены. Эта техника редко находит применение, но знать ее полезно на случай, когда производительность и размер кода критичны.

Используйте параметризованные типы массивов с осторожностью

В этом совете показана полезность параметризованных типов массивов при соблюдении мер предосторожности, рекомендуемых принципом непристойного обнажения.

В разделе «Принцип непристойного обнажения» главы 5 мы видели, что параметризованный тип массива, например `List<Integer>[]` из примера 5.3, является удобной фикцией: поскольку параметризованный тип несберегаемый, типом элемента массив является простой `List`, поэтому можно записать список `List` какого-нибудь другого несберегаемого типа в массив, и это ни приведет ни к ошибке компиляции, на даже к исключению `ArrayStoreException` во время выполнения. Принцип непристойного обнажения предостерегает от

вынесения такого параметризованного типа массива за пределы контекста, в котором массив был создан, потому что система типов в этом случае не обеспечивает никакой безопасности.

Однако при должной осторожности параметризованные типы массивов могут быть полезны. Рассмотрим, к примеру, конструирование недвоичного дерева, в котором каждый узел может иметь произвольное число дочерних узлов. Обычно эту ситуацию моделируют узлом, содержащим список потомков:

org/jgcbook/chapter07/G_use_generic_array_types_with_care/ListTreeNode

```
class ListTreeNode<T> {
  private T data;
  private final List<ListTreeNode<T>> children;
  public ListTreeNode(T data) {
    this.data = data;
    children = new ArrayList<>();
  }
  public void addChild(ListTreeNode<T> child) { this.children.add(child); }
  public void removeChild(ListTreeNode<T> child) { this.children.
remove(child); }
  public T getData() { return data; }
  public List<ListTreeNode<T>> getChildren() { return children; }
}
```

Но в приложениях, где требуется максимальная производительность, а число дочерних узлов может быть очень велико, предпочтительнее может быть массив, содержащий ссылки на дочерние узлы. Тогда добавление и удаление дочерних узлов становится невозможным, зато данные в узлах и даже узлы целиком можно заменять:

org/jgcbook/chapter07/G_use_generic_array_types_with_care/ArrayTreeNode

```
class ArrayTreeNode<T> {
  private T data;
  private final ArrayTreeNode<T>[] children;
  public ArrayTreeNode(T data, int childCount) {
    this.data = data;
    this.children =
      (ArrayTreeNode<T>[]) new ArrayTreeNode[childCount]; // непроверенное
                                                          // приведение
  }
  public void replaceData(T newData) { this.data = newData; }
  public void replaceChild(ArrayTreeNode<T> child, int index) {
    this.children[index] = child;
  }
  public ArrayTreeNode<T>[] getChildren() { return children; }
  public T getData() { return data; }
}
```

И все это будет работать идеально при условии, что ссылки на параметризованный массив содержатся внутри класса, где опасность использования ковариантного подкласса массива очевидна. Однако если API нарушает принцип непристойного обнажения, как поступает метод `getChildren`, то клиент рискует столкнуться с неожиданным исключением приведения класса.

org/jgcbook/chapter07/G_use_generic_array_types_with_care/Program_1

```
ArrayTreeNode<Integer> integerNode = new ArrayTreeNode<>(3, 2);
ArrayTreeNode<? extends Number>[] numberTreeNodes = integerNode.getChildren();
numberTreeNodes[0] = new ArrayTreeNode<>(1.0, 2);
Integer data = integerNode.getChildren()[0].getData(); // исключение
                                                       // приведения класса
```

Используйте маркеры типов для передачи информации о типе времени выполнения

Этот совет показывает, что маркеры типов могут быть полезны не только в ситуациях, где мы с ними встречались ранее, в главе 5.

В разделе «Классная альтернатива» главы 5 мы видели, что маркер типа вида `Class<T>` можно использовать для создания массива элементов типа `T`. Это пример общей техники, полезной для реализации параметризованных методов и классов, которым нужно получать доступ или конкретизировать типы, заданные во время выполнения. Например, обобщенный фабричный метод, определенный в классе с конструктором без аргументов, мог бы выглядеть так:

org/jgcbook/chapter07/H_use_type_tokens_for_run_time_typing/Program_1

```
public static <T> T createInstanceParameterized(Class<T> clazz) {
  try {
    Constructor<T> constructor = clazz.getDeclaredConstructor();
    constructor.setAccessible(true);
    return constructor.newInstance();
  } catch (ReflectiveOperationException e) {
    throw new RuntimeException(e);
  }
}
```

Непараметризованная версия `createInstance` могла бы работать во время выполнения ничуть не хуже:

org/jgcbook/chapter07/H_use_type_tokens_for_run_time_typing/Program_1

```
public static Object createInstanceWildcard(Class<?> clazz) {
  try {
    Constructor<?> constructor = clazz.getDeclaredConstructor();
    constructor.setAccessible(true);
    return constructor.newInstance();
  } catch (ReflectiveOperationException e) {
    throw new RuntimeException(e);
  }
}
```

Но параметризованная версия типобезопасна, поэтому при ее использовании приведение не нужно.

org/jgcbook/chapter07/H_use_type_tokens_for_run_time_typing/Program_1

```
class Foo {};
Foo f1 = createInstanceParameterized(Foo.class);
Foo f2 = (Foo) createInstanceWildcard(Foo.class);
```

Помимо создания объектов, маркеры типов можно использовать также для

типобезопасного доступа к объектам, тип которых неизвестен на этапе компиляции. Например, рассмотрим реализацию реестра, в котором каждый маркер типа отображается на список объектов этого типа.

org/jgcbook/chapter07/H_use_type_tokens_for_run_time_typing/TypeSafeRegistry

```
public class TypeSafeRegistry {
  private Map<Class<?>,List<?>> typeSafeRegistry = new HashMap<>();

  public <T> void addObject(Class<T> type, T object) {
    getObjects(type).add(object);
  }

  @SuppressWarnings("unchecked")
  public <T> List<T> getObjects(Class<T> type) {
    List<?> untypedList =
      typeSafeRegistry.computeIfAbsent(type, k -> new ArrayList<>());
    return (List<T>) untypedList; // непроверенное приведение
  }
}
```

Хотя приведение в методе `getObjects` непроверенное, мы можем быть уверены в его безопасности, потому что метод `addObject` будет принимать только объекты, тип которых совпадает с типом маркера, используемого в качестве ключа.

Заключение

Этой главой мы завершаем первую часть книги. Оценив фундаментальные преимущества дженериков, а также способы борьбы с трудностями, которые их иногда сопровождают, мы готовы перейти к самой важной области их применения: каркасу коллекций Java.

Часть II

Коллекции

Каркас коллекций Java – это набор интерфейсов и классов, собранных в пакеты `java.util` и `java.util.concurrent`. Они предоставляют клиентам различные модели организации объектов и различные реализации каждой модели. Иногда эти модели называют *абстрактными типами данных*, а нам они нужны, потому что в различных программах требуются разные способы организации объектов. В одном случае вам хочется организовать объекты в виде последовательного списка, потому что порядок важен и среди объектов имеются дубликаты. В другом случае подходящим типом является множество, так как порядок не важен, а дубликаты требуется устранить. Эти два типа данных (и другие) представлены разными интерфейсами в каркасе коллекций, и в этой части мы рассмотрим примеры их использования. Но это не всё! Ни один из этих типов не имеет единственной «лучшей» реализации, т. е. такой, которая была бы лучше всех прочих для всех операций. Например, связанный список может быть лучше реализации списка на основе массива, если требуется вставлять элементы в середину списка и удалять их из середины, но для произвольного доступа такая организация гораздо хуже. Поэтому, выбирая реализацию структуры данных для своей программы, вы должны понимать, как она будет использоваться, а также знать, что имеется в вашем распоряжении.

Эта часть книги организована следующим образом.

Глава 8 «Основные интерфейсы каркаса коллекций Java». Здесь мы познакомимся с интерфейсами каркаса коллекций, приведем примеры использования каждого и в общих чертах опишем, как из разрозненных интерфейсов получается единое (по большей части) целое.

Глава 9 «Предварительные сведения». Здесь описываются многочисленные идеи и продукты, оказавшие влияние на дизайн, – это поможет понять, как работает каркас коллекций.

Глава 10 «Интерфейс Collection». Изучается интерфейс `Collection` – изначальный корень каркаса коллекций, и приводится сквозной пример, который мы будем разрабатывать в последующих главах.

Глава 11 «Интерфейс SequencedCollection». Приводится краткое описание `SequencedCollection` – первого нового интерфейса в каркасе коллекций со времен Java 6 и первого типа, который объединяет существующие функции, а не вводит новые.

Глава 12 «Множества». В этой главе начинается детальное изучение основных интерфейсов, включающее описание их операций и реализаций интерфейса `Set`.

Глава 13 «Очереди». Изучается функциональность интерфейса `Queue`, реализации которого поддерживают взаимодействие производителя и потребителя, занимающее центральное положение во многих крупных системах.

Глава 14 «Списки». Изучаются методы и реализации неизменно популярного интерфейса `List`.

Глава 15 «Отображения». Изучается последний из пяти основных интерфейсов каркаса коллекций, `Map`, самый важный для поддержания состояния системы в форме, допускающей конкурентное обновление.

Глава 16 «Класс `Collections`». Описываются специализированные и обобщенные алгоритмы коллекций, собранные в служебном классе `Collections`.

Глава 17 «Наставление по использованию каркаса коллекций Java». Приводятся инструкции по использованию каркаса коллекций Java для проектирования и реализации крупных – и не очень крупных – систем.

Глава 18 «Ретроспективный взгляд на дизайн». Рассматриваются некоторые из важных проектных решений, сформировавших облик каркаса, с точки зрения долгого опыта его использования сообществом.

Глава 8

Основные интерфейсы каркаса коллекций Java

Коллекция – это объект, который предоставляет доступ к группе объектов, позволяя обрабатывать их единообразно. *Каркас коллекций* предлагает единое представление набора типов коллекций благодаря спецификации и реализации общих структур данных, следующих единообразным правилам, которые позволяют им работать совместно. На рис. 8.1 показаны основные интерфейсы каркаса коллекций Java, все они находятся в пакете `java.util` и еще одном – `java.lang.Iterable`, – который не входит в каркас. Интерфейс `Iterable` (см. раздел «Итерируемые объекты и итераторы» главы 9) определяет контракт (см. раздел «Контракты» главы 9), который класс – любой класс, а не только один из входящих в каркас – должен реализовывать, чтобы его можно использовать в «усовершенствованном предложении `for`», которое обычно называется предложением *foreach*.

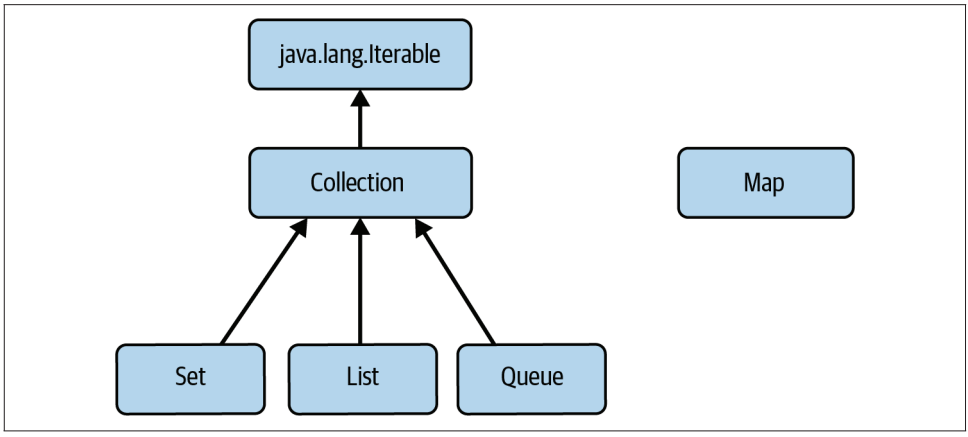


Рис. 8.1. Основные интерфейсы каркаса коллекций Java

Интерфейсы, перечисленные на рис. 8.1, – вместе с подынтерфейсами, которые встретятся нам в этой главе, – определяют структуру и функциональность каркаса коллекций. Выбирать, какие коллекции использовать в программе, следует с учетом описанной здесь функциональности интерфейсов, а пере-

менные должны быть определены в терминах интерфейсов, а не типов реализации – в той мере, в какой это возможно. Выбор реализации производится на более позднем этапе проектирования программы.

Центральным интерфейсом на рис. 8.1 является `Collection`, он раскрывает базовую функциональность, требуемую от любой коллекции, кроме `Map`. Его методы поддерживают добавление и удаление одного или нескольких элементов, проверку принадлежности коллекции одного или нескольких значений, а также инспектирование и экспорт элементов. У него нет конкретной реализации; конкретные классы коллекций реализуют заодно какой-то из его подынтерфейсов.

Использование разных типов коллекций

Мы можем воспользоваться простым примером облака слов, чтобы исследовать, каким образом различные типы коллекций на рис. 8.1 могут быть полезны для реализации решения.

Генераторы облака слов принимают на входе список слов и обычно игнорируют порядок, поэтому на основе следующего списка имен из фильма «Три балбеса» (*https://ru.wikipedia.org/wiki/Три_балбеса*) мог бы быть сгенерирован любой из списков, показанных на рис. 8.2.

```
"Larry", "Curly", "Moe"
```

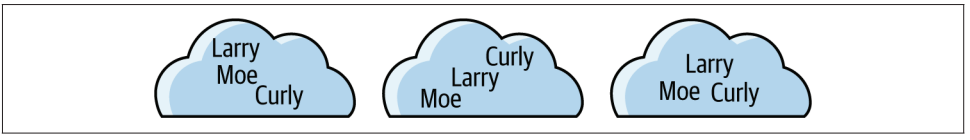


Рис. 8.2. Простое облако слов

Рассмотрим, как мог бы быть представлен вход генератора облака слов. Если вы уже что-то знаете о каркасе коллекций, то можете сами прикинуть варианты, прежде чем читать дальше.

Set

Самый простой способ представления имен – коллекция `Set`, для которой порядок элементов не важен, дубликатов быть не может. Она не добавляет никаких операций сверх имеющихся в `Collection`, но контакты операций, добавляющих элементы, требуют, чтобы они не создавали дубликатов.

Экземпляр `Set`, содержащий имена балбесов, можно создать в оболочке JShell:

```
jshell> Set<String> stoogesSet = Set.of("Larry", "Curly", "Moe")
stoogesSet ==> [Curly, Larry, Moe]
```

Чтобы убедиться в несущественности порядка элементов множества, вы можете выполнить этот код несколько раз в разных экземплярах JShell – порядок имен на выходе будет различаться.

Теперь сделаем задачу чуть более реалистичной. Смысл облака слов заключается в том, что размер шрифта, которым печатается слово, должен за-

виселъ от количества его повторений в данных, поданных на вход генератора. Но поскольку множество `Set` не может содержать дубликатов, оно не является подходящей структурой данных для представления содержимого облака слов. В поисках типа коллекции, которая может содержать дубликаты, мы можем обратиться к списку `List`.

List

`List` – это коллекция, в которой порядок элементов важен и которая может содержать дубликаты. Она добавляет операции, поддерживающие доступ по индексу.

Облако слов, сгенерированное следующим кодом в JShell,

```
jshell> List<String> stoogesList = List.of("Larry", "Curly", "Larry", "Moe")
stoogesList ==> [Larry, Curly, Larry, Moe]
```

могло бы выглядеть, как показано на рис. 8.3. Оно отражает тот факт, что строка `"Larry"` встречается в два раза чаще, чем две другие. Хотя порядок слов в каждом облаке по-прежнему случайный, это не потому, что при помещении в коллекцию информация о порядке элементов теряется; напротив, список сохраняет порядок. Но порядок может быть проигнорирован генератором облака слов.

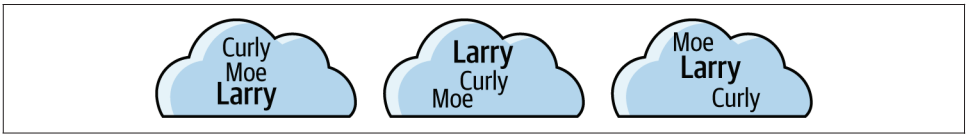


Рис. 8.3. Облака слов, сгенерированные на основе данных в `List`

Поскольку любое множество `Set` может быть представлено списком `List`, зачем вообще нужен отдельный тип данных? Причина в том, что операции `Set` позволяют с гораздо меньшими затратами гарантировать уникальность элементов в случае, когда нам нужно именно это. А возможность игнорировать порядок позволяет сделать операции `Set` гораздо более эффективными.

Но в данном случае важно сохранить количество вхождений каждого элемента, поэтому `Set` не подходит. `List` лучше, но является ли этот выбор самым лучшим? У списка есть два существенных недостатка: он сохраняет произвольный (в данном примере бессмысленный) порядок и, что важнее, для подсчета количества вхождений одного элемента необходимо просмотреть все элементы списка. В нашем простеньком примере это, может быть, и не важно, но реальные структуры данных могут насчитывать десятки миллионов элементов.

Поэтому еще лучшим представлением входного списка было бы отображение `Map` – коллекция, способная создавать ассоциации слов с частотой вхождения и эффективно находить эти ассоциации (или *записи*).

Map

В коллекции `Map` для хранения и поиска элементов используются записи, содержащие ключ и значение. Ключи `Map` образуют множество `Set`, т. е. среди них не может быть дубликатов, а их порядок неважен. Операции `Map` позволяют добавлять или удалять одну или несколько записей ключ–значение и произво-

дить поиск этих записей. Отображения используются в ситуациях, когда требуется, чтобы операции чтения-записи были простыми и, главное, быстрыми.

Показанный ниже фрагмент кода в JShell генерирует отображение `Map`, в котором хранятся частоты имен в нашем облаке слов:

```
jshell> Map<String,Integer> stoogesMap = Map.of("Curly", 1, "Larry", 2, "Moe", 1)
stoogesMap ==> {Curly=1, Moe=1, "Larry", 2}
```

Вы можете представлять себе `Map` как таблицу типа, показанную на рис. 8.4.

Рис. 8.4. Данные облака слов, представленные в виде `Map`

Все три таблицы на рис. 8.4 представляют одно и то же отображение `Map`, потому что строки таблицы, т. е. пары ключ–значение, составляющие `Map`, образуют множество `Set`, так что их порядок неважен; если выполнить этот скрипт в разных экземплярах JShell, то вы увидите разный порядок пар. Кроме того, сами ключи образуют множество, поэтому не может быть двух строк с одинаковым ключом (скажем, `"Larry"`). Таким образом, у `Map` нет ни одного из двух недостатков, свойственных `List`: не нужно сохранять порядок пар ключ-значение, а для нахождения частоты вхождений любого элемента требуется только поиск в таблице, который, как мы увидим, может быть очень быстрым.

Но и отображений может оказаться недостаточно. Для достижения максимальной плотности облака реальные приложения облаков слов размещают «самые большие» слова (с наибольшей частотой, а значит, и размером шрифта) сначала, а затем окружают меньшими словами[1]. Поэтому в нашем примере нужно, чтобы записи таблицы были упорядочены специальным образом. Для такой цели каркас коллекций предоставляет интерфейс `SequencedMap`, он появился в версии Java 21 для унификации ранее существовавших возможностей различных реализаций `Map`.

SequencedMap

`SequencedMap` – это отображение `Map`, в котором записи хранятся в определенном порядке. Некоторые реализации `SequencedMap` автоматически сортируют записи в порядке ключей. В показанном ниже коде одна такая реализация (`TreeMap`) используется для создания отображения, показанного на рис. 8.5; ключами являются частоты слов, перечисленные в порядке убывания, и каждая частота отображается на множество слов с такой частотой. Если вы не знакомы с потоками Java, не старайтесь понять этот довольно сложный код во всех деталях; нужно только знать, что переменная `stoogesSequencedMap` представляет данные

[1] Весь процесс довольно сложен, подробнее о нем можно прочитать в работе Steele and Iliinsky (2010, глава 3).

в форме, пригодной для алгоритма построения облака слов. Если алгоритм просто перебирает записи по одной, то он будет видеть их в порядке убывания размера шрифта. В каждой записи значение ссылается на множество слов, которые должны быть напечатаны шрифтом размера, определяемого ключом:

```
jshell> import static java.util.stream.Collectors.*
jshell> SequencedMap<Integer, Set<String>> stoogesSequencedMap =
...>     Map.of("Curly", 1, "Larry", 2, "Moe", 1)
...>         .entrySet().stream()
...>         .collect(groupingBy(Map.Entry::getValue,
...>          () -> new TreeMap<Integer,Set<String>>(Comparator.reverseOrder()),
...> mapping(Map.Entry::getKey, toSet())));

stoogesSequencedMap ==> {2=[Larry], 1=[Moe, Curly]}
```

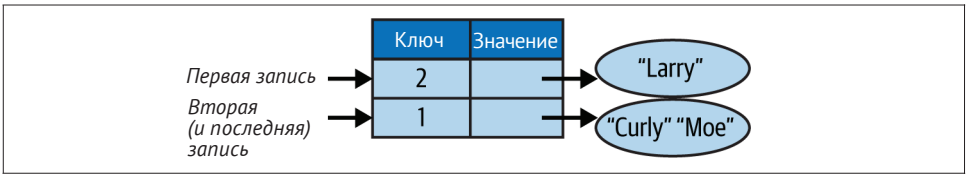


Рис. 8.5. Данные облака слов, представленные в виде `SequencedMap`

Queue

`Queue` отличается от коллекций, которые встречались нам до сих пор. Множества, списки и отображения обычно «принадлежат» какому-то другому объекту и являются частью его состояния (см. раздел «Не забывайте о "владельцах" коллекций» главы 17, где эта идея и ее следствия описываются более подробно). Напротив, очереди обычно не принадлежат какому-то одному объекту, а используются для передачи значений от *производителей потребителям*. У очереди может быть несколько производителей и несколько потребителей; это могут быть объекты, потоки или процессы.

Хотя `Queue` наследует операции `Collection`, эти операции используются не так, как в других типах коллекций: производители вызывают методы, которые добавляют элементы в *конец* очереди, а потребители – методы, которые удаляют элементы из ее *начала*. Таким образом, информация передается от производителей потребителям. Как и у других коллекций, у очереди есть память, но эта память временная, элементы в ней хранятся только между моментом *помещения* в очередь производителем и моментом *извлечения* из очереди потребителем. Поэтому очереди следует рассматривать как каналы или трубы, по которым информация курсирует между объектами, а не как часть состояния какого-то другого объекта.

Упорядоченные коллекции

`SequencedMap` – лишь одна из нескольких коллекций, сохраняющих порядок, а иногда и навязывающих его. Такие *упорядоченные коллекции* отличаются от `Collection`, `Set` и `Map` тем, что имеют определенный порядок, который в документации называется *порядком вхождения* (encounter order, см. главу 11). Они отличаются и от

очереди `Queue`, для которой порядок тоже определен, тем, что обходить их можно в обоих направлениях. Порядок упорядоченных коллекций может быть определен двумя способами: иногда, как в случае `List`, элементы сохраняют порядок, в котором добавлялись, а иногда, как в случае `NavigableSet` (см. раздел «SequencedSet и NavigableSet» ниже), порядок диктуется значениями элементов. Эти типы называются соответственно *внешне упорядоченными* и *внутренне упорядоченными*, что отражает различие между порядком, произвольно навязанным элементам, например порядком добавления, и порядком, являющимся свойством, внутренне присущим самим элементам, например,алфавитным порядком строк.

Для унификации упорядоченных коллекций обоих видов в Java 21 был включен новый интерфейс, `SequencedCollection`. На рис. 8.6 показано, как стал выглядеть рис. 8.1 после добавления в каркас упорядоченных интерфейсов.

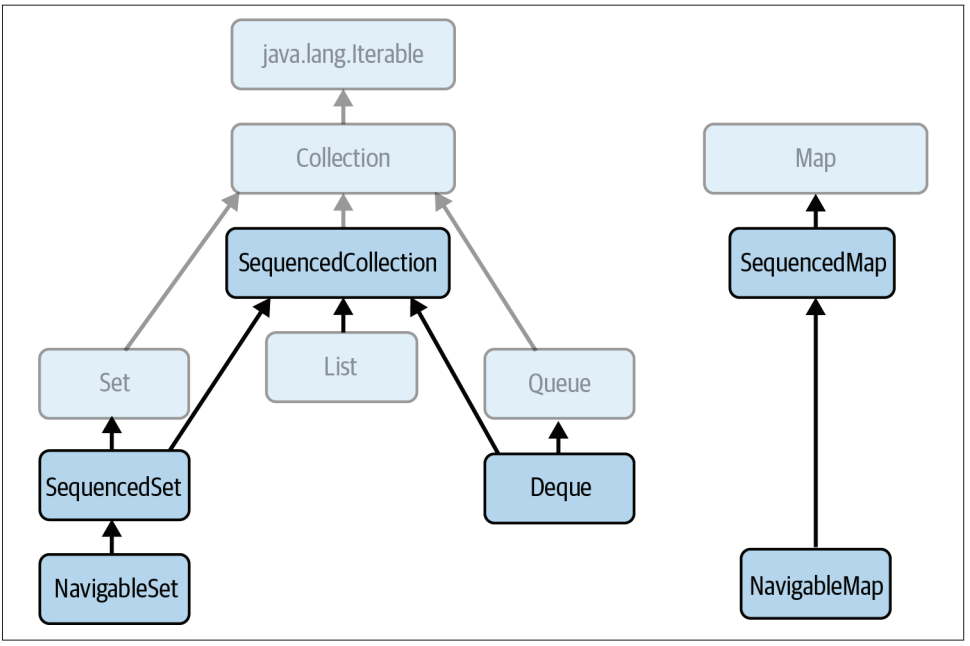


Рис. 8.6. Упорядоченные интерфейсы в каркасе коллекций Java

SequencedCollection

Вообще говоря, коллекции поддерживают операции добавления (`add`), удаления (`remove`) и в некоторых случаях доступа к элементам (`get`). Интерфейс `SequencedCollection` предоставляет варианты этих операций, применимые к первому и последнему элементу коллекции: `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst` и `getLast`. Дополнительно `SequencedCollection` предоставляет обратное представление, т. е. способ работать с коллекцией так, будто порядок элементов изменен на противоположный. Это упрощает многие проблемы программирования и часто дает более эффективные реализации. Мы подробно рассмотрим упорядоченные коллекции и их представления в главе 11 и в разделе «Представления» главы 9.

SequencedSet и NavigableSet

`SequencedSet` – это внешне или внутренне упорядоченное множество `Set`, раскрывающее также методы `SequencedCollection`. `NavigableSet` – это внутренне упорядоченное `SequencedSet`, которое, следовательно, автоматически сортирует свои элементы и предоставляет дополнительные методы для нахождения элементов, соседствующих с целевым значением.

Deque

`Deque` – это двусторонняя очередь (или дек), которая может принимать и отдавать элементы с обеих сторон. `Deque`, как и `Queue`, можно использовать в качестве канала для передачи информации между производителями и потребителями. Возможность удалять элементы из конца очереди позволяет реализовать заимствование работы – технику балансировки нагрузки, при которой простаивающие потоки «заимствуют» задачи у более занятых потоков, чтобы максимизировать степень параллелизма. Деки также можно использовать для хранения состояния объекта, если обновление состояния требует выполнения операций с обеих сторон.

SequencedMap и NavigableMap

`SequencedMap` – это `Map`, ключи которого образуют `SequencedSet`. `NavigableMap` – это `SequencedMap`, ключи которого образуют `NavigableSet`, так что элементы автоматически сортируются в порядке ключей, а методы позволяют находить ключи и пары ключ–значение, соседствующие с целевым значением ключа.

Заключение

Целью краткого обзора в этой главе было поверхностное знакомство с основными возможностями каркаса коллекций. В главах 10–15 мы по очереди рассмотрим все интерфейсы. Но сначала в главе 9 обсудим фундаментальные идеи, положенные в основу дизайна каркаса коллекций; это поможет использовать коллекции более эффективно.

Глава 9

Предварительные сведения

Для освоения любого каркаса понять идеи, лежащие в основе его дизайна, так же важно, как познакомиться с деталями реализации. В этой главе мы рассмотрим концепции каркаса коллекций, послужившие основанием для проектирования отдельных классов.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter09*.

Итерируемые объекты и итераторы

Чтобы удовлетворить очень общему требованию единообразной обработки каждого элемента коллекции, мы должны уметь обходить ее, или *итерировать*. Эта возможность предоставляется тремя разными способами с помощью интерфейса `java.lang.Iterable<E>`. Поскольку это суперинтерфейс `Collection`, его методы применимы к любой коллекции, кроме отображений. В интерфейсе объявлены три метода:

`void forEach(Consumer<? super T> action)`	Выполняет действие `action` для каждого элемента `Iterable`
`Iterator<T> iterator()`	Возвращает итератор по элементам типа `T`
`Spliterator<T> spliterator()`	Создает `Spliterator` по элементам, описываемым этим итерируемым объектом

Обсуждение сплитераторов (`Spliterator`) мы отложим до раздела «Параллельные потоки» ниже. Но остальные два метода понадобятся уже здесь. Первый предлагает самый простой способ обхода коллекции; например, чтобы напечатать каждый элемент коллекции `coll` на терминале, мы можем написать:

```
coll.forEach(System.out::println);
```

Но если действие, совершаемое на каждой итерации, сложнее, чем можно без напряжения выразить одним потребителем `Consumer`, то `Iterator`, возвращенный вторым методом, можно использовать двумя способами. Интерфейс `Iterator` объявляет три метода:

`boolean hasNext()`	Возвращает `true`, если больше элементов нет
`E next()`	Возвращает следующий элемент коллекции
`void remove()`	Удаляет последний элемент, возвращенный итератором

Пользуясь стандартной идиомой явного использования итератора, предыдущий пример можно было бы записать так (скажем, для коллекции строк):

```
for (Iterator<String> itr = coll.iterator() ; itr.hasNext() ; ) {
  System.out.println(itr.next());
}
```

Эта запись более громоздка, чем предложение *foreach*, которое мы рассмотрим ниже, поэтому используется она реже. Она важна, когда в процессе итерирования требуется внести *структурное* изменение в коллекцию – добавить или удалить элементы. (Как мы только что видели, `Iterator` объявляет только метод удаления элементов из коллекции, но его подынтерфейс `ListIterator`, доступный реализациям `List`, предоставляет также методы для добавления и замены элементов.) Бывают также ситуации, когда необходимо управлять итерированием в зависимости от значений встретившихся элементов; см. пример 10.2.

Если не считать этих случаев, то предложение *foreach*, в котором `Iterator` используется на внутреннем уровне, удобнее. Примеру выше соответствует следующий код на основе *foreach*:

```
for (String s : coll) {
  System.out.println(s);
}
```

Предложение *foreach* позволяет обходить массив или любой класс, который реализует интерфейс `Iterable`. Поскольку интерфейс `Collection` расширяет `Iterable`, любое множество, список или очередь можно обойти с помощью *foreach*. И вашу собственную реализацию `Iterable` тоже можно будет использовать с *foreach*. Например, в следующем фрагменте объект `Counter` инициализируется счетчиком целых чисел, а итератор по нему возвращает эти числа в порядке возрастания, начиная с 1, в ответ на вызовы `next`:

org/jgcbook/chapter09/A_iterable_and_iterators/Counter

```
class Counter implements Iterable<Integer> {
  private final int count;
  public Counter(int count) { this.count = count; }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      private int i = 0;
      public boolean hasNext() { return i < count; }
      public Integer next() { i++; return i; }
      public void remove() { throw new UnsupportedOperationException(); }
    };
  }
}
```

Теперь объекты `Counter` можно использовать в предложении *foreach*:

org/jgcbook/chapter09/A_iterable_and_iterators/Counter

```
int total = 0;
for (int i : new Counter(3)) {
  total += i;
}
assert total == 6;
```

На практике `Iterable` редко реализуют непосредственно, поскольку *foreach* в основном используется в сочетании с массивами и стандартными классами коллекций. Использовать итераторы напрямую вместо предложения *foreach* необходимо главным образом тогда, когда в процессе обхода коллекции требуется внести в нее *структурное* изменение – добавить, изменить или удалить элементы.

Большинство программистов рано или поздно совершают типичную ошибку: в процессе итерирования производят структурное изменение непосредственно, вместо того чтобы вызвать один из методов итератора. Если коллекция, которую вы обходите, – одна из стандартных реализаций `Collection` – `ArrayList`, `HashMap` и т. д., – то вы с удивлением увидите исключение `ConcurrentModificationException` в однопоточном коде. Итераторы по этим коллекциям возбуждают это исключение, когда обнаруживают, что коллекция, от которой они получены, подверглась структурной модификации в обход самого итератора. Например, следующий код возбуждает `ConcurrentModificationException`:

org/jgcbook/chapter09/A_iterable_and_iterators/Program_1

```
List<String> strings = new ArrayList<>(List.of("alpha", "bravo", "charlie"));
for (String s : strings) {
  if (! s.contains("r")) {
    strings.remove(s); // возбуждает ConcurrentModificationException
  }
}
```

Обоснованием такого поведения служит тот факт, что структурное изменение, произведенное в процессе обхода, – возможное свидетельство того, что к коллекции обратился другой поток; если вы хотите, чтобы совершающий обход поток вносил структурные изменения, то должны делать это с помощью итератора. Позволив другому потоку обращаться к потоконебезопасной коллекции, мы, скорее всего, получим ошибку впоследствии, когда ее будет трудно диагностировать. Чтобы избежать этой проблемы, итераторы общего назначения в каркасе коллекций обладают свойством *быстрого отказа*: их методы проверяют, были ли внесены структурные модификации с момента последнего вызова итератора, и если да, то возбуждают исключение `ConcurrentModificationException`. Резоны, стоящей за этой политикой, обсуждаются в разделе «Конкурентная модификация» главы 18.

В исправленной версии кода для удаления элементов используется сам итератор:

org/jgcbook/chapter09/A_iterable_and_iterators/Program_2

```
List<String> strings = new ArrayList<>(List.of("alpha", "bravo", "char-
lie"));
for (Iterator<String> itr = strings.iterator() ; itr.hasNext() ; ) {
  String s = itr.next();
```

```
    if (! s.contains("r")) {
      itr.remove();
    }
  }
  assert strings.equals(List.of("bravo", "charlie"));
```

Эта стратегия работает, только если итератор раскрывает метод для внесения нужных структурных модификаций. Все итераторы раскрывают метод `remove`, но только итераторы списка, полученные от метода `listIterator`, умеют добавлять элементы в конец или в середину списка.

Потоковая альтернатива

Если ограничения на объем памяти не мешают создать новую копию списка, то потоки предлагают более элегантное решение этой проблемы:

org/jgcbook/chapter09/A_iterable_and_iterators/Program_3

```
List<String> strings = new ArrayList<>(List.of("alpha", "bravo", "charlie"));
List<String> modifiedStrings = strings.stream()
    .filter(s -> s.contains("r"))
    .toList();
assert modifiedStrings.equals(List.of("bravo", "charlie"));
```

У конкурентных коллекций есть и другие стратегии обработки конкурентных модификаций, например слабо согласованные итераторы. Мы обсудим их в разделе «Коллекции и потокобезопасность» ниже в этой главе.

Реализации

Мы кратко рассмотрели интерфейсы каркаса коллекций, которые определяют ожидаемое от коллекции поведение. Но каждый из этих интерфейсов можно реализовать несколькими способами. Почему бы не включить в каркас только лучшую реализацию каждого интерфейса? Ведь это здорово упростило бы жизнь – до такой степени, что не осталось бы ничего жизнеспособного. Если реализация быстра, как гончая, при решении одних задач, то по закону Мёрфи она была бы медленной, как черепаха, при решении других. Раз ни для какого интерфейса не существует «наилучшей» реализации, которая подходила бы для всех ситуаций, то вам придется искать компромиссы, основанные на том, какие операции встречаются в вашем приложении чаще всего, и выбирать из реализации ту, что оптимизирует именно эти операции.

Основные три вида операций, которые требуются от большинства интерфейсов коллекций, – вставка и удаление элементов в указанной позиции, поиск элементов в содержимом и обход (итерирование) элементов коллекции. Разные реализации предлагают разные варианты операций, но основные различия между ними обсуждаются в терминах выполнения этих трех. В этом разделе мы кратко рассмотрим четыре основные структуры, положенные в основу реализаций, а впоследствии, когда они понадобятся, мы обсудим и детали.

Массивы

Эти структуры знакомы по языку Java – да и практически любому языку программирования, начиная с Fortran. Поскольку массивы реализованы на уровне оборудования, они обладают свойством произвольного доступа к элементам, но вставка и удаление элементов в произвольной позиции производятся медленнее (потому что приходится перемещать другие элементы). В каркасе коллекций массивы положены в основу структур `ArrayList`, `CopyOnWriteArrayList`, `EnumSet`, `EnumMap` и многих реализаций `Queue` и `Deque`. Они также образуют важную часть механизма реализации хеш-таблиц (см. ниже).

Линейные связанные списки

Как следует из названия, речь идет о цепочках связанных звеньев. Каждое звено содержит ссылку на данные и ссылку на следующее звено списка (а в некоторых реализациях еще и на предыдущее звено). Характеристики производительности связанных списков совершенно не такие, как у массивов: доступ к элементам по номеру позиции медленный, зато операции вставки и удаления выполняются за постоянное время, так как сводятся к изменению ссылок на звенья. Связанные списки – базовая структура данных в классах `ConcurrentLinkedQueue`, `LinkedBlockingQueue` и `LinkedList`.

Другие связанные структуры данных

Связанные структуры особенно полезны для представления нелинейных типов, таких как деревья и списки с пропусками (см. раздел «ConcurrentSkipListSet» главы 12), особенно если при добавлении элементов их нужно реорганизовывать. Такие структуры предлагают недорогой способ поддержания данных в отсортированном порядке, что обеспечивает быстрый поиск по содержимому. Деревья – базовые структуры данных в классах `TreeSet` и `TreeMap`. Списки с пропусками используются в классах `ConcurrentSkipListSet` и `ConcurrentSkipListMap`.

Хеш-таблицы

Предоставляют способ хранения элементов с индексированием по содержимому, а не по целочисленной позиции, как в списках. В отличие от массивов и связанных списков хеш-таблицы не поддерживают доступ к элементам по номеру позиции, зато доступ по содержимому, равно как вставка и удаление, обычно очень быстрый. Хеш-таблицы – базовая структура для многих реализаций `Set` и `Map`, включая `HashSet` и `LinkedHashSet`, `HashMap` и `LinkedHashMap`, а также `WeakHashMap`, `IdentityHashMap` и `ConcurrentHashMap`.

Индексирование хешированных коллекций опирается на два метода `Object`: `hashCode` и `equals`, которые применяются для нахождения позиции, куда нужно вставить новый элемент, и для поиска уже имеющегося элемента. Как эти методы должны быть связаны между собой, определено в контракте `hashCode`. Важнейшее требование, на которое Java-программисты иногда не обращают внимания – с катастрофическими последствиями, – заключается в том, что если метод `equals` говорит, что два объекта равны, то результаты вызова для них `hashCode` должны быть одинаковы. Если `hashCode` зависит от поля экземпляра или любых других данных об объекте, которые

не используются в методе `equals`, то первый этап операции поиска, скорее всего, приведет не туда, куда надо. Это может случиться, в частности, если вы забудете переопределить метод `Object::hashCode`; в этом случае возвращаемое им значение будет независимо от реализации (в OpenJDK оно обычно генерируется случайно), но почти наверняка не будет одинаковым для двух разных экземпляров.

Проиллюстрируем эту проблему на модельном классе `Person`:

org/jgcbook/chapter09/B_implementations/Person

```
class Person {
  private final String name;
  public Person(String name) {
    this.name = name;
  }
  public boolean equals(Object o) {
    return o instanceof Person p && name.equals(p.name);
  }
}
```

Этот класс должен был бы переопределить `hashCode`, так чтобы он зависел от того же поля, что и `equals` (т. е. от `name`). Но это не так, и в результате хешированные коллекции будут работать с ним неправильно.

org/jgcbook/chapter09/B_implementations/Person

```
Set<Person> people = new HashSet<>();
people.add(new Person("Alice"));
assert ! people.contains(new Person("Alice"));
```

Представления

В каркасе коллекций *представление* структуры данных дает способ работать с ней так, будто она была каким-то образом преобразована – либо в по-другому организованную структуру того же типа, либо в структуру совершенно другого типа. Самыми очевидными примерами являются базовые структуры, рассмотренные в предыдущем разделе. Но хотя, например, замена элемента в `ArrayList` реализована путем замены элемента в базовом массиве, мы обычно не называем `ArrayList` представлением массива. Дело в том, что `ArrayList` – больше чем представление; он может изолировать пользователя от ограничений массивов. Например, в список часто добавляются элементы. Массивы не могут поддержать эту операцию, но `ArrayList` скрывает это ограничение и, если необходимо, перемещает все элементы в новый базовый массив большего размера. Поэтому `ArrayList` – гораздо больше, чем простое представление одного массива, в связи с чем этот термин обычно не используется для его описания.

Но иногда простое представление – то, что доктор прописал. Например, допустим, что нам нужно проверить присутствие конкретного объекта в массиве. Очевидный способ – перебрать элементы массива, проверяя каждый на равенство с искомым. Альтернатива – воспользоваться методом `contains` интерфейса `List`, поручив ему сделать всю работу. Но массив – это не `List`, поэтому чем может быть полезен метод `List` для обработки массива? Конечно, мы хотели бы избежать затрат на создание нового объекта `ArrayList` и физического копи-

рования всех элементов массива в новую коллекцию. В такой ситуации лучше получить *представление* массива в виде списка `List` – объекта, который «выглядит» как `List`, но реализует все свои операции применительно к базовому массиву. Метод `asList` служебного класса `Arrays` как раз и дает такое представление. Возвращаемое им простое представление поддерживает некоторые операции `List`, например `contains`, а также методы `get` и `set` для получения или изменения элементов массива, но не допускает *структурных изменений*, т. е. добавления и удаления элементов, потому что они не поддерживаются базовым массивом.

Данные представления фактически находятся в базовой структуре, поэтому изменения, внесенные в эту структуру, становятся сразу видны в представлении и наоборот. Например, следующий код компилируется и выполняется без ошибок:

org/jgcbook/chapter09/C_views/Program_1

```
Integer[] arr = {1, 2, 3};
var list = Arrays.asList(arr);
list.set(0, 3);                 // изменили списковое представление...
assert arr[0] == 3;             // и базовый массив тоже изменился
arr[2] = 0;                     // теперь изменили базовый массив...
assert list.get(2) == 0;        // и списковое представление тоже изменилось
```

API коллекций раскрывает много методов, возвращающих представления. Например, ключи, равно как и значения, `Map` можно рассматривать как `Set`; коллекции можно трактовать как немодифицируемые и т. д. У каждого из этих представлений свои правила, определяющие, какие модификации принимаются и отражаются на базовой коллекции. В порядке убывания позволительности:

- все изменения;
- некоторые структурные и все неструктурные модификации;
- только неструктурные модификации;
- запрещены любые модификации (полностью немодифицируемое представление).

Поэтому некоторые операции интерфейсов, реализуемых этими представлениями, помечены как *факультативные*. Это, пожалуй, самый спорный аспект дизайна каркаса коллекций Java. В главе 18 и в разделе «Контракты» ниже этот вопрос обсуждается более подробно; сами же представления будут обсуждаться в последующих главах, каждое в контексте своей базовой коллекции.

В общем случае представления можно компоновать для операций чтения, и зачастую они коммутативны, т. е. могут применяться в любом порядке. Например:

```
List<String> names = List.of("alpha", "bravo", "charlie", "delta");
List<String> reverseThenSublist = names.reversed().subList(1, 3);
List<String> subListThenReverse = names.subList(1, 3).reversed();
assert reverseThenSublist.equals(subListThenReverse);
```

Производительность

Полезность каркаса коллекций зависит от того, насколько производительность коллекций влияет на работу системы в целом. К сожалению, ее очень трудно предсказать теоретически и гарантировать практически. На нее влияют многие факторы, в том числе:

- как часто выполняются операции коллекции;
- какие операции выполняются наиболее часто;
- временная сложность каждой выполняемой операции;
- сколько порождается объектов-сирот (на которые не осталось ссылок) и каковы накладные расходы на их уборку в мусор;
- свойства локальности коллекции (обсуждаются в следующем подразделе);
- степень параллелизма на уровне команд и потоков.

Изучение того, как сочетание этих факторов влияет на быстродействие реальной системы – предмет настройки производительности, а самое важное и часто цитируемое правило в этой области было сформулировано Дональдом Кнутом в работе Donald Knuth (1974): «Преждевременная оптимизация – корень всех зол». Смысл этого замечания двоякий. Во-первых, для многих программ производительность вообще не критична. Если программа выполняется редко или и так потребляет мало ресурсов, то оптимизация – бессмысленная трата сил и даже может причинить вред. Во-вторых, даже для программ, которые должны работать быстро, оценка того, какая *именно* часть критична, обычно требует точных измерений; в той же статье Кнут добавляет: «Часто является ошибкой вынесение *априорных* суждений о том, какие части программы действительно критичны, поскольку совокупный опыт программистов, использовавших инструменты измерения, показывает, что их интуитивные догадки неверны».

Отсюда вытекает важное следствие, касающееся сравнения производительности различных коллекций. Например, `CopyOnWriteArrayList` обеспечивает очень эффективные конкурентные операции чтения ценой очень дорогих операций записи. Поэтому если вы хотите использовать ее в системе, требующей высокопроизводительного конкурентного доступа к списку, то должны быть уверены – и при необходимости подтвердить это измерениями, – что операций чтения намного больше, чем операций записи.

Память

Традиционно алгоритмы оценивались по потреблению двух ресурсов: времени и памяти. Алгоритмы из каркаса коллекций, вообще говоря, могут похвастаться эффективным использованием памяти, а вкупе с экспоненциальным уменьшением стоимости памяти на протяжении 50 лет, начиная с 1970-х годов, это означает, что пространственная сложность на протяжении длительного времени привлекала меньше внимания, чем временная (см. раздел «Счетчик команд и нотация *O* большое» ниже в этой главе). Но современные тенденции в архитектуре оборудования и программного обеспечения усложнили картину и вновь выдвинули проблемы памяти на первый план. Программные соображения относительно просты: сборка мусора может добавить заметные накладные расходы к работе программы, часто выделяющей память.

Влияние дизайна оборудования нуждается в более подробных пояснениях. На протяжении сорока лет, начиная с 1970-х годов, быстродействие процессоров росло экспоненциально со скоростью, намного превосходящей все остальные компоненты, включая саму память и шину памяти – компоненты, которые в совокупности отвечают за то, чтобы процессоры не простаивали в ожидании данных и команд процессорам. Различные виды доступной памяти подчиня-

ются одному и тому же правилу, общему для всех технологий систем хранения, включая память на плате, флеш-память, дисковые накопители и даже ленты (которые все еще используются для хранения архивов): стоимость хранения тем выше, чем больше быстродействие носителя. Чем быстрее носитель может сохранять и извлекать данные, тем больше стоимость в расчете на один байт емкости. Это заставляет проектировщиков создавать *иерархии памяти*, в основании которых находится много дешевой памяти, например дисков и даже лент, а на вершине – небольшие объемы дорогой статической RAM, расположенной на кристалле, физически близко к процессору. На рис. 9.1 показана упрощенная картина верхней части иерархии – компоненты, расположенные на плате и на кристалле. Память на кристалле организована в виде кешей первого, второго и (иногда) третьего уровня, каждый из которых медленнее, больше по размеру и менее энергоемкий, чем предыдущий.

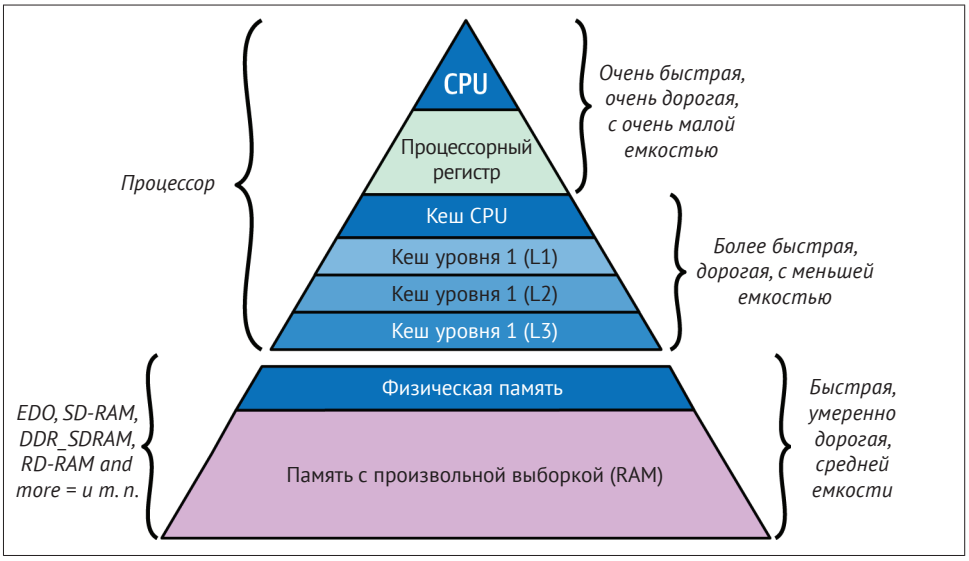


Рис. 9.1. Иерархия памяти

Если данные отсутствуют на одном уровне, то они ищутся на более низких уровнях. Если этот уровень – кеш, то говорят о *непопадании в кеш*. Данные в кешах организованы в виде блоков, называемых *строками кеша*, обычно длиной 64 байта. Если попытка найти байт данных закончилась непопаданием в кеш, то всю строку кеша нужно отбросить и загрузить на ее место байт из более низкого уровня иерархии, а вместе с ним еще 63 близких байта. Эти дорогостоящие непопадания в кеш случаются реже в программах, которые соблюдают принцип *пространственной локальности*, гласящий, что программы склонны повторно использовать данные и команды, близкие к тем, что использовались недавно. Две программы, исполняющие одно и то же число команд, могут резко различаться по времени работы, если одна из них пространственно локальна, а другая нет.

Пространственная локальность объектно ориентированных программ часто низкая, потому что объекты могут располагаться в любом месте памяти, но у некоторых программ она особенно плохая. Например, программа, которая обхо-

дит массив элементов примитивного типа, демонстрирует хорошую пространственную локальность и предсказуемую закономерность доступа к данным, поэтому последовательное чтение данных уменьшает количество непопаданий в кеш. Обход массива ссылочных типов хуже, потому что, хотя доступ к самому массиву предсказуем и локализован, о доступе к объектам по ссылкам этого не скажешь. Связанная структура и того хуже: доступ к памяти в общем случае и не предсказуем, и не локализован, потому непопадание в кеш случается часто и обходится дорого. Каждый дополнительный уровень косвенности повышает вероятность непопадания в кеш и еще более дорогого страничного отказа.

Счетчик команд и нотация О большое

Хотя, как мы заметили в предыдущем разделе, потребление памяти больше нельзя отделять от времени выполнения, все же есть смысл попытаться независимо оценить последнее. Традиционно предполагалось, что время выполнения пропорционально количеству процессорных операций, необходимых для завершения программы. Это предположение также требует некоторых уточнений, которые мы обсудим в конце этого раздела. Но сначала зададимся вопросом, как оценить количество команд? Детальный анализ может оказаться сложным. Сравнительно простой пример приведен в классической книге Дональда Кнута «Сортировка и поиск» (Donald Knuth 1998), где выводится количество команд для сортировки нескольких списков вставками на модельной машине MIX:

$$3.5N^2 + 24.5N + 4M + 2,$$

где N – число элементов списка, а M – число списков.

Для краткого описания эффективности алгоритма такой способ удобным не назовешь. Очевидно, что для использования в общем случае нужен более широкий мазок. Наиболее употребительна нотация *О большое*. Это способ абстрактного описания эффективности алгоритма без деталей, необходимых для точной оценки производительности конкретной программы, работающей на конкретной машине. Основная причина популярности этой нотации в том, что она позволяет описать, как время выполнения алгоритма зависит от размера набора данных в предположении, что последний достаточно велик. Например, в выражении выше первые два члена сравнимы по величине для малых N; на самом деле для $N < 7$ второй член больше. Но с ростом N первый член начинает преобладать, и, когда N достигает 100, первый член оказывается в 15 раз больше второго. Рисуя очень широкими мазками, мы говорим, что в худшем случае этот алгоритм требует времени порядка $O(N^2)$. Нам не так уж важен коэффициент пропорциональности, потому что он не существен для ответа на единственно важный вопрос о любом алгоритме: «Что будет со временем выполнения, когда размер данных возрастет – скажем, удвоится»? Для вышеупомянутой сортировки вставками в худшем случае ответ – «возрастет в 4 раза». Из-за этого оценка $O(N^2)$ довольно плоха – хуже любой, которая встретится нам в этой книге.

В табл. 9.1 показаны некоторые типичные оценки времени выполнения с примерами алгоритмов, к которым они применимы. Есть много других оценок, в том числе гораздо худших, чем в таблице. Большой класс важных задач можно решить только с помощью алгоритмов, требующих $O(2^N)$ шагов – т. е. когда N уд-

ваивается, счетчик команд возводится в квадрат! Для любых наборов данных, кроме совсем уж маленьких, это неприемлемо медленно, хотя квантовые компьютеры смогут изменить эту ситуацию для некоторых важных задач.

Таблица 9.1. Некоторые часто встречающиеся оценки *O* большое

Время	Стандартное название	Изменение счетчика команд при удвоении *N*	Примеры алгоритмов
$O(1)$	Постоянное	Не изменяется	Вставка в хеш-таблицу (см. раздел «Реализации интерфейса Set» главы 12)
$O(\log N)$	Логарифмическое	Увеличивается на постоянную величину	Вставка в дерево (см. раздел «TreeSet» главы 12)
$O(N)$	Линейное	Удваивается	Линейный поиск
$O(N \log N)$		Удваивается и увеличивается на величину, пропорциональную *N*	Сортировка слиянием (см. раздел «Изменение порядка элементов списка» главы 16
$O(N^2)$	Квадратичное	Возрастает в четыре раза	Сортировка вставками в худшем случае (когда входные данные отсортированы в обратном порядке)

Иногда приходится рассматривать ситуации, когда стоимость операции зависит от состояния структуры данных. Например, добавление в конец `ArrayList` обычно можно выполнить за постоянное время, если только базовый массив не заполнился. В таком случае нужно выделить память для нового массива большего размера и переместить в него содержимое старого массива. Стоимость этой операции линейно зависит от числа элементов в массиве, но встречается такая ситуация сравнительно редко. В подобных случаях мы вычисляем *амортизированную стоимость* операции, т. е. предел полной стоимости ее *N*-кратного выполнения, поделенной на *N*, когда *N* стремится к бесконечности. В случае добавления элемента в `ArrayList` полная стоимость для *N* элементов равна $O(N)$, поэтому амортизированная стоимость равна $O(1)$.

Анализ времени выполнения в терминах *O* большого часто бывает весьма полезен для оценочного сравнения времени выполнения программ, и по этой причине во все главы этой книги, посвященные интерфейсам, включен такой анализ типичных операций для каждой реализации. Но все же есть причины с осторожностью относиться к асимптотическим оценкам в качестве замены времени выполнения, а именно:

- предположение о преобладающем члене подразумевает, что размер данных всегда будет достаточно велик, чтобы перевесить постоянные множители и слагаемые; например, считается, что $O(N + c)$ можно заменить на $O(N)$, если *c* постоянно, поскольку во всех интересных нам случаях $N >> c$ (т. е. *N* гораздо больше *c*). Это может быть не так для очень больших слагаемых;
- время выполнения машинных команд может различаться. Например, сложность вставки элемента в первую позицию `ArrayList` равна $O(N)$, потому что каждый элемент массива нужно переместить в следующую позицию. Но если эту операцию можно реализовать аппаратно путем операции копирования одного блока данных в памяти, то факти-

ческое время выполнения может быть вполне сравнимо со временем выполнения аналогичной операции в `LinkedList`, хотя сложность последней составляет $O(1)$;

- параллелизм еще усложняет картину. Благодаря параллелизму на уровне команд команды исполняются поэтапно, причем различные этапы соседних команд исполняются одновременно. Поэтому полное время выполнения программы обычно гораздо меньше суммы времен выполнения отдельных ее команд. То же самое, но в более крупном масштабе, относится к параллелизму на уровне потоков, когда несколько потоков могут выполнять свои задачи одновременно за исключением тех участков, где они конкурируют за ресурсы;
- временная сложность одной операции сама по себе не важна; важно время, необходимое для выполнения целого сценария. Рассмотрим, к примеру, обход всей структуры данных с заменой всех ее элементов (если такой сценарий возможен в вашем приложении) вместо простого продвижения итератора на один элемент. Для расчета времени, потребного для такого сценария, нужно принимать во внимание такие факторы, как стоимость непопадания в кеш и амортизированная стоимость сборки мусора, которые зависят от характеристик структуры данных, не улавливаемых временной сложностью отдельных операций. Этот момент подробнее иллюстрируется в табл. 14.2.

В заключение еще раз подчеркнем ценность теоретического анализа эффективности, в том числе с помощью нотации *O* большое: он может подсказать правильное направление на этапе проектирования. Например, вы никогда не отдадите предпочтение алгоритму сложности $O(N^2)$ перед алгоритмом сложности $O(N)$ для большого набора данных. Но когда проблема производительности возникает в работающей системе, нет никакой замены точным измерениям для сравнения различных потенциальных реализаций.

Неизменяемость и немодифицируемость

Функциональное программирование имеет свои привлекательные черты; в своих лучших образцах функциональные программы элегантны и доказуемо корректны – куда в большей степени, чем объектно ориентированные программы. Поэтому некоторые черты функциональных языков, обеспечивающие эти преимущества, постепенно отыскали дорогу в Java – началось параметризованными типами, затем продолжилось потоками и лямбда-выражениями (см. раздел «Лямбда-выражения и потоки» ниже) и достигло кульминации сопоставлением с образцами в `switch`, оператором `instanceof` и записями, которые на момент написания книги все еще являются часть проекта Amber. Важная особенность функционального стиля – тот факт, что структура данных в нем *неизменяемы*, т. е. их состояние нельзя изменить после создания. Неизменяемость несет с собой ряд преимуществ:

- неизменяемые объекты потокобезопасны;
- неизменяемые объекты идеально инкапсулированы, что устраняет необходимость *защитного копирования* (прием, благодаря которому объекты гарантируют целостность свих данных, раскрывая только их копии; см. раздел «Не забывайте о “владельцах” коллекций» главы 17);

- неизменяемость гарантирует стабильность поиска в коллекциях с ключами и в упорядоченных коллекциях (см. раздел «Контракты» ниже);
- неизменяемость уменьшает число возможных состояний программы, что делает ее проще, прозрачнее и податливее для понимания и рассуждения.

Но реализовать эти преимущества в Java-программе трудно. Чтобы объект был по-настоящему неизменяемым, весь его объектный граф, т. е. сам объект и все, на что он ссылается (прямо или косвенно), не должны претерпевать наблюдаемых изменений после конструирования. Очевидно, что неизменяемые компоненты графа, например объекты-обертки и строки, проблемы не составляют, но для изменяемых объектов трудновыполнимым требованием зачастую является монопольный доступ к ним, который должен был бы гарантировать граф.

Это породило альтернативные и конфликтующие терминологии для описания неизменяемости коллекций.

Такие каркасы, как Guava и Eclipse Collections, называют неизменяемость всего объектного графа *глубокой неизменяемостью*. Коллекцию, которая запрещает модификацию на первом уровне, т. е. попытки добавить, удалить или заменить элемент, они называют *поверхностно неизменяемой*, или просто *неизменяемой*.

В документации по каркасу коллекций Java термин *неизменяемость* означает неизменяемость всего объектного графа. То, что в других каркасах называется «поверхностной неизменяемостью», в документации Java – и в этой книге – называется *немодифицируемостью*.

Немодифицируемость – это вид частичной неизменяемости, который не обладает всеми рассмотренными выше преимуществами неизменяемости, поэтому естественно возникает вопрос о ее полезности. Но кое-какие реальные преимущества у нее все же есть.

- Коллекции неизменяемых объектов, в том числе оберток и строк, встречаются не так уж редко, а немодифицируемые коллекции таких объектов обладают всеми преимуществами неизменяемости.
- Даже частичная неизменяемость уменьшает число состояний программы, которые нужно рассматривать при рассуждениях о ее корректности.
- Поскольку в основе немодифицируемых множеств и отображений могут лежать массивы, а не хешированные структуры, возможна весьма значительная экономия памяти.

Дополнительное преимущество немодифицируемости заключается в том, что в принципе (и в других каркасах, но не в каркасе коллекций Java это сделано) ее можно отразить в системе типов. С другой стороны, чтобы объектный граф был неизменяемым, он должен иметь монопольный доступ ко всем изменяемым компонентам. В общем случае это свойство невозможно поддержать на этапе компиляции и даже с помощью статического анализа, не говоря уже об ограничениях системы типов. Согласны, к объектному графу, целиком состоящему из неизменяемых объектов, это возражение неприменимо, но в общем случае за коллекциями стоят массивы, которые в Java всегда изменяемы.

В разделе «Фундаментальные проблемы дизайна каркаса коллекций» главы 18 мы продолжим обсуждение этой темы, а в главе 17 рассмотрим практические ситуации, в которых следует отдать предпочтение немодифицируемым коллекциям.

Контракты

Читая о проектировании программного обеспечения, вы, вероятно, встречали термин *контракт* зачастую без всяких объяснений. На самом деле в программной инженерии этому термину придается значение, очень близкое к тому, что люди понимают под контрактом. В повседневной жизни контракт определяет, что две стороны вправе ожидать друг от друга: их взаимные обязательства по исполнению сделки. Очевидно, контракт должен описывать обязательства поставщика услуг перед клиентом. Но и у клиента тоже могут быть обязательства – в том числе (но не только) обязанность заплатить, – и их невыполнение автоматически освобождает поставщика от выполнения своих обязательств. Например, в условиях перевозки, принятых на себя авиакомпанией, – по крайней мере, для того класса авиабилетов, который могут позволить себе авторы книги, – компания освобождается от обязанности перевозить пассажиров, опоздавших на рейс. Это позволяет авиакомпании планировать обслуживание в предположении, что все пассажиры будут пунктуальны; она не обязана прилагать дополнительные усилия для обслуживания клиентов, не выполнивших свою часть контракта.

Точно так же контракты работают и в программах. Если в контракте метода оговорены предусловия на его аргументы (т. е. обязательства клиента), то метод должен вернуть описанные в контракте результаты, только если эти предусловия выполнены. Например, двоичный поиск (см. разделе «Нахождение конкретных значений в списке» главы 16) – быстрый алгоритм поиска элемента, но только в упорядоченном списке, и он вправе вернуть неверный результат, если применяется к неупорядоченному списку. Таким образом, в контракте метода `Collections::binarySearch` может быть оговорено «если список не отсортирован, то результат не определен», так что автор двоичного поиска вправе написать код, который для неупорядоченного списка вернет абсолютно любой результат. Если возможно, предусловия проверяются, и в случае их нарушения возбуждается исключение; это разумный подход для библиотек общего назначения, которые будут использоваться в самых разных ситуациях программистами самой разной квалификации. В API менее общих библиотек его обычно избегают, потому что он без необходимости ограничивает гибкость библиотеки. Да и не всегда это возможно: временная сложность проверки предусловия для двоичного поиска равна $O(N)$ – больше, чем у самого поиска. В таких случаях готовность к получению неверного результата – наилучшая альтернатива, не нарушающая условий контракта.

Короче говоря, клиенту нужно знать только, как обеспечить выполнение своей стороны контракта; если он этого не сделает, то библиотека освобождается от всех гарантий правильности.

В Java считается хорошей практикой кодировать в соответствии с интерфейсом, а не с конкретной реализацией, поскольку это обеспечивает максимальную гибкость при выборе реализации. Эта идея подробно исследуется в разделе «Соблюдайте баланс интересов клиента и библиотеки при проектировании API» главы 17. Что для этого требуется от поведения реализаций? Например, если ваш клиент пользуется методами интерфейса `List`, а на этапе выполнения работает `ArrayList`, то вам желательно знать, что предположения, которые вы

сделали о поведении списков `List`, действительны и для `ArrayList`. Таким образом, класс, реализующий интерфейс, обычно должен выполнить все обязательства, оговоренные в условиях контракта интерфейса. Более слабая форма этих обязательств уже проверена компилятором: класс, объявляющий о том, что реализует интерфейс, обязан предоставить объявления конкретных методов, совпадающие с объявлениями в интерфейсе. Но контракты идут дальше – определяют также и поведение этих методов.

Однако обязательства, налагаемые интерфейсами каркаса коллекций, в некоторых отношениях необычны. Руководящим принципом при проектировании каркаса была простота – важная характеристика библиотеки, которую должен зарубить себе на носу любой программист на Java. В каркасах, которые отделяют модифицируемые интерфейсы от немодифицируемых, типов гораздо больше, чем в каркасе коллекций Java, поэтому проектировщики каркаса решили отказаться от такого разделения. Но поскольку многие представления коллекций (а начиная с Java 9, также и немодифицируемые коллекции) поддерживают не все операции записи, в контрактах интерфейсов эти операции помечены как *факультативные* (optional). Например, представление `Set` ключей отображения `Map` допускает удаление ключей, но не их добавление (см. главу 15), тогда как в других представлениях коллекций может быть запрещено как добавление, так и удаление элементов (например, в списковом представлении, возвращенном методом `Arrays::asList`), или вообще все операции модификации, как в коллекциях, обернутых немодифицируемой оберткой (см. раздел «Немодифицируемые коллекции» главы 16). Контракты интерфейсов из каркаса коллекций позволяют реализациям возбуждать исключение `UnsupportedOperationException` при вызове факультативных методов. Мы подробно обсудим это проектное решение в разделе «Фундаментальные проблемы дизайна каркаса коллекций» главы 18.

До сих пор мы обсуждали только функциональные требования, но в контрактах могут быть также оговорены гарантии производительности. Однако, чтобы в полной мере понимать характеристики производительности класса, вы часто должны знать детали алгоритмов реализации. В следующих главах при обсуждении различных имеющихся в JDK реализаций интерфейсов каркаса мы также будем сообщать некоторые подробности об их алгоритмах там, где это может оказаться полезным. Это может помочь при выборе реализации, но помните, что эта информация не вырублена в камне; контракты обязывают, но одно из их основных преимуществ – то, что они позволяют изменять реализации в случае появления новых алгоритмов или когда усовершенствование оборудования изменяет сравнительные достоинства прежних алгоритмов[1]. И конечно, если вы используете другую реализацию, то детали алгоритма, не оговоренные контрактом, могут быть совершенно иными.

[1] Удивительный пример такого рода имел место, когда эту книгу уже готовились передать в печать. Появилось сообщение, что студент факультета информатики придумал способ радикально ускорить некоторые операции с хеш-таблицей, перевернув тем самым царившие десятилетиями представления (Farach-Colton et al. 2025).

Организация по содержимому

В разделе «Упорядоченные коллекции» главы 8 была выдвинута идея внутренне упорядоченных коллекций, элементы которых автоматически ставятся на место, определяемое их содержимым, на котором определено отношение порядка (см. главу 3). Это не единственные классы коллекций, автоматически организующие свои элементы в соответствии с содержимым; хешированные коллекции, очереди с приоритетами и очереди с задержкой обладают тем же свойством. В случаях внутренне упорядоченных последовательностей и очередей организацию по содержимому можно наблюдать извне благодаря операциям коллекции; в случае хешированных коллекций это не так, но такая организация используется при поиске элементов. У всех этих коллекций есть общая черта – неявный инвариант: для правильной работы значения полей, по которым элементы позиционируются, не должны изменяться после вставки в коллекцию. Например, рассмотрим другую версию класса `Person`, в которой на этот раз метод `hashCode` определен, но зависит от изменяемого поля `name`:

org/jgcbook/chapter09/F_contracts/Person

```
class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int hashCode() {
    return name.hashCode();
  }
  public boolean equals(Object o) {
    return o instanceof Person p && name.equals(p.name);
  }
}
```

Теперь объект типа `Person` можно надежно извлечь, но только если значение `name` не изменилось. Если же это поле было модифицировано, то коллекция не сможет найти объект ни по старому, ни по новому значению.

org/jgcbook/chapter09/F_contracts/Person

```
Set<Person> people = new HashSet<>();
Person alice = new Person("Alice");
people.add(alice);
alice.setName("Bob");
assert ! people.contains(new Person("Alice"));
assert ! people.contains(new Person("Bob"));
```

Чтобы избежать подобных проблем, придерживайтесь следующей рекомендации: если вы собираетесь хранить объекты в `Set`, `Map` или внутренне упорядоченной очереди `Queue`, то делайте поля, используемые для организации элементов коллекции, неизменяемыми.

Лямбда-выражения и потоки

В разделе «Неизменяемость и немодифицируемость» выше мы видели, что преимущества функционального программирования подвигли объектно ориентированные языки к внедрению многих возможностей функциональных языков. После включения параметризованных типов в Java 5, следующим крупным шагом стала версия Java 8, в которой появились лямбда-выражения и потоки. Лямбда-выражения открыли новый способ представления функций в Java-программах. Например, предложение

```
Function<Integer,String> intToString = i -> i.toString();
```

объявляет функцию `intToString`, которая принимает аргумент типа `Integer` и возвращает его строковое представление.

Важность лямбда-выражений для обработки коллекций проистекает из их использования в потоковых операциях. Потоки – это механизм транспортировки последовательности значений из источника в приемник с попутным выполнением ряда операций, которые обычно реализованы в виде лямбда-выражений, отвечающих за преобразование, удаление или вставку новых значений. Эта модель предлагает альтернативу традиционному использованию коллекций для агрегирования данных. Простой, хотя и несколько искусственный пример (из книги «Mastering Lambdas» [Naftalin 2014]) иллюстрирует такую смену мышления. Запись `Point` представляет точку, определенную своими координатами `x` и `y`, и имеет единственный метод `distanceFrom`, который вычисляет расстояние до другой точки.

org/jgcbook/chapter09/G_lambdas_and_streams/Point

```
record Point(int x, int y) {
  public double distanceFrom(Point p) { ... }
}
```

Не будь потоков, нам пришлось бы обрабатывать коллекции в несколько этапов: коллекция итеративно обрабатывается, при этом порождается новая коллекция, которая тоже итеративно обрабатывается и т. д. Показанный ниже код начинается с создания коллекции экземпляров типа `Integer`, применяет к ним какое-то преобразование для порождения множества экземпляров типа `Point` и, наконец, находит максимум расстояний от этих точек до начала координат.

org/jgcbook/chapter09/G_lambdas_and_streams/Program_1

```
Point origin = new Point(0, 0);
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
List<Point> pointList = new ArrayList<>();
for (Integer i : intList) {
  pointList.add(new Point(i % 3, i / 3));
}
double maxDistance = 0;
for (Point p : pointList) {
  maxDistance = Math.max(p.distanceFrom(origin), maxDistance);
}
```

Реальный код такого типа имеет несколько недостатков: он очень многословный; промежуточная коллекция `pointList` – лишние затраты, которые создают повышенную нагрузку на сборщик мусора и могут даже привести к ис-

черпанию места в куче; присутствует трудно выявляемое неявное предположение о том, что максимальное значение для пустого списка равно 0; и, что, пожалуй, хуже всего, назначение программы трудно понять, потому что существенные операции перемежаются с кодом обработки коллекций.

Ниже приведен эквивалентный код, представленный в виде конвейерной обработки оригинальной коллекции `intList` с помощью ряда преобразований:

org/jgcbook/chapter09/G_lambdas_and_streams/Program_2

```
Point origin = new Point(0, 0);
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
OptionalDouble maxDistance = intList.stream()
    .map(i -> new Point(i % 3, i / 3))
    .mapToDouble(p -> p.distanceFrom(origin))
    .max();
```

Этот пример демонстрирует в миниатюре преимущества потокового кода: он короче и лучше воспринимается, часто потребляет меньше промежуточной памяти, элегантно обрабатывает пустой источник и никогда не пытается изменить исходную коллекцию.

Начиная с этого места, мы всюду, где код работы с коллекциями может выиграть от частичного перехода к использованию потоков, будем показывать альтернативный потоковый код во врезке, надеясь, что вы сможете извлечь пользу из сравнения обеих идиом.

Параллельные потоки

Основной целью проектирования потокового API было создание возможности для распределения рабочих нагрузок между несколькими процессорными ядрами, не требующей значительных усилий со стороны программиста приложения. Это достигается применением *рекурсивной декомпозиции*, когда набор данных разбивается на части для раздельной обработки, а по ее завершении результаты объединяются.

Например, если имеется четырехъядерный процессор и список из *N* элементов на основе массива, то программа могла бы определить алгоритм `solve`, который распараллеливает задачу следующим образом (псевдокод сильно упрощен):

```
if список задач содержит больше N/4 элементов {
  leftTask = task.getLeftHalf()
  rightTask = task.getRightHalf()
  doInParallel {
    leftResult = leftTask.solve()
    rightResult = rightTask.solve()
  }
} else {
  result = combine(leftResult, rightResult)
  result = task.solveSequentially()
}
```

Для реализации рекурсивной декомпозиции нужно знать, как разбивать на задачи подобным образом и как в конечном итоге выполнить достаточно мелкие задачи, не требующие дальнейшего разбиения. Стратегия этих операций зависит от источника данных. Соответственно с любой реализацией `Iterable` –

иными словами, подтипом `Collection` – связан объект `Spliterator`, который содержит стратегии, подходящие для этой коллекции.

В данном случае для разбиения основанного на массиве списка имеется очевидная реализация: разделить массив пополам, разделить каждую из получившихся частей пополам и т. д., пока очередное деление не даст больше частей, чем имеется ядер для их обработки, или пока накладные расходы разбиения не перевесят выигрыш от распараллеливания обработки.

Каркас потоков скрывает сложность программирования рекурсивной декомпозиции за удобной абстракцией *параллельных потоков*. Если поток создан параллельным или сделан параллельным посредством вызова метода `parallel`, то каркас применяет рекурсивную декомпозицию к набору данных, обрабатываемому поточно, и назначает каждой из получившихся частей свой поток выполнения (thread). Ниже единственное изменение, которое нужно внести в наш пример, выделено полужирным шрифтом:

```
OptionalDouble maxDistance =
    intList.stream()
        .parallelStream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p -> p.distance(0, 0))
        .max();
```

В идеале каждому потоку назначается ядро, на котором он сможет выполняться без прерываний. В лучшем случае – когда имеется большой набор данных, требующий счетной обработки с очень небольшим объемом ввода-вывода, – это приводит к почти идеальному распараллеливанию.

Дополнительные сведения о параллельных потоках см. в книге Naftalin (2014).

Коллекции и потокобезопасность

Работающая Java-программа исполняет один или несколько *потоков выполнения* (thread). Поток можно уподобить облегченному процессу, поэтому программу, одновременно исполняющую несколько потоков, можно рассматривать как компьютер, одновременно исполняющий несколько программ, но с одним важным различием: различные потоки могут одновременно обращаться к одним и тем же ячейкам памяти и другим системным ресурсам. На машинах с несколькими процессорами истинно конкурентного исполнения потоков можно достичь путем назначения отдельного процессора каждому потоку. Но если потоков больше, чем процессоров, – а это обычный случай, – то многопоточность реализуется путем *квантования времени*, когда процессор исполняет несколько команд одного потока, а потом переключается на следующий.

Для использования многопоточных программ есть две основательных причины. Очевидная – применимая к многоядерным и многопроцессорным компьютерам – разделить работу и выполнить ее быстрее. (Эта причина становится тем более веской, чем пристальнее проектировщики оборудования рассматривают параллелизм как способ повышения общей производительности.) Вторая причина заключается в том, что две операции могут занимать разное, не всегда известное заранее время, а мы не хотим, чтобы для реакции на одну операцию нужно было дожидаться завершения другой. Особенно это

относится к графическому интерфейсу пользователя (GUI), когда реакция на нажатие пользователем кнопки должна быть немедленной, а не задерживаться на время, необходимое программе для завершения какой-нибудь счетной части приложения.

Хотя конкурентность может быть важна для достижения хорошей производительности, за нее приходится платить. Разные потоки, одновременно обращающиеся к одной и той же ячейке памяти, могут приводить к неожиданным результатам, если мы не примем меры для ограничения доступа. Рассмотрим пример 9.1, где в классе `ArrayStack` используется массив и индекс, чтобы реализовать интерфейс `Stack`, моделирующий стек чисел типа `int`. Чтобы `ArrayStack` работал правильно, переменная `index` всегда должна указывать на позицию после верхнего элемента стека независимо от того, сколько элементов помещено в стек или выбрано из него. Это *инвариант* класса. Теперь подумайте, что может произойти, если два потока одновременно попытаются поместить элемент в стек. Войдя в метод `push`, оба будут выполнять выделенные полужирным шрифтом строчки. В однопоточной среде это нормально, но в многопоточной может нарушить инвариант.

Пример 9.1. *Потоконебезопасная реализация стека*

org/jgcbook/chapter09/H_collections_and_thread_safety/ArrayStack

```
interface Stack {
  public void push(int elt);
  public int pop();
  public boolean isEmpty();
}

public class ArrayStack implements Stack {
  private final int MAX_ELEMENTS = 10;
  private int[] stack;
  private int index;
  public ArrayStack() {
    stack = new int[MAX_ELEMENTS];
    index = 0;
  }
  public void push(int elt) {
    if (index != stack.length) {
      stack[index] = elt;
      index++;
    } else {
      throw new IllegalStateException("stack overflow");
    }
  }
  public int pop() {
    if (index != 0) {
      return stack[--index];
    } else {
      throw new IllegalStateException("stack underflow");
    }
  }
  public boolean isEmpty() { return index == 0; }
}
```

Один из многих возможных сценариев неправильного выполнения этого кода показан на рис. 9.2: поток A выполняет первую выделенную строчку, поток B выполняет первую выделенную строчку, а затем вторую, и, наконец, поток A выполняет вторую выделенную строчку. Теперь в стеке оказалось только значение, помещенное в него потоком B, потому что оно затерло значение, помещенное потоком A. С другой стороны, переменная `index` могла быть увеличена на два[1], и в таком случае она больше не указывает на позицию после верхнего элемента стека, как должна. Про такую программу говорят, что в ней имеет место *состояние гонки*, потому что результат зависит от относительных скоростей выполнения нескольких потоков. Если, как в этом случае, исполнение нарушает инвариант, то программа остается в противоречивом состоянии и, скорее всего, аварийно завершится, потому что другие части программы полагаются на то, что инвариант истинный.

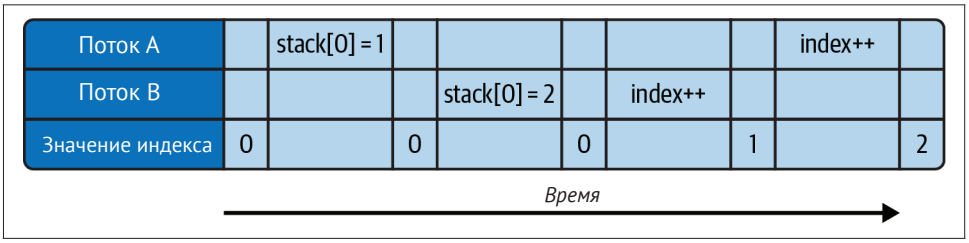


Рис. 9.2. Состояние гонки

Поскольку за время существования Java роль конкурентного программирования сильно возросла, в каркасе коллекций пришлось уделить повышенное внимание гибким и эффективным политикам конкурентности. Будучи пользователем коллекций на платформе Java, вы должны хотя бы в общих чертах понимать политики конкурентности разных коллекций, чтобы знать, какую выбрать и как ей правильно пользоваться. В этом разделе мы кратко опишем различные способы конкурентной работы с коллекциями и вытекающие из них следствия, интересные программисту. Полное изложение общей теории конкурентного программирования см. в книге «Concurrent Programming in Java» (Lea 1999), а детали, касающиеся конкурентности в Java и в реализациях коллекций, – в книге «Java Concurrency in Practice» (Goetz et al. 2006).

Синхронизация и унаследованные коллекции

Код наподобие показанного выше в классе `ArrayStack` не является потокобезопасным – он работает, когда исполняется одним потоком, но может перестать работать в многопоточной среде. Поскольку наблюдавшееся нами некорректное поведение имело место, когда два потока одновременно исполняли метод `push`, мы могли бы изменить программу, сделав это невозможным. Использование ключевого слова `synchronized` в объявлении метода `push` гарантирует, что, как только какой-то поток начнет выполнять его, всем остальным потокам будет отказано в выполнении этого метода до тех пор, пока первый поток не выйдет из него:

[1] Влияние этого несинхронизированного кода на `index` тоже недетерминированно.

```
public synchronized void push(int elt) { ... }
```

Это называется *синхронизацией* доступа к *критической секции* кода – в данном случае всему методу `push`. Прежде чем начать исполнение синхронизированного кода, поток должен захватить блокировку на некоторый объект, называемый *монитором*, – для методов монитор по умолчанию совпадает с текущим объектом. Пока один поток удерживает блокировку, другой поток, пытающийся войти в критическую секцию, синхронизированную по этой блокировке, *блокируется*, т. е. приостанавливается, до тех пор, пока не сможет получить блокировку. Синхронизированная версия `push` потокобезопасна; в многопоточной среде каждый поток ведет себя так же, как в однопоточной.

Чтобы защитить инвариант и сделать весь класс `ArrayStack` потокобезопасным, методы `pop` и `isEmpty` также должны быть синхронизированы по одному и тому же объекту. Метод `isEmpty` не записывает разделяемые данные, поэтому синхронизация нужна не для предотвращения состояния гонки, а по другой причине. Каждый поток может пользоваться отдельным кешем памяти, а это означает, что результаты записи из одного потока могут быть видны другому не сразу. Такое поведение допускается моделью памяти Java Memory Model (Java Memory Model – JMM) ради максимизации степени параллелизма. JMM, являющаяся частью спецификации языка Java (Gosling et al. 2023, § 17.4), описывает, как гарантировать согласованность данных между потоками: либо с помощью ключевого слова `volatile`, либо позаботившись о том, чтобы чтение производилось только после того, как и читатель, и писатель синхронизированы по одному и тому же монитору.

На самом деле полная синхронизация методов была политикой классов коллекций со времен JDK 1.0: для `Vector`, `Hashtable` и их подклассов все методы, обращающиеся к данным экземпляра, синхронизированы. Сейчас эти классы считаются унаследованными, и пользоваться ими не рекомендуется из-за высокой цены, которую должны платить все клиенты этих классов независимо от того, нужна им потокобезопасность или нет[1]. Синхронизация может обходиться очень дорого: принудительная постановка потоков в очередь для входа в критическую секцию, замедляет выполнение программы в целом, а накладные расходы на управление блокировками в случае частой конкуренции за них могут оказаться очень высокими. Модель конкурентности в JDK 1.0 страдает от той же проблемы, что и синхронизированные обертки, – мы кратко обсудим ее в следующем разделе, а более развернуто в разделе «Избегайте синхронизированных оберток коллекций» главы 17, – они требуют синхронизации нескольких вызовов методов вызывающей стороной для достижения согласованного поведения.

Синхронизированные коллекции и итераторы с быстрым отказом

Накладные расходы на внутреннюю синхронизацию коллекций из JDK 1.0 подвели проектировщиков к мысли избегать ее, когда в версию JDK 1.2 впервые

[1] Исключением является класс `java.util.Properties`, который, хоть и является подклассом унаследованного класса `Hashtable`, используется настолько широко, что был модифицирован и теперь внутри себя пользуется классом `ConcurrentHashMap`.

был включен каркас коллекций. Вместо этого в платформенных реализациях интерфейсов `List`, `Set` и `Map` был расширен выбор политик конкурентности, доступных программисту. Чтобы добиться максимальной производительности при однопоточном выполнении, новые коллекции позволяют вообще отказаться от управления конкурентностью. (Аналогичные изменения были внесены в другие синхронизированные классы – например, синхронизированный класс `StringBuffer` был дополнен в Java 5 его несинхронизированным эквивалентом, `StringBuilder`.)

Вместе с этим изменением пришла и новая политика конкурентности для итераторов коллекций. В многопоточной среде поток, получивший итератор, обычно продолжает использовать его, тогда как другие потоки модифицируют исходную коллекцию. Таким образом, поведение итератора должно рассматриваться как неотъемлемая часть политики конкурентности коллекции. Для коллекций Java 2 итераторы придерживаются политики *быстрого отказа*, описанной в разделе «Итерируемые объекты и итераторы» выше: при каждом доступе к своей коллекции они проверяют, не произошло ли в ней структурных изменений (в общем случае это означает, что элементы были добавлены или удалены), сравнивая счетчики модификаций коллекций с хранящимся внутри итератора значением. Несовпадение означает, что коллекция была модифицирована, но не с помощью методов итератора; в таком случае итератор немедленно завершает работу и возбуждает исключение `ConcurrentModificationException`, вместо того чтобы пытаться продолжать обход модифицированной коллекции с непредсказуемыми результатами. Заметим, что такое поведение помогает находить и диагностировать ошибки, но контрактом коллекции оно не гарантируется.

Появление в Java коллекций без принудительной синхронизации было желанным изменением. Однако во многих ситуациях потокобезопасные коллекции все же необходимы, поэтому каркас предоставляет возможность использовать новые коллекции со старой политикой конкурентности с помощью синхронизированных оберток (см. главу 16). Они создаются посредством вызова одного из фабричных методов в классе `Collections`, которому передается несинхронизированная коллекция, подлежащая инкапсуляции. Мы рассмотрим их подробнее в разделе «Избегайте синхронизированных оберток коллекций» главы 17, но краткое резюме этого раздела будет полезно прямо сейчас. Там мы приходим к выводу, что полезность оберток сильно ограничена: поскольку предлагаемая ими потокобезопасность распространяется только на индивидуальные операции, клиентам приходится реализовывать собственную политику блокировки, синхронизируя несколько вызовов методов, чтобы добиться согласованного поведения. Это делает потокобезопасные коллекции менее полезными и снижает их производительность, поскольку клиентские блокировки необходимо удерживать на протяжении нескольких вызовов методов.

Конкурентные коллекции

В Java 5 были добавлены потокобезопасные конкурентные коллекции как часть гораздо большего набора средств обеспечения конкурентности, включающего примитивы – атомарные переменные и блокировки, – которые предоставляют Java-программисту доступ к относительно недавним возможностям оборудования для управления конкурентными потоками (прежде всего операциям срав-

нения и установки, которые объясняются в следующем разделе). Атомарные переменные получили название от предоставляемых ими *атомарных операций*, которые с точки зрения других потоков выполняются за один шаг. Например, метод `incrementAndGet` класса `AtomicInteger` инкрементирует значение объекта и возвращает новое значение, исключая всякую возможность вмешательства со стороны других потоков (см. раздел «Коллекции и потокобезопасность» выше).

Конкурентные коллекции устраняют необходимость реализовывать блокировку на стороне клиента, описанную в предыдущем разделе. На самом деле для таких коллекций внешняя синхронизация вообще невозможна, потому что не существует одного такого объекта, который после захвата блокировки блокировал бы все методы. Если нужна атомарная операция – например, вставка элемента в `Map`, только если его там еще нет, – то конкурентная коллекция предоставляет метод для ее атомарного выполнения, в данном случае `ConcurrentMap.putIfAbsent`.

Если вам нужна потокобезопасность, то конкурентные коллекции обеспечат гораздо лучшую производительность, чем синхронизированные. В основном это связано с тем, что их пропускная способность не снижается из-за необходимости сериализовывать доступ, как в случае синхронизированных коллекций. Кроме того, синхронизированные коллекции вынуждены нести затраты на управление блокировками, а при высокой конкуренции за ресурсы они могут быть велики. В итоге эффективность конкурентного доступа со стороны большого числа потоков может вырасти на два порядка.

Механизмы работы конкурентных коллекций

Конкурентные коллекции обеспечивают потокобезопасность благодаря нескольким механизмам. Первый из них – единственный, в котором не используются новые примитивы, – *копирование при записи*. Классы, в которых применяется копирование при записи, хранят свои значения во внутреннем массиве, фактически неизменяемом; любое изменение значения коллекции приводит к созданию нового массива для представления новых значений. В этих классах синхронизация используется, но только на очень краткие промежутки времени, когда создается новый массив. Поскольку операции чтения не нуждаются в синхронизации, коллекции, копируемые при записи, обеспечивают хорошую производительность в ситуациях, для которых спроектированы, – тех, где операций чтения значительно больше, чем операций записи. Этот механизм используется в классах коллекций `CopyOnWriteArrayList` и `CopyOnWriteArraySet`.

Вторая группа потокобезопасных коллекций опирается на аппаратные операции, носящие общее название «сравнить и установить» (compare-and-set – CAS), которые фундаментально превосходят традиционную синхронизацию. Чтобы понять, как работает CAS, рассмотрим ситуацию, когда значение одной переменной подается на вход длительного вычисления, конечный результат которого используется для обновления переменной. При традиционной синхронизации все вычисление было бы атомарным, т. е. конкурентный доступ к переменной со стороны других потоков был бы запрещен. Это уменьшает возможности параллельного выполнения, а значит, и пропускную способность. Алгоритм на основе CAS ведет себя иначе: поток создает локальную копию переменной и выполняет вычисление, не запрашивая монопольный доступ. И лишь когда он будет готов обновить переменную, вызывается команда CAS, которая за одну

атомарную операцию сравнивает текущее значение переменной с начальным и, если они совпадают, записывает в нее новое значение. Если же значения не совпадают, значит переменная была модифицирована другим потоком; в этой ситуации CAS-поток может попытаться повторить вычисление с новым значением, сдаться или – в некоторых алгоритмах – продолжить работу, потому что вмешавшийся поток проделал работу за него! Такой стиль блокировки называется *оптимистическим* управлением конкурентностью, в отличие от традиционного пессимистического стиля, когда всем прочим потокам запрещается доступ к переменной, пока данный поток не закончит работу. CAS используется, в частности, в коллекциях `ConcurrentLinkedQueue` и `ConcurrentSkipListMap`.

В третьей группе используются реализации интерфейса `java.util.concurrent.locks.Lock`, появившегося в Java 5 в качестве более гибкой альтернативы классической синхронизации. Объект `Lock` демонстрирует то же базовое поведение, что и классическая синхронизация, но поток может также захватить его при специальных условиях: только если блокировка в данный момент никем не захвачена, или если она освободилась в течение установленного периода тайм-аута, или если поток не был прерван. В отличие от синхронизированного кода, в котором блокировка объекта удерживается в течение всего времени выполнения блока кода или метода, `Lock` удерживается до вызова метода `unlock` (что позволяет освободить блокировку `Lock` в блоке или методе, отличном от того, в котором она была захвачена). Некоторые классы коллекций, входящих в эту группу, применяют этот механизм, чтобы разделить коллекцию на части, которые можно заблокировать по отдельности, что повышает степень конкурентности. Например, в классе `LinkedBlockingQueue` есть отдельные блокировки для начала и конца очереди, чтобы элементы можно было добавлять и удалять параллельно. К числу других коллекций, пользующихся такими блокировками, входит большинство реализаций интерфейса `BlockingQueue`.

Итераторы

Описанные выше механизмы ведут к политикам итераторов, более пригодным для конкурентного использования, чем быстрый отказ; в них конкурентная модификация неявно рассматривается как проблема, которую нужно устранить. Коллекции, копируемые при записи, обладают *слепковыми итераторами* (snapshot iterator). За этими коллекциями стоят массивы, которые после создания никогда не изменяются; если значение в коллекции нужно изменить, то создается новый массив. Это означает, что итератор может читать значения в одном из этих массивов (но не модифицировать их), не опасаясь, что они будут изменены другим потоком. Слепковые итераторы не возбуждают исключение `ConcurrentModificationException`.

В коллекциях, опирающихся на команду CAS, имеются *слабо согласованные* итераторы, отражающие некоторые, но, возможно, не все изменения, внесенные в обслуживаемые ими коллекции с момента их создания. Например, если элементы коллекции были модифицированы или удалены до того, как итератор дошел до них, то он обязательно отразит такие изменения, но для вставок такой гарантии не дается. В третьей группе коллекций, описанной в предыдущем разделе, также имеются слабо согласованные итераторы. Они не возбуждают исключение `ConcurrentModificationException`.

Заключение

Эта глава является подготовительной для углубленного изучения каркаса коллекций. Чтобы извлечь максимум пользы из этого каркаса, очень полезно понимать идеи, заложенные при его проектировании, а также изменения в оборудовании, которые продолжают влиять на его развитие.

В главе 10 мы будем рассматривать самый важный интерфейс в каркасе коллекций: `Collection`, являющийся супертипом всех коллекций, кроме отображений.

Глава 10

Интерфейс Collection

Интерфейс `Collection<E>` является корнем иерархии типов в каркасе коллекций. Он раскрывает базовую функциональность, ожидаемую от всех коллекций, кроме отображений.

Методы Collection

Методы, объявленные в интерфейсе `Collection`, можно разбить на четыре группы: добавление элементов, удаление, опрос и обеспечение доступности элементов для дальнейшей обработки.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter10*.

Добавление элементов

`boolean add(E e)`	добавить элемент `e`
`boolean addAll(Collection<? extends E> c)`	добавить содержимое `c`

Контракты этих методов требуют, чтобы после выполнения коллекция содержала элемент или элементы, переданные в аргументе. Возвращаемое значение показывает, изменил ли метод содержимое коллекции. Например, попытка добавить элемент в множество, которое уже содержит этот элемент, завершается успешно, но возвращает `false`. Если коллекция не содержала элемента или элементов и метод не смог добавить их, то он должен возбудить исключение. Это может случиться, потому что коллекция не поддерживает операцию или потому что метод пытался добавить недопустимый элемент, например, добавить `null` в коллекцию, которая не может содержать элементов, равных `null`.

Чтобы сохранить однородность коллекции, эти методы позволяют добавлять только элементы или коллекции элементов того же типа, что параметр коллекции. Для методов следующей группы это не так.

Удаление элементов

`void clear()`	удалить все элементы
`boolean remove(Object o)`	удалить элемент `o`
`boolean removeAll(Collection<?> c)`	удалить все вхождения элементов, принадлежащих `c`
`boolean retainAll(Collection<?> c)`	удалить элементы, *не* принадлежащие `c`
`boolean removeIf(Predicate<? super E> p)`	удалить элементы, для которых предикат `p` равен `true`

За исключением метода `clear`, который опустошает коллекцию, методы этой группы, равно как и первой, возвращают значение типа `boolean`, показывающее, была ли коллекция изменена в результате их действия.

Если аргумент `o` метода `remove` равен `null`, то метод удаляет из коллекции только один `null`, если он в ней присутствует. В противном случае если в коллекции имеются элементы `e`, для которых `o.equals(e)`, то удаляется один из них, а если нет, то коллекция не изменяется, и метод возвращает `false`. Это поведение – удаление только одного элемента – противоположно поведению метода `removeAll`, который удаляет из данной коллекции все вхождения элементов, принадлежащих переданной коллекции.

В отличие от методов добавления элементов, эти методы – а также методы из следующей группы – принимают в качестве аргументов элементы или коллекции элементов любого типа. В разделе «Сравнение Object и E» главы 18 это проектное решение обсуждается более подробно. Однако эти методы похожи на методы добавления элементов тем, что классы коллекций могут не реализовывать их. Этот вопрос подробно обсуждается в разделе «Фундаментальные проблемы дизайна каркаса коллекций» главы 18.

Опрос содержимого коллекции

`boolean contains(Object o)`	возвращает `true`, если `o` присутствует
`boolean containsAll(Collection<?> c)`	возвращает `true`, если все элементы `c` присутствуют в коллекции
`boolean isEmpty()`	возвращает `true`, если в коллекции нет ни одного элемента
`int size()`	возвращает количество элементов (или значение `Integer.MAX_VALUE`, если оно меньше)

Тут требуются некоторые пояснения по поводу значения, возвращаемого методом `size` для очень больших коллекций. Значение `Integer.MAX_VALUE`, вероятно, было выбрано в предположении, что настолько большие коллекции – больше двух миллиардов элементов – встречаются редко. Но даже если так, альтернативный дизайн – возбуждение исключения вместо возврата произвольно выбранного значения – был бы предпочтительнее, потому что в контракте `size` могло бы быть ясно прописано, что если метод вернул какое-то значение, то это значение правильно.

Обеспечение доступности содержимого коллекции для дальнейшей обработки

`Iterator<E> iterator()`	возвращает `Iterator` по элементам
`Spliterator<E> spliterator`	возвращает `Spliterator` по элементам
`Stream<E> stream()`	возвращает последовательный поток `Stream`, для которого эта коллекция является источником
`Stream<E> parallelStream()`	возвращает параллельный поток `Stream`, для которого эта коллекция является источником
`Object[] toArray()`	копирует содержимое в `Object[]`
`<T> T[] toArray(T[] t)`	копирует содержимое в `T[]` (для любого `T`)
`<T> T[] toArray(IntFunction<T[]> generator)`	копирует содержимое в `T[]`, созданный генератором

Первые два метода в этой группе создают объекты, которые делают элементы коллекции доступными для последовательной или параллельной обработки, как описано в разделе «Параллельные потоки» главы 9. Там было сказано, что чаще всего объекты `Spliterator` употребляются внутри методов `stream` и `parallelStream`, которые поддерживают функциональный стиль обработки элементов потока.

Последние три метода, различные перегруженные варианты `toArray`, возвращают массив, содержащий элементы коллекции. Первый создает новый массив `Object[]`, а второй и третий принимают `T[]` или функцию, порождающую `T[]` заданного размера, соответственно и возвращают `T[]`, содержащий элементы коллекции. Эти методы важны, потому что многие API – в основном, старые и те, для которых производительность стоит на первом месте, – раскрывают методы, принимающие или возвращающие массивы.

В разделе «Как создавать массивы» главы 5 аргументы последних двух методов требуются для того, чтобы передать виртуальной машине сберегаемый тип массива. Во время выполнения метода будет создан новый массив, хотя если массив, переданный в качестве аргумента второму перегруженному варианту `toArray`, достаточно длинный, то он используется для размещения элементов коллекции, которые перезаписывают ранее находившиеся там значения. Но фактически эта возможность менее эффективна, чем передача массива нулевой длины:

```
Collection<String> cs = ...
String[] sa = cs.toArray(new String[0])
```

хотя следующий эквивалентный вызов третьего перегруженного варианта `toArray` выражает намерение наиболее ясно:

```
Collection<String> cs = ...
String[] sa = cs.toArray(String[]::new);
```

Почему в объявлениях двух последних методов допустим *любой* тип `T`? Переменная-тип `T` не связана с параметром-типом коллекции `E`, из-за чего во время выполнения возможны ошибки, которые, казалось бы, должны быть обнаружены на этапе компиляции. Например, код

```
List.of(1, 2, 3).toArray(new String[0]) // исключение сохранения в массиве
```

компилируется успешно, но на этапе выполнения возбуждает исключение `ArrayStoreException`.

Неплохой, на первый взгляд, способ связать две переменных-типа в объявлении второго перегруженного варианта `toArray` мог бы выглядеть как-то так:

```
<T super E> T[] toArray(T[] a)
```

что позволяет создавать массив любого супертипа `E`. Но ограниченные снизу переменные-типы в языке не допускаются, возможно, из-за ограничений алгоритма вывода типов, который применяется со времен появления параметризованных типов в Java 5 (Smith and Cartwright 2008) (см. также Langer, *https://oreil.ly/nZQmH*).

Даже если бы ограниченные снизу параметры-типы были разрешены, это не обеспечило бы полной типобезопасности – по-прежнему было бы возможно написать код, который компилируется без предупреждений, но возбуждает `ArrayStoreException` на этапе выполнения. Например:

```
List<Integer> list = List.of(1, 2, 3);
Object[] array = new String[3];
list.toArray(array); // возбуждает ArrayStoreException
```

Разумеется, никто не стал бы писать такой тупой код, но если вы надумаете разрешить `T` и `E` представлять разумные типы, то подобная ситуация может встретиться в реальной программе. Исключение возбуждается, потому что тип элементов массива на этапе выполнения – `String` (поэтому массив не может принимать значение типа `Integer`), но статический тип элементов переменной `array` – `Object`. Как обычно, ковариантность массивов позволяет обойти статическую проверку типов.

Часто можно встретить прием, позволяющий обойти отсутствие ограниченных снизу параметров-типов с помощью статических вспомогательных методов (см. Langer, *https://oreil.ly/nZQmH*). Метод вида

```
static <O,I extends O> O[] toArray(Collection<I> c, O[] a) {
  return c.toArray(a);
}
```

смог бы обнаружить некоторые ошибки. Однако описанная выше ковариантность массивов все равно позволяет его обмануть.

Остается вопрос: «Почему бы не ограничить элементы массива самим типом `E`, которым параметризована коллекция?» Главная причина заключается в желании оставить возможность задавать для элементов массива более узкий тип, чем у коллекции, в случае, когда все элементы коллекции принадлежат одному и тому же подтипу:

```
List<Object> l = List.of("zero", "one");
String[] a = l.toArray(new String[0]);
```

Здесь список объектов на деле содержит только строки, поэтому его можно преобразовать в `String[]` с помощью операции, аналогичной методу `promote`, описанному в разделе «Тесты экземпляров и приведения» главы 5. Конечно, если список содержит объект, не являющийся строкой, то ошибка обнаруживается на этапе выполнения, а не компиляции (см. раздел «Массивы» главы 2).

```
List<Object> l = List.of("zero", "one",2);
String[] a = l.toArray(new String[0]); // возбуждает ArrayStoreException
```

В общем случае бывает желательно скопировать коллекцию данного типа в массив более узкого типа (например, скопировать список объектов в массив строк, как показано выше) или более широкого типа (например, скопировать список строк в массив объектов).

Иногда требуется отобразить коллекцию данного типа в массив никак не связанного с ним типа (например, отобразить список целых в массив строк), но для этого потребуется нечто большее, чем просто копирование, и таким способом задачу не решить. Вы можете попытаться сделать это, потому что несвязанный параметр-тип `T` во втором и третьем перегруженных вариантах `toArray` позволяет написать такой код, но в результате получите исключение `ArrayStoreException` во время выполнения.

Одним из недостатков этого дизайна является то, что применительно к коллекциям типов-оберток не выполняется автоматическая распаковка в соответствующий массив элементов примитивного типа:

```
List<Integer> l = List.of(0, 1, 2);
int[] a = l.toArray(new int[0]); // ошибка компиляции
```

Это недопустимо, потому что параметр `T` в вызове метода должен – как и любой параметр-тип – быть ссылочным типом. Вызов сработал бы, если бы мы заменили оба вхождения `int` на `Integer`, но зачастую это не выход, потому что из соображений производительности или совместимости нам требуется массив примитивного типа. В таких случаях приходится прибегать к явному копированию массива:

```
jshell> List<Integer> integers = List.of(0, 1, 2);
integers ==> [0, 1, 2]
jshell> int[] ints = new int[integers.size()];
ints ==> int[3] { 0, 0, 0 }
jshell> for (int i=0; i<integers.size(); i++) { ints[i] = integers.get(i); }
jshell> ints
ints ==> int[3] { 0, 1, 2 }
```

В этой ситуации потоковый API предлагает более элегантную и понятную альтернативу:

```
jshell> int[] ints = integers.stream()
   ...> .mapToInt(Integer::intValue)
   ...> .toArray();
ints ==> int[3] { 0, 1, 2 }
```

Использование методов коллекций

Напишем небольшой пример, иллюстрирующий использование классов коллекций. Авторы книги вечно пытаются организовать свое время; нашим последним усилием в этом направлении стало написание собственного (очень простого) менеджера планируемых задач. В этом примере вы увидите, как писать код, придерживаясь классической идиомы каркаса коллекций, а во врезках – с помощью более современного функционального стиля потокового API.

Начнем с определения интерфейса задач и записей для представления двух разных видов задач: написать код и позвонить по телефону.

Вот интерфейс, определяющий задачу:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Task

```
public interface Task extends Comparable<Task> {
  default int compareTo(Task t) {
    return toString().compareTo(t.toString());
  }
}
```

А вот две конкретные реализации типа `Task`:

org/jgcbook/chapter10/A_using_the_methods_of_collection/CodingTask

```
public record CodingTask(String spec) implements Task {}
```

и

org/jgcbook/chapter10/A_using_the_methods_of_collection/PhoneTask

```
public record PhoneTask(String name, String number) implements Task {}
```

Для использования в упорядоченных коллекциях (например, `SequencedSet` и `SequencedMap`) задачи должны допускать сравнение. Другие необходимые им методы – `equals`, `hashCode` и `toString` – автоматически подставляются реализацией записи. Естественный порядок на множестве задач (см. раздел «Интерфейс Comparable» главы 3) соответствует порядку на их строковых представлениях, а, поскольку задачи являются записями, равенство двух задач определяется равенством их строковых полей. Поскольку естественный порядок на множестве строк согласован с равенством, т. е. `compareTo` возвращает `0` тогда и только тогда, когда `equals` возвращает `true`, то и для задач естественный порядок согласован с равенством (см. разделы «Согласованность с equals» главы 3 и «Несогласованность с equals» главы 18).

Задача кодирования определяется своим именем, а задача телефонного звонка – именем и номером телефона абонента. Будучи записями, все строковые поля которых неизменяемы, объекты этих типов сами неизменяемы (см. раздел «Неизменяемость и немодифицируемость» главы 9).

Метод по умолчанию `toString` записи всегда возвращает строку, начинающуюся именем типа записи. Поскольку строка `"CodingTask"` предшествует `"PhoneTask"` в алфавитном порядке и поскольку задачи упорядочены в соответствии с результатами метода `toString`, отсюда следует, что задачи кодирования располагаются раньше задач звонка в естественном порядке – весьма кстати для людей, которые предпочитают писать код, а не болтать по телефону!

Определим также пустую задачу:

org/jgcbook/chapter10/A_using_the_methods_of_collection/EmptyTask

```
public record EmptyTask() implements Task {
  public String toString() {
    return "";
  }
}
```

Поскольку пустая строка предшествует любой другой в естественном порядке на множестве строк, пустая задача оказывается раньше всех прочих в естественном порядке на множестве задач. Эта задача окажется полезной, когда мы станем конструировать представления диапазонов для отсортированных множеств (см. раздел «Получение диапазонных представлений» главы 12).

В примере 10.1 показано, как определить последовательность подлежащих выполнению задач. (Конечно, в реальной системе задачи, скорее всего, хранились бы в базе данных и выбирались оттуда.)

Пример 10.1. *Демонстрационные данные для менеджера задач*

org/jgcbook/chapter10/A_using_the_methods_of_collection/Example101

```
PhoneTask mikePhone = new PhoneTask("Mike", "987 6543");
PhoneTask paulPhone = new PhoneTask("Paul", "123 4567");
CodingTask databaseCode = new CodingTask("db");
CodingTask guiCode = new CodingTask("gui");
CodingTask logicCode = new CodingTask("logic");

Collection<PhoneTask> phoneTasks = new HashSet<>();
Collection<CodingTask> codingTasks = new HashSet<>();
Collection<Task> mondayTasks = new HashSet<>();
Collection<Task> tuesdayTasks = new HashSet<>();

Collections.addAll(phoneTasks, mikePhone, paulPhone);
Collections.addAll(codingTasks, databaseCode, guiCode, logicCode);
Collections.addAll(mondayTasks, logicCode, mikePhone);
Collections.addAll(tuesdayTasks, databaseCode, guiCode, paulPhone);

assert phoneTasks.equals(Set.of(mikePhone, paulPhone));
assert codingTasks.equals(Set.of(databaseCode, guiCode, logicCode));
assert mondayTasks.equals(Set.of(logicCode, mikePhone));
assert tuesdayTasks.equals(Set.of(databaseCode, guiCode, paulPhone));
```

Добавление элементов

Мы можем добавить в план новые задачи:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_1

```
PhoneTask ruthPhone = new PhoneTask("Ruth", "567 1234");
mondayTasks.add(ruthPhone);
assert mondayTasks.equals(Set.of(logicCode, mikePhone, ruthPhone));
```

или объединить два плана с помощью метода `addAll`, реализующего объединение множеств:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_2

```
Collection<Task> allTasks_1 = new HashSet<>(mondayTasks);
allTasks_1.addAll(tuesdayTasks);
assert allTasks_1.equals(Set.of(logicCode, mikePhone, ruthPhone,
    databaseCode, guiCode, paulPhone));
```

Потоковая альтернатива

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_3

```
Collection<Task> allTasks_2 = Stream.of(mondayTasks,tuesdayTasks)
    .flatMap(Collection::stream)
    .collect(Collectors.toSet());
assert allTasks_2.equals(Set.of(logicCode, mikePhone, ruthPhone,
    databaseCode, guiCode, paulPhone));
```

Коллекция `allTasks_2`, получающаяся при работе потоковой альтернативы, может быть не такой же, как `allTasks_1`. Контракт метода `Collectors::toSet` не дает гарантий относительно типа возвращенного множества. В версии Java 21 OpenJDK возвращает `HashSet`, поэтому потоковая альтернатива в действительности дает такой же результат. Но возможно, хотя и маловероятно, что будущие реализации станут возвращать, к примеру, немодифицируемое множество. Если вы хотите точно указать тип возвращаемого множества, то используйте метод `Collectors::toCollection`, передав ему конструктор коллекции. Например, чтобы `allTasks_2` имела типа `HashSet`, замените выделенную строку такой:

```
.collect(Collectors.toCollection(HashSet::new))
```

Этот комментарий относится ко многим из последующих примеров потоковых альтернатив.

Удаление элементов

После того как задача выполнена, ее можно удалить из плана:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_4

```
boolean wasPresent = mondayTasks.remove(mikePhone);
assert wasPresent;
assert mondayTasks.equals(Set.of(logicCode, ruthPhone));
```

Мы также можем очистить план полностью, потому что все задачи выполнены (в реальности мы вряд ли будем часто пользоваться этим методом).

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_5

```
mondayTasks.clear();
assert mondayTasks.equals(Collections.EMPTY_SET);
```

Методы удаления также позволяют комбинировать разными способами целые коллекции. Например, чтобы узнать, какие задачи, кроме телефонных звонков, запланированы на вторник, можно написать:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_6

```
Collection<Task> tuesdayNonPhoneTasks = new HashSet<>(tuesdayTasks);
tuesdayNonPhoneTasks.removeAll(phoneTasks);
assert tuesdayNonPhoneTasks.equals(Set.of(databaseCode, guiCode));
```

Потоковая альтернатива

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_7

```
Collection<Task> tuesdayNonPhoneTasks = tuesdayTasks.stream()
    .filter(t -> ! phoneTasks.contains(t))
    .collect(Collectors.toSet());
```

А чтобы узнать, какие звонки запланированы на тот же день, можно воспользоваться методом `retainAll`, который реализует пересечение множеств:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_8

```
Collection<Task> phoneTuesdayTasks = new HashSet<>(tuesdayTasks);
phoneTuesdayTasks.retainAll(phoneTasks);
assert phoneTuesdayTasks.equals(Set.of(paulPhone));
```

Получить тот же результат можно и по-другому:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_9

```
Collection<PhoneTask> tuesdayPhoneTasks = new HashSet<>(phoneTasks);
tuesdayPhoneTasks.retainAll(tuesdayTasks);
assert tuesdayPhoneTasks.equals(Set.of(paulPhone));
```

Потоковая альтернатива

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program-_10

```
Collection<PhoneTask> tuesdayPhoneTasks = phoneTasks.stream()
    .filter(tuesdayTasks::contains)
    .collect(Collectors.toSet());
```

Заметим, что `phoneTuesdayTasks` имеет тип `Collection<Task>`, тогда как `tuesdayPhoneTasks` – более точный тип `Collection<PhoneTask>`.

Последний пример объясняет сигнатуры методов в этой и следующей группе. Мы уже обсуждали (в разделе «Ограниченные или неограниченные?» главы 2), почему они принимают аргументы типа `Object` или `Collection<?>`, тогда как методы добавления в коллекцию ограничивают аргументы типом параметра. Возьмем для примера метод `retainAll` – его контракт требует удаления из этой коллекции элементов, которые не встречаются в коллекции, переданной в аргументе. Это не накладывает никаких ограничений на содержимое коллекции-аргумента; в примере выше она может содержать экземпляры задач любого вида, а не только `PhoneTask`. Не видно даже причин ограничивать аргумент коллекциями супертипов типа-параметра; на самом деле мы хотим видеть наименее ограничительный тип из всех возможных, каковым является `Collection<?>`. Подобное рассуждение применимо также к методам `remove`, `removeAll`, `contains` и `containsAll`. В разделе «Сравнение Object и E» главы 18 эта тема обсуждается подробнее.

Опрос содержимого коллекции

Эти методы позволяют, например, проверить, что предшествующие операции отработали правильно. Мы здесь будем использовать `assert`, чтобы заставить систему испытать нашу веру в то, что мы все запрограммировали правильно. Например, первое из утверждений ниже неверно и приводит к исключению `AssertionError`, если `tuesdayPhoneTasks` не содержит `paulPhone`:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_11

```
assert tuesdayPhoneTasks.contains(paulPhone);
assert tuesdayTasks.containsAll(tuesdayPhoneTasks);
assert mondayTasks.isEmpty();
assert mondayTasks.size() == 0;
```

Обеспечение доступности содержимого коллекции для последующей обработки

Методы этой группы предоставляют итератор по коллекции, подают ее содержимое потоку или преобразуют в массив.

В разделе «Итерируемые объекты и итераторы» главы 9 было показано, что большинство примеров использования итераторов можно заменить более простым предложением *foreach*, в котором они используются неявно. Но есть случаи, когда *foreach* не помогает; например, явный итератор необходимо использовать, если требуется изменить структуру коллекции, не нарвавшись на исключение `ConcurrentModificationException`, или если нужно обработать два списка параллельно. Чтобы привести пример, предположим, мы решили, что во вторник у нас нет времени на телефонные звонки. Возникает искушение воспользоваться *foreach* и отфильтровать их из нашего списка задач, но эта идея работать не будет по причинам, описанным в разделе «Итерируемые объекты и итераторы» главы 9.

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_12

```
// возбуждает ConcurrentModificationException
for (Task t : tuesdayTasks) {
  if (t instanceof PhoneTask) {
    tuesdayTasks.remove(t);
  }
}
```

Явное использование итератора ничего не дает, если вы продолжаете использовать методы `Collection` для модификации структуры:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_13

```
// возбуждает ConcurrentModificationException
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
  Task t = it.next();
  if (t instanceof PhoneTask) {
    tuesdayTasks.remove(t);
  }
}
```

А вот использование метода *итератора* для изменения структуры приносит желанный результат:

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_14

```
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
  Task t = it.next();
  if (t instanceof PhoneTask) {
    it.remove();
  }
}
assert tuesdayTasks.equals(Set.of(databaseCode, guiCode));
```

Потоковая альтернатива

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_15

```
Set<Task> tuesdayCodeTasks = tuesdayTasks.stream()
    .filter(t -> !(t instanceof PhoneTask))
    .collect(Collectors.toSet());
assert tuesdayCodeTasks.equals(Set.of(databaseCode, guiCode));
```

Преимуществами потоковой версии являются ясность, краткость и возможность распараллелить операции, просто заменив вызов `stream` вызовом `parallelStream`. Расплачиваться за все это придется дополнительной памятью, требуемой для создания новой коллекции.

Чтобы привести другой пример, предположим, что мы ужасно педантичны и желаем хранить все списки задач в порядке возрастания, а при объединении двух списков задач в один хотим сохранить порядок. В примере 10.2 показано, как объединить две коллекции в третью при условии, что итераторы каждой возвращают свои элементы в естественном порядке возрастания. Этот метод предполагает, что объединяемые коллекции не содержат элементов `null`; если таковой встретится, то метод возбуждает исключение `NullPointerException`.

Пример 10.2. *Объединение коллекций с сохранением естественного порядка*

org/jgcbook/chapter10/A_using_the_methods_of_collection/MergeCollections

```
public class MergeCollections {
  static <T extends Comparable<? super T>> List<T> merge
      (Collection<? extends T> c1, Collection<? extends T> c2) {
    List<T> mergedList = new ArrayList<T>();
    Iterator<? extends T> itr1 = c1.iterator();
    Iterator<? extends T> itr2 = c2.iterator();
    T c1Element = getNextElement(itr1);
    T c2Element = getNextElement(itr2);
    // на каждой итерации берется задача у одного из итераторов;
    // продолжать, пока у обоих итераторов не останется задач
    while (c1Element != null || c2Element != null) {
      // использовать текущий элемент c1, если либо текущий c2
      // равен null, либо оба не равны null и c1 предшествует
      // c2 в естественном порядке
      boolean useC1Element = c2Element == null ||
          c1Element != null && c1Element.compareTo(c2Element) < 0;
      if (useC1Element) {
        mergedList.add(c1Element);
```

```
            c1Element = getNextElement(itr1);
          } else {
            mergedList.add(c2Element);
            c2Element = getNextElement(itr2);
          }
        }
        return mergedList;
      }
      static <E> E getNextElement(Iterator<E> itr) {
        if (itr.hasNext()){
          E nextElement = itr.next();
          if (nextElement == null) throw new NullPointerException();
          return nextElement;
        } else {
          return null;
        }
      }
    }
```

Мы не можем использовать коллекции `mondayTasks` и `tuesdayTasks` из примера 10.1 для демонстрации в этом примере, потому что они имеют тип `HashSet`, от итераторов которого не требуется возвращать элементы в каком-то определенном порядке. Но мы можем использовать вместо этого тип `TreeSet`, еще одну реализацию интерфейса `Set`, итераторы которого по умолчанию возвращают элементы множества в естественном порядке (детали см. в разделе «NavigableSet» главы 12):

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_16

```
Collection<Task> sortedMondayTasks = new TreeSet<>(mondayTasks);
Collection<Task> sortedTuesdayTasks = new TreeSet<>(tuesdayTasks);
Collection<Task> mergedTasks =
    MergeCollections.merge(sortedMondayTasks, sortedTuesdayTasks);
assert mergedTasks.equals(
    List.of(databaseCode, guiCode, logicCode, mikePhone, paulPhone));
```

Потоковая альтернатива

org/jgcbook/chapter10/A_using_the_methods_of_collection/Program_17

```
<T> List<T> merge(Collection<? extends T> c1, Collection<? extends T> c2) {
  return Stream.of(c1, c2)
      .flatMap(Collection::stream)
      .sorted()
      .collect(Collectors.toCollection(ArrayList::new));
}
```

Этот метод, очевидно, короче и, пожалуй, понятнее, чем пример 10.2. Алгоритм Timsort, используемый в нем для сортировки, может распознавать расположенные по возрастанию серии, поэтому он не менее эффективен с точки зрения потребления памяти, чем `MergeCollections`. Однако он может объединять коллекции, только применяя метод сравнения; если два подлежащих объединению списка упорядочены как-то иначе – скажем, с сохранением порядка вхождения, когда списки сцепляются, – то у `MergeCollections` есть преимущество.

Реализация Collection

У интерфейса `Collection` нет конкретных реализаций. Класс `AbstractCollection`, содержащий частичную реализацию, – одна из пяти заготовок реализаций (остальными являются `AbstractSet`, `AbstractList`, `AbstractQueue` и `AbstractMap`), он предоставляет функциональность, общую для различных конкретных реализаций каждого интерфейса. Эти заготовки призваны помочь проектировщику новых реализаций интерфейсов каркаса. Например, `Collection` мог бы послужить интерфейсом для «мешков» (неупорядоченных списков), а взявшийся за их реализацию программист мог бы расширить `AbstractCollection` и обнаружить, что большая часть кода уже готова. Рабочий пример с применением `AbstractList` см. в разделе «Настраивайте коллекции с помощью абстрактных классов» главы 17.

Конструкторы коллекций

Прежде чем продолжить рассмотрение четырех основных видов коллекций (множества, очереди, списки и отображения), мы должны дать пояснения по поводу двух форм конструктора, общих для большинства их реализаций. Если взять `HashSet` в качестве примера реализации `Collection`, то эти конструкторы будут иметь вид:

```
public HashSet()
public HashSet(Collection<? extends E> c)
```

Первый создает пустое множество, а второй – множество, содержащее элементы любой коллекции параметризованного типа (или одного из его подтипов). Этот конструктор дает такой же эффект, как создание пустого множества конструктором по умолчанию с последующим добавлением содержимого коллекции методом `addAll`. Иногда его называют «копирующим конструктором», но вообще-то этот термин лучше оставить для конструкторов, которые создают копию объекта того же *класса*, тогда как конструкторы второго вида могут принимать любой объект, реализующий *интерфейс* `Collection<? extends E>`. В книге Bloch (2017, совет 13) предложен термин «преобразующий конструктор».

Реализации интерфейса `Map` устроены аналогично. Например, среди конструкторов `HashMap` есть такие два варианта:

```
public HashMap()
public HashMap(Map<? extends K,? extends V> m)
```

Не у всех классов коллекций есть конструкторы обоих видов – например, `ArrayBlockingQueue` нельзя создать, не зафиксировав емкость, а `SynchronousQueue` вообще не может содержать элементов, поэтому конструктор второго вида не актуален. Кроме того, у многих классов коллекций есть и другие конструкторы, помимо этих двух, но какие именно, зависит не от реализуемого ими интерфейса, а от самой реализации; эти дополнительные конструкторы служат для конфигурирования реализации.

Заключение

В этой главе мы изучили интерфейс `Collection`, изначальный корневой тип в каркасе коллекций. Понимать, как он работает, крайне важно для эффективного использования каркаса.

С другой стороны, `SequencedCollection` – недавнее дополнение. Он объединяет классы, имеющие следующую общую особенность: они представляют последовательность значений, обладают операциями, применимыми к началу и к концу последовательности, и допускают обход в обоих направлениях. Это тема следующей главы.

Глава 11

Интерфейс SequencedCollection

Интерфейс `SequencedCollection` занимает особое место в дизайне каркаса коллекций. Он не описывает контракт отдельной абстрактной структуры данных, а унифицирует поведения нескольких ранее существовавших типов – `List`, `NavigableSet`, `Deque` и `LinkedHashSet`, – разбросанных по иерархии типов и больше, по существу, никак не связанных между собой (см. рис. 11.1).

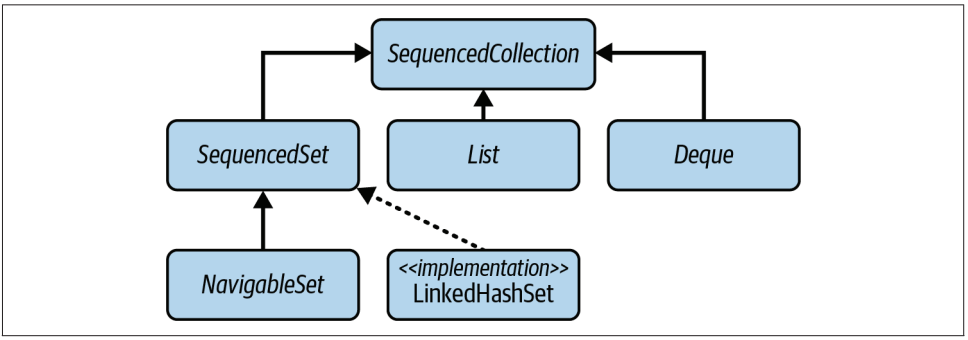


Рис. 11.1. Иерархия типов `SequencedCollection`

Общего у этих коллекций то, что они представляют последовательность элементов, порядок которых семантически значим (определенный на них полный порядок в документации JDK называется *порядком вхождения* (encounter order)) и может быть внутренним или навязанным извне (см. раздел «Упорядоченные коллекции» главы 8).

Методы SequencedCollection

Впервые набор методов, которые теперь обобщены на все коллекции с порядком вхождения, появился в классе `Deque` (см. раздел «Deque» главы 13); они позволяли добавлять, инспектировать и удалять первый и последний элемент последовательности. Некоторые возможности были доступны в этих коллекциях и раньше, правда, без систематизации и под другими именами; например, `NavigableSet` поддерживал удаление первого и последнего элемента, но не их инспекцию, тогда как `List` напрямую поддерживал только добавление

последнего элемента, а во всех остальных операциях нужно было указывать номер позиции. В табл. 12.1 показано соответствие между вызовами методов `NavigableSet` и их эквивалентов в `SequencedCollection`, в табл. 14.1 такое же соответствие для `List`. Для `Deque` такая таблица не нужна, потому что, за исключением `reversed`, все операции `SequencedCollection` позаимствованы у `Deque`.

Конечно, было бы невозможно унифицировать поведение таких разных коллекций, не вводя некоторых специальных случаев. Например, не имеет смысла добавлять первый или последний элемент в `NavigableSet`, учитывая, что позиции всех элементов в этой коллекции определяются внутренним алгоритмом сортировки. Другой пример – немодифицируемые списки не могут поддержать такие структурные модификации, как добавление и удаление элементов. Эти специальные случаи будут рассмотрены в главах, посвященных конкретным типам коллекций.

Единственный новый метод, предоставляемый `SequencedCollection`, – `reversed`, который дает представление в обратном порядке (см. раздел «Представления» главы 9). Это позволяет обрабатывать любую упорядоченную коллекцию с порядком вхождения от конца к началу (ранее было доступно только клиентам `NavigableSet` с помощью метода `descendingSet`).

Добавление элементов

`void addFirst(E e)`	добавить `e` первым элементом в коллекцию
`void addLast(E e)`	добавить `e` последним элементом в коллекцию

Эти методы добавляют элементы в начало или в конец `SequencedCollection`. Их контракты говорят, что после успешного выполнения первым (соответственно последним) элементом коллекции в порядке вхождения должен быть элемент, переданный методу.

Инспектирование элементов

`E getFirst()`	вернуть первый элемент этой коллекции
`E getLast()`	вернуть последний элемент этой коллекции

Эти методы инспектируют элемент в начале или в конце `SequencedCollection`. Вызов метода инспектирования для пустой коллекции возбуждает исключение `NoSuchElementException`.

Удаление элементов

`E removeFirst()`	удалить и вернуть первый элемент этой коллекции
`E removeLast()`	удалить и вернуть последний элемент этой коллекции

Эти методы удаляют элементы из начала или конца `SequencedCollection`. Вызов метода удаления для пустой коллекции возбуждает исключение `NoSuchElementException`.

Генерирование обращенного представления

`SequencedCollection<E> reversed()` вернуть обращенное представление этой коллекции

Этот метод возвращает представление коллекции в обратном порядке. Конечно, мы предпочли бы, чтобы вызов `reversed` для, скажем, `NavigableSet` (интерфейс, наследующий `SequencedCollection`), вернул `NavigableSet`, а не параметризованную `SequencedCollection`, как указано в объявлении. По этой причине все четыре интерфейса, наследующие `SequencedCollection`, – `List`, `SequencedSet`, `NavigableSet` и `Deque`, – объявляют ковариантные версии (см. раздел «Ковариантное переопределение» главы 3) метода `reversed`. Например, объявление `reversed` в `SequencedSet` имеет вид:

```
SequencedSet<E> reversed()
```

Аналогичные переопределенные версии объявлены в трех других интерфейсах.

Обратный порядок влияет на итерирование и другие операции, зависящие от порядка. Он влияет на порядок элементов, возвращаемых `Iterable::foreach`, `iterator`, `listIterator`, `spliterator`, `stream` и `parallelStream`. Он меняет на противоположный смысл `first` и `last` в семействе методов `add`/`get`/`remove`, а также в методах `indexOf`/`lastIndexOf` интерфейса `List`. Порядок помещения элементов в массив методами из семейства `toArray` также отражает обращенный порядок вхождения. Эти эффекты распространяются и на представления возвращенного представления.

Контракт оговаривает, что любая успешная модификация этого представления должна быть видна в базовой коллекции, но обратное – видимость в представлении изменений, внесенных в базовую коллекцию, – зависит от реализации. Самая употребительная реализация, `ArrayList`, этим свойством обладает: модификации, внесенные в базовую коллекцию, видны в обращенном представлении.

Заключение

В этой главе мы дали краткий обзор интерфейса `SequencedCollection`, потому что он был включен не ради совершенно новых возможностей, а чтобы унифицировать возможности уже существовавших до того упорядоченных реализаций, в частности `NavigableSet` и `Deque`, и сделать их доступными в других местах (например, в `LinkedHashSet` и `LinkedHashMap`).

Интерфейсы, описанные в этой и предыдущей главе, `Collection` и `SequencedCollection`, необходимы для рассмотрения предмета следующей главы: интерфейса `Set` и его реализаций.

Глава 12

Множества

Множеством называется совокупность элементов, не содержащая дубликатов; добавление в множество уже присутствующего в нем элемента не дает никакого эффекта. В интерфейсе `Set<E>` объявлены те же методы, что и в `Collection`, но он определен отдельно, чтобы можно было добавить контракт метода `add` (и `addAll`, определяемого в терминах `add`), отражающий это свойство. Метод `equals` переопределен: контракт `Set` говорит, что `Set` может быть равно только другому `Set` и только в том случае, когда оба множества имеют одинаковый размер и состоят из одних и тех же элементов. Метод `hashCode` также переопределен, как должно быть всегда, когда переопределяется `equals` (см. «Хеш-таблицы» в разделе «Реализации» главы 9). Хеш-код множества `Set` равен сумме хеш-кодов его элементов.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter12.*

Рассмотрим пример операций над множествами. Возвращаясь к примеру менеджера задач, который мы описали в разделе «Использование методов коллекций» главы 10, предположим, что в понедельник у нас есть свободное время для телефонных звонков. Мы можем создать новую коллекцию задач на понедельник, добавив все телефонные задачи в множество запланированных на этот день. Пусть `mondayTasks` и `phoneTasks` объявлены, как в примере 10.1.

Пользуясь множеством, мы можем написать:

org/jgcbook/chapter12/Program_1

```
Set<Task> phoneAndMondayTasks = new HashSet<>(mondayTasks);
phoneAndMondayTasks.addAll(phoneTasks);
assert phoneAndMondayTasks.equals(Set.of(logicCode, mikePhone, paulPhone));
```

Это работает благодаря способу обработки дубликатов. Задача `mikePhone`, хотя она и находится одновременно в `mondayTasks` и `phoneTasks`, встречается в `phoneAndMondayTasks` только один раз – и в данном приложении это хорошо, потому что мы точно не хотим выполнять одну и ту же задачу дважды.

Определение множества: отношения эквивалентности

Несмотря на кажущуюся простоту, чтобы понять, как именно работают разные реализации `Set`, нам нужно более внимательно присмотреться к тому, как определяется дубликат, – т. е. к *отношению эквивалентности* на множестве. В предыдущем примере использовался класс `HashSet`, для которого отношение эквивалентности задает метод `equals`: иными словами, два объекта в `HashSet` являются дубликатами тогда и только тогда, когда метод `equals`, вызванный от имени одного с аргументом, равным второму, возвращает `true`. Этот выбор может показаться очевидным, но он не единственный (хотя в документации `Set` и его реализаций для версии JDK 21 часто подразумевается именно он). В некоторых других реализациях также используется `equals`, например в `CopyOnWriteArraySet` и семействе классов, которое мы называем *UnmodifiableSet* (см. раздел «UnmodifiableSet» ниже в этой главе). Другой способ, не столь распространенный, дает отношение тождественности: множество, в котором используется это отношение, содержит ссылки на все добавленные в него уникальные объекты. Примером может служить класс `EnumSet` (см. раздел «EnumSet» ниже в этой главе) – хотя, поскольку перечисления являются синглтонами (т. е. существуют в единственном экземпляре), результат метода `equals` совпадает с результатом отношения тождественности для всех сравнений. Другой пример – представление в виде множества ключей `IdentityHashMap` или любого множества, созданного из `IdentityHashMap` (см. раздел «IdentityHashMap» главы 15) с помощью метода `newSetFromMap` интерфейса `Collections`.

Третий вариант отношения эквивалентности определен контрактом `NavigableSet`. Множество `NavigableSet` (см. раздел «NavigableSet» ниже в этой главе) хранит элементы отсортированными в соответствии с отношением порядка – естественного или определенного компаратором `Comparator` (см. главу 3). В `NavigableSet` это отношение порядка используется еще и для другой цели: два объекта считаются эквивалентными, если они равны согласно этому отношению, т. е. если метод сравнения возвращает 0 – независимо от того, удовлетворяют ли они отношению равенства.

Тот факт, что в разных множествах используются разные отношения эквивалентности, не создает никаких трудностей, коль скоро различные отношения совместимы (см. главу 3), т. е. дают один и тот же результат применительно к любой паре значений. Но в случае несовместимых отношений возникает путаница; например, два объекта могут не удовлетворять отношению равенства, даже если, согласно отношению порядка на множестве, они равны. К несчастью, эта путаница усугубляется документацией интерфейса `Set` и его реализаций, поскольку местами авторы забывают упомянуть, что в некоторых реализациях используются отношения эквивалентности, отличные от `equals`. (Надеемся, что в будущих версиях документации эта проблема будет исправлена.)

Чтобы избежать путаницы, вы должны понимать последствия использования различных отношений эквивалентности для разных множеств. Одно очевидное последствие заключается в том, что множества могут содержать дубликаты, удовлетворяющие требованиям `equals`, и наоборот, могут отвергать добавление элементов, не удовлетворяющих этим требованиям. Другое, не столь очевидное, последствие касается равенства множеств: чтобы определить, рав-

ны ли множества A и B, мы должны для каждого элемента B проверить, есть ли эквивалентный ему элемент в A. Если роли поменяются и при этом A и B пользуются разными отношениями эквивалентности, то результаты могут отличаться, т. е. отношение равенства перестает быть симметричным.

Более подробное обсуждение этой проблемы с примерами имеется в разделе «Несогласованность с equals» главы 18.

Реализации интерфейса Set

При использовании методов Collection в главе 10 мы подчеркивали, что они будут работать с любой реализацией Collection, а не только Set. Но на практике вы, конечно, должны выбрать конкретную реализацию. В этой главе рассматриваются различные реализации Set, которые предоставляются каркасом коллекций и различаются как скоростью выполнения базовых операций add, contains и итерирования, так и порядком, в котором итераторы возвращают элементы. В конце этой главы мы сведем в единую таблицу характеристики производительности различных реализаций.

Из всех реализаций Set, имеющихся в каркасе коллекций, четыре реализуют интерфейс Set непосредственно, а остальные наследуют подынтерфейсу SequencedSet (см. раздел «SequencedSet» ниже в этой главе). В этом разделе будут рассмотрены именно эти четыре класса: HashSet, CopyOnWriteArraySet, EnumSet и семейство реализаций, которые мы называем *UnmodifiableSet* (см. раздел «UnmodifiableSet» ниже в этой главе).

HashSet

Этот класс – наиболее употребительная реализация Set. Как следует из названия, он реализован с помощью *хеш-таблицы* – массива, в котором позиции элементов определены их содержимым. Поскольку хранение элементов в хеш-таблицах, равно как и их извлечение, определены содержимым, они прекрасно приспособлены для реализации операций Set (каркас коллекций использует их также в различных реализациях Map). Например, чтобы реализовать операцию contains(Object o), нужно поискать элемент o и вернуть true, если он найден.

Позиция элемента в хеш-таблице вычисляется хеш-функцией от его содержимого. Хеш-функции проектируются так, чтобы обеспечить как можно больший разброс результатов (*хеш-кодов*) по элементам соответствующего типа. Например, в классе String вычисление хеш-кода производится так:

```
int hash = 0;
for (char ch : str.toCharArray()) {
  hash = hash * 31 + ch;
}
```

Традиционно хеш-таблицы получают индекс в таблице в виде остатка от деления хеш-кода на длину таблицы. На практике деление – медленная операция, поэтому в классах из каркаса коллекций используются битовые маски. Это означает, что существенны только биты в конце хеш-кода, поэтому для его вычисления используются простые числа (здесь 31), так как умножение на простое число не приводит к потере информации в младших битах, как было бы, к примеру, в случае умножения на 2.

Очевидно, что если в таблице меньше позиций, чем может быть сохранено значений, то иногда два разных значения будут хешироваться в одну и ту же позицию (это называется *коллизией*). Например, никакая индексированная значениями типа `int` таблица не может быть достаточно большой для хранения всех возможных строк без коллизий. Проблему можно смягчить, придумав хорошую хеш-функцию – распределяющую элементы по таблице равномерно, – но, если коллизия все-таки происходит, то нужен способ хранить конфликтующие элементы в одной и той же позиции, или *ячейке* (bucket). Для этого часто применяется связанная структура – список или дерево, – как показано на рис. 12.1. Более подробно мы будем изучать связанные структуры позже, а пока достаточно сказать, что разные элементы, попавшие в одну ячейку, все же остаются доступными за счет организации цепочки ссылок.

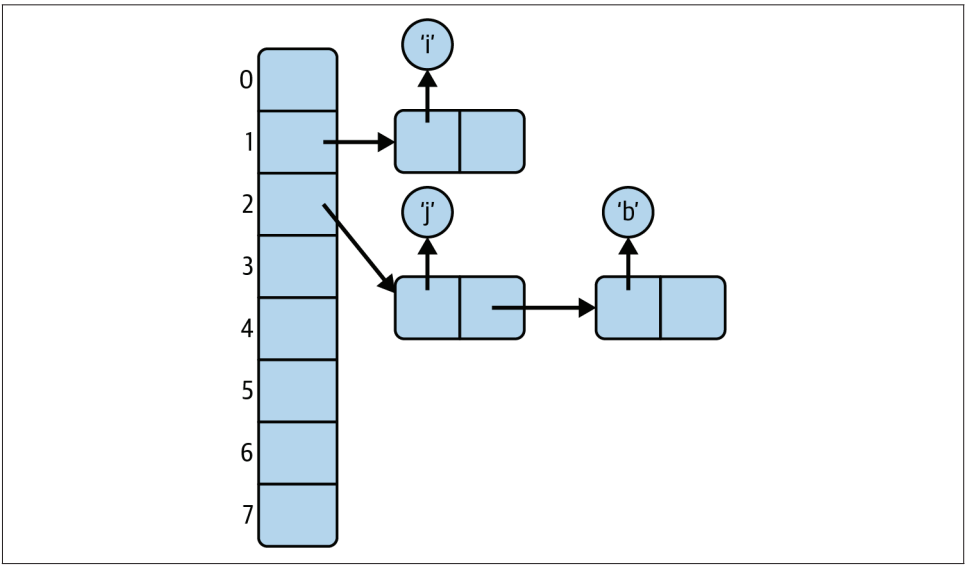


Рис. 12.1. Хеш-таблица с цепочкой переполнения

На рис. 12.1 изображена ситуация, получающаяся после выполнения следующего кода в OpenJDK из версии Java 21:

```
Set<Character> s1 = new HashSet<Character>(8);
s1.add('j');
s1.add('i');
s1.add('b');
```

Значения индексов элементов таблицы были вычислены с использованием трех младших битов (для длины таблицы 8) хеш-кода каждого элемента. В этой реализации хеш-код `Character` – это просто значения Юникода содержащегося в нем символа. На практике хеш-таблицы обычно гораздо больше. Заметим также, что этот рисунок упрощен: в действительности `HashSet` реализована с помощью специализированного класса `HashMap` (см. раздел «HashMap» главы 15), поэтому в каждом элементе цепочки находится не одна, а две ссылки – на ключ и на значение. Следовательно, картина больше похожа на изобра-

женную на рис. 15.2, только значения для всех ключей одинаковы. На рис. 12.1 показан только ключ, потому что, когда хеш-таблица используется для представления множества, лишь ключ и важен.

При отсутствии коллизий стоимость вставки или извлечения элемента постоянна. По мере заполнения хеш-таблицы коллизии становятся все вероятнее; в предположении, что хеш-функция качественная, вероятность коллизии в слабо заполненной таблице пропорциональна ее *коэффициенту заполнения*, который определяется как число элементов в таблице, поделенное на ее емкость (количество ячеек). Если коллизия все же имеет место, то необходимо создать структуру переполнения – связанный список или дерево, – а затем обойти ее, что увеличивает стоимость вставки.

В качестве структуры переполнения обычно используется линейный список, но, когда его длина для данной ячейки превышает определенный порог, он преобразуется в красно-черное дерево (см. раздел «TreeSet» ниже в этой главе). На узлах этого дерева используется отношение порядка, индуцированное методом `compareTo` ключей, если они реализуют интерфейс `Comparable`. Если же нет, то для упорядочения узлов используется сравнение по имени класса, а если и это невозможно, то сравнение хеш-кодов идентификатора объекта. Этих резервных способов упорядочения узлов следует избегать, когда важна максимальная производительность. То же предостережение относится к классам `HashMap` и `ConcurrentHashMap`. Если размер хеш-таблицы фиксирован, то ее производительность ухудшается по мере добавления новых элементов и роста коэффициента заполнения. Чтобы этого не происходило, размер таблицы увеличивается путем *перехеширования* – копирования всех элементов в новую таблицу большего размера, – как только коэффициент заполнения достигает определенного порога.

Для обхода хеш-таблицы необходимо посетить каждую ячейку и посмотреть, занята она или нет. Следовательно, количество команд пропорционально сумме емкости хеш-таблицы и числа содержащихся в ней элементов. Поскольку итератор посещает каждую ячейку по очереди, порядок возврата элементов зависит от их хеш-кодов, так что никаких гарантий относительно того, в каком порядке они возвращаются, нет. В реализации OpenJDK из Java 21 хеш-таблица, показанная на рис. 12.1, возвращает элементы в порядке возрастания индекса и обратного обхода связанных списков. Распечатка дает следующий результат:

```
[i, j, b]
```

В следующем разделе мы будем рассматривать класс `LinkedHashSet`, вариант этой реализации, в которой итератор возвращает элементы в том порядке, в каком они вставлялись.

Главное достоинство реализации множеств с помощью хеш-таблицы – постоянная временная сложность (для слабо заполненных таблиц и хорошей хеш-функции) базовых операций `add`, `remove`, `contains` и `size`. Главный недостаток – низкая производительность для сильно заполненных таблиц, а также эффективность обхода: обход таблицы требует посещения каждой ячейки, поэтому стоимость включает множитель, определяемый длиной таблицы, вне зависимости от размера множества, которое она содержит.

Конструкторы HashSet

`HashSet` имеет стандартные конструкторы, с которыми мы познакомились в разделе «Конструкторы коллекций» главы 10, а также два дополнительных конструктора:

```
HashSet(int initialCapacity)
HashSet(int initialCapacity, float loadFactor)
```

Оба они создают пустое множество, но дают некоторый контроль над размером хеш-таблицы – ее размер не меньше заданной емкости (в текущей реализации он равен следующей за этой емкостью степенью двойки) – и факультативно над желаемым коэффициентом заполнения. У большинства коллекций на основе хеш-таблиц есть похожие конструкторы. Их можно использовать для создания таблицы, достаточно большой для хранения всех ожидаемых элементов, обойдясь без дорогостоящих операций изменения размера. Но на практике они только запутывают – потому что параметр типа `int` часто неправильно интерпретируют как количество элементов, хотя на самом деле он нужен для вычисления размера таблицы. Кроме того, эти параметры трудно использовать, потому что их правильное вычисление на основе ожидаемого максимального числа элементов зависит от реализации и чревато ошибками. Поэтому в Java 19 были добавлены статические фабричные методы, которые принимают только один параметр – ожидаемое количество элементов. Эти методы рекомендованы, потому что их проще использовать и они в меньшей степени зависят от изменений реализации. Таким фабричным методом для `HashSet` является `newHashSet`:

```
static <T> HashSet<T> newHashSet(int numElements)
```

Коллекция `HashSet` не синхронизирована и не потокобезопасна, ее итераторы обладают свойством быстрого отказа.

CopyOnWriteArraySet

Функциональная спецификация `CopyOnWriteArraySet` тоже прямо выводится из контракта `Set`, но характеристики производительности у нее иные, чем у `HashSet`. Этот класс реализован как тонкая обертка вокруг экземпляра `CopyOnWriteArrayList`, который, в свою очередь, основан на массиве. Массив считается неизменяемым; любая модификация множества приводит к созданию нового массива. Поэтому сложность метода `add` равна $O(N)$, как и метода `contains`, который по необходимости реализуется в виде линейного поиска. Очевидно, что не следует использовать `CopyOnWriteArraySet` в контексте, где ожидается много операций поиска или вставки. Но реализация на основе массива означает, что стоимость итерации составляет $O(1)$ на каждый элемент – быстрее, чем у `HashSet`, – и есть еще одно преимущество, весьма привлекательное в некоторых приложениях: она обеспечивает потокобезопасность (см. раздел «Коллекции и потокобезопасность» главы 9) без увеличения стоимости операций чтения. Этим она выгодно отличается от коллекций, в которых потокобезопасность всех операций обеспечивается использованием блокировок (например, синхронизированных коллекций, которые обсуждаются в разделе «Синхронизированные коллекции» главы 16). Операции над блокировками

всегда являются потенциально узким местом в многопоточных приложениях. С другой стороны, операции чтения в коллекциях с копированием при записи реализуются в базовом массиве, а благодаря его неизменяемости к ним может обращаться любой поток, не опасаясь вмешательства со стороны конкурентной операции записи.

Когда имеет смысл использовать множество с такими характеристиками? Типичная ситуация – управление разделяемой конфигурацией в многопоточной среде. Например, серверное приложение могло бы хранить глобальное конфигурационное множество разрешенных IP-адресов, которое несколько потоков читают очень часто, но обновляется оно редко. Процессу обновления должно быть запрещено мешать операциям чтения; в случае реализации множества с блокировками накладные расходы, необходимые для гарантии такого поведения, поровну несут операции чтения и записи, тогда как в `CopyOnWriteArraySet` они возлагаются только на операции записи. Это разумно в ситуации, когда операции чтения производятся гораздо чаще, чем вносятся изменения в конфигурацию сервера.

Поскольку для класса `CopyOnWriteArraySet` не предусмотрено никаких конфигурационных параметров, единственными конструкторами являются стандартные, обсуждавшиеся в разделе «Конструкторы коллекций» главы 10.

EnumSet

`EnumSet` – множество, которое может содержать подмножество элементов перечисления, от пустого до всего множества. Этот класс существует для того, чтобы можно было воспользоваться преимуществом эффективных реализаций, возможных, когда максимальное число возможных элементов фиксировано, и каждому можно присвоить уникальный индекс. Эти два условия выполнены для множества элементов из одного и того же класса `Enum`; количество ключей фиксировано элементами типа перечисления, а метод `ordinal` возвращает гарантированно уникальные значения для каждого элемента. Кроме того, значения, возвращаемые `ordinal`, образуют компактный диапазон, начинающийся с нуля, – идеально для использования в качестве индексов массива или, в стандартной реализации, индексов битового вектора. Поэтому методы `add`, `remove` и `contains` реализованы как битовые операции с постоянной временной сложностью. Битовые операции над одним словом выполняются очень быстро, а для представления `EnumSet` над типами перечисления можно использовать значение типа `long`, позволяющее представить до 64 элементов. Например, перечисление `Day` объявлено так:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

а затем множество выходных, определенное как `EnumSet.of(Day.SATURDAY, DAY.SUNDAY)`, на внутреннем уровне представлено значением типа `long`, равным 65, т. е. $2^0 + 2^6$. Перечисления большего размера обрабатываются аналогично, но с небольшими накладными расходами из-за необходимости включать несколько слов в представление.

`EnumSet` – запечатанный абстрактный класс, который реализует эти различные представления с помощью двух разных подклассов, закрытых на уровне пакета: один для перечислений, которые можно представить позициями од-

ного `long`, т. е. содержащих не более 64 элементов, другой – для всех остальных. Эти конкретные реализации скрыты от программиста, а снаружи видны фабричные методы, которые вызывают конструктор нужного подкласса. Следующая группа статических фабричных методов позволяет создавать экземпляры `EnumSet` с разным начальным содержимым – пустым, только с заданными элементами перечисления и со всеми элементами перечисления:

`<E extends Enum<E>> EnumSet<E> of(E first, E... rest)`	создать множество, первоначально содержащее указанные элементы
`<E extends Enum<E>> EnumSet<E> range(E from, E to)`	создать множество, первоначально содержащее все элементы в диапазоне, определяемом двумя конечными точками
`<E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)`	создать множество, первоначально содержащее все элементы `elementType`
`<E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)`	создать множество `elementType`, первоначально пустое

`EnumSet` содержит сбереженный тип своих элементов, который будет использоваться на этапе выполнения для проверки допустимости новых элементов. Этот тип передается фабричными методами двумя разными способами. Методы `of` и `range` получают по меньшей мере один элемент перечисления, у которого можно запросить класс, где он объявлен (т. е. класс `Enum`, которому элемент принадлежит). Методам `allOf` и `noneOf` элементы перечисления не передаются, но зато передается объект класса (см. раздел «Классная альтернатива» главы 5).

Наиболее распространенные случаи создания `EnumSet` оптимизированы второй группой методов, которая позволяет эффективно создавать множества с одним, двумя, тремя, четырьмя и пятью элементами типа перечисления:

```
<E extends Enum<E>> EnumSet<E> of(E e)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
```

Эти методы появились задолго до немодифицируемых коллекций, но последние сегодня используются настолько чаще, что иногда полезно напомнить, что не все фабричные методы порождают немодифицируемые коллекции. Вот вам пример: создаваемые здесь экземпляры `EnumSet` модифицируемые.

Третий набор методов позволяет создавать `EnumSet` из существующей коллекции:

`<E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)`	создать `EnumSet` с тем же типом элементов, что у `s`, и с теми же элементами

`<E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)`	создать `EnumSet` из элементов `c`, которая должна содержать хотя бы один элемент
`<E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)`	создать `EnumSet` с тем же типом элементов, что у `s`, содержащий элементы, отсутствующие в `s`

Коллекция, переданная в качестве аргумента второму варианту `copyOf`, должна быть непустой, чтобы можно было определить тип элемента.

`EnumSet` соблюдает контракт `Set`, дополненный требованием о том, что итераторы должны возвращать элементы в естественном порядке (том, в котором объявлены элементы перечисления). Этот класс не является потокобезопасным, но, в отличие от несинхронизированных коллекций общего назначения, его итераторы не отказывают быстро. Хотя документация (в версии Java 21) утверждает, что они слабо согласованы (см. раздел «Коллекции и потокобезопасность» главы 9), это верно лишь отчасти: для самого часто используемого конкретного подкласса `EnumSet` – перечисления с 64 или менее элементами – итератор на самом деле слепковый.

UnmodifiableSet

Ни в документации, ни в коде каркаса коллекций вы не найдете упоминаний имени *UnmodifiableSet<E>*. Это имя мы придумали для обозначения семейства закрытых на уровне пакета классов, к которым программисты клиентских приложений никогда не обращаются по имени, но которые важны, потому что предоставляют реализацию немодифицируемых множеств, полученных от различных перегруженных вариантов фабричных методов `Set.of` и `Set.copyOf`. Свойства членов этого семейства описаны в документации `Set`:

- они немодифицируемы: элементы нельзя ни добавлять, ни удалять. Вызов любого изменяющего метода возбуждает исключение `UnsupportedOperationException`;
- они не допускают `null`. Попытка поместить в них при создании элементы `null` приводит к исключению `NullPointerException`;
- они не выносят дубликатов в момент создания. Если передать фабричному методу повторяющиеся элементы, то он возбудит исключение `IllegalArgumentException`.

Фабричные методы (статические, как и все фабричные методы; в следующих таблицах ключевое слово `static` для краткости опущено) можно разбить на три группы, аналогичные группам для `EnumSet`, с тем очевидным отличием, что множества после создания оказываются немодифицируемыми, а экземпляры `EnumSet` нет. Первая группа содержит только перегруженный вариант метода `of`, который не имеет аргументов и возвращает пустое немодифицируемое множество:

`<E> Set<E> of()`	возвращает немодифицируемое множество, не содержащее ни одного элемента

Вторая группа методов позволяет создавать немодифицируемое множество, содержащее до 10 заданных элементов:

`<E> Set<E> of(E e1)`	возвращает немодифицируемое множество, содержащее один элемент
`<E> Set<E> of(E e1, E e2)`	возвращает немодифицируемое множество, содержащее два элемента
`<E> Set<E> of(E e1, E e2, E e3)`	возвращает немодифицируемое множество, содержащее три элемента

дополнительные перегруженные варианты с числом аргументов от 4 до 9

`<E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)`	возвращает немодифицируемое множество, содержащее десять элементов

Третья группа методов позволяет создавать немодифицируемое множество из коллекции, массива или списка с переменным числом аргументов:

`<E> Set<E> copyOf(Collection<? extends E> coll)`	возвращает немодифицируемое множество, содержащее произвольное число элементов
`<E> Set<E> of(E... elements)`	возвращает немодифицируемое множество, содержащее произвольное число элементов

Классы, входящие в состав *UnmodifiableSet*, благодаря своей немодифицируемости могут воспользоваться массивами фиксированной длины в качестве базовых структур. Не неся накладных расходов ни на пустые ячейки таблиц, ни на связанные структуры переполнения, эти реализации потребляют значительно меньше места, чем хешированные структуры. Итерирование также более эффективно и обладает дополнительным преимуществом хорошей пространственной локальности (см. раздел «Память» главы 9). Но за быстрое итерирование приходится расплачиваться тем, что определить принадлежность элемента множеству можно только линейным поиском с временной сложностью $O(N)$.

В реализации из OpenJDK итераторы для *UnmodifiableSet* обладают необычной характеристикой: порядок обхода определен случайно для каждого экземпляра виртуальной машины. Причина такого решения в том, чтобы не повторять странную проблему совместимости, имевшую место для `HashSet`. В прошлом разработчики клали в основу тестов и даже производственного кода кажущийся фиксированным порядок обхода `HashSet`. Но поскольку в контракте `HashSet` не оговорен порядок обхода, авторы реализаций изменяли его при переходе от одной версии к другой, что приводило к поломкам кода после обновления. Итераторы для немодифицируемых множеств спроектированы так, чтобы разработчики больше не сталкивались с этой ошибкой.

Представления отображений в виде множеств

В каркасе коллекций многие множества реализованы как обертки вокруг соответствующего отображения, хотя сами отображения инкапсулированы и не видны клиенты. Однако обратное неверно: у некоторых отображений нет со-

ответствующего множества, и в частности у тех трех, что мы увидим в главе 15: `WeakHashMap`, `IdentityHashMap` и `ConcurrentHashMap`. Чтобы получить множество для одного из них с такими же порядком, характеристиками конкурентности и производительности, можно вызвать метод `Collections::newSetFromMap` от имени непустого отображения типа `Map<E,Boolean>`, где `E` — тип элемента множества, которое вы собираетесь создать. Например, чтобы создать конкурентное множество `Integer`, можно было бы написать:

```
Set<Integer> concurrentIntegerSet =
    Collections.newSetFromMap(new ConcurrentHashMap<Integer,Boolean>())
```

Эта идиома гарантирует, что после создания множества-представления прямой доступ к базовому отображению будет невозможен, как того требует спецификация `newSetFromMap`.

SequencedSet

`SequencedSet` – это внешне или внутренне упорядоченное множество `Set`, которое также раскрывает методы `SequencedCollection`. Оно объединяет методы этих двух интерфейсов, дополняя их только в одном отношении: предоставляет ковариантное переопределение метода `reversed` интерфейса `SequencedCollection`, чтобы вернуть значение типа `SequencedSet` (см. раздел «Генерирование обращенного представления» главы 11). У этого интерфейса есть прямая реализация, `LinkedHashSet`, и, кроме того, его расширяет интерфейс `NavigableSet` (фактически через свой родительский интерфейс `SortedSet`, но см. вступительные замечания в разделе «NavigableSet» ниже). На рис. 12.2 показаны связи между этими типами.

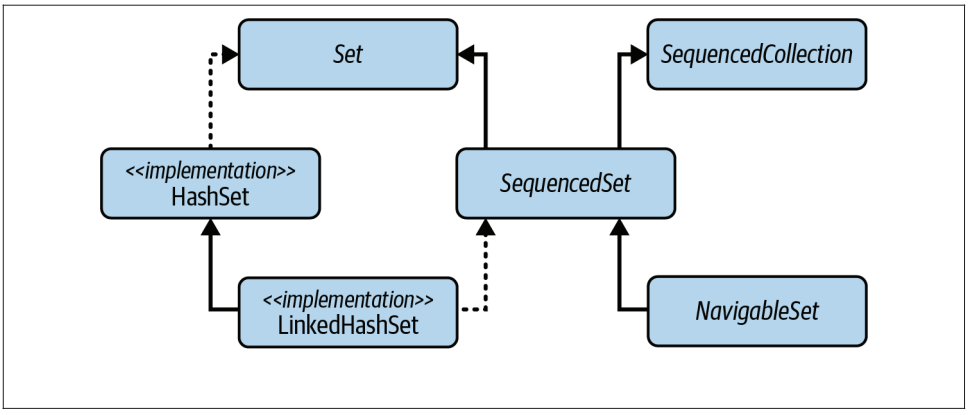


Рис. 12.2. SequencedSet и связанные с ним типы

LinkedHashSet

Этот класс наследует `HashSet` и реализует интерфейс `SequencedSet` посредством хранения своих элементов в связанном списке, как показано изогнутыми стрелками на рис. 12.3.

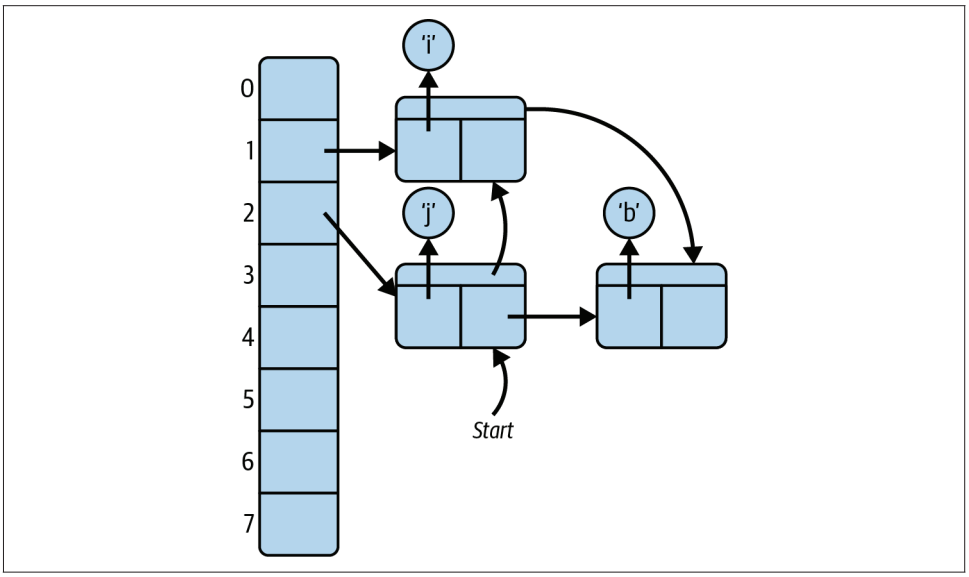


Рис. 12.3. Связанная хеш-таблица

Ситуация на рис. 12.3 возникла бы при выполнении следующего кода:

```
jshell> LinkedHashSet.<Character>newLinkedHashSet(3)
$1 ==> []
jshell> Collections.addAll($1, 'j', 'i', 'b');
$2 ==> true
```

Итераторы `LinkedHashSet` возвращают элементы в том порядке, в котором они вставлялись:

```
jshell> $1
$1 ==> [j, i, b]
```

Доступны все методы `SequencedSet`:

```
jshell> $1.getLast()
$3 ==> 'b'
jshell> $1.reversed()
$4 ==> [b, i, j]
jshell> $1.addLast('k')
jshell> $1
$1 ==> [j, i, b, k]
```

Но заметим, что вызов `addFirst` или `addLast` от имени уже существующего элемента приводит к его перемещению соответственно в первую или последнюю позицию:

```
jshell> $1.addFirst('k')
jshell> $1
$1 ==> [k, j, i, b]
```

У связанной структуры есть полезное следствие в терминах улучшенной эффективности обхода: метод `next` требует постоянного времени, потому что

для посещения каждого элемента по очереди можно использовать связанный список. Это полная противоположность классу HashSet, для которого посетить нужно каждую ячейку хеш-таблицы вне зависимости от того, занята она или нет. В случае большой, но разреженной таблицы это крайне неэффективно, но в общем случае накладные расходы, связанные с хранением связанного списка, означают, что отдавать предпочтение LinkedHashSet перед HashSet имеет смысл, только если такая ситуация вероятна в вашем приложении.

Конструкторы LinkedHashSet предлагают те же возможности, что конструкторы HashSet, для конфигурирования базовой хеш-таблицы. И по аналогии с HashSet имеется современный фабричный метод newLinkedHashSet, который принимает в качестве аргумента ожидаемое число элементов и оптимально выбирает размер таблицы.

Этот класс не синхронизирован и не потокобезопасен, его итераторы обладают свойством быстрого отказа.

NavigableSet

Интерфейс NavigableSet добавляет к контракту SequencedSet гарантию того, что итератор будет обходить множество в порядке возрастания элементов, а, кроме того, добавляет методы для нахождения элементов, соседних с целевым значением. В отличие от LinkedHashSet элементы упорядочены внутренне методом сравнения, взятым из естественного порядка, или компаратором. Поэтому addFirst и addLast не способны выполнить свои контракты и возбуждают исключение UnsupportedOperationException. До появления NavigableSet единственным подынтерфейсом Set был интерфейс SortedSet, который гарантирует порядок обхода, но не объявляет методов поиска ближайшего соседа. SortedSet по-прежнему включен в JDK – он расширяет SequencedSet, и, в свою очередь, расширяется NavigableSet – но больше не представляет интереса, так как на платформе у него нет прямых реализаций.

Мы можем использовать NavigableSet, чтобы добавить полезную функциональность в менеджер задач. До сих пор методы Collection и Set не помогали в упорядочении задач, а это, безусловно, одно из главных требований, предъявляемых к менеджеру задач. В примере 12.1 определена запись, PriorityTask, которая добавляет к задаче приоритет. Всего существует три приоритета, HIGH, MEDIUM и LOW, объявленные таким образом, что HIGH оказывается первым (т. е. наименьшим) при естественном порядке, что соответствует интуитивному представлению: задачи с «приоритетом 1» должны быть рассмотрены раньше задач с «приоритетом 2»:

org/jgcbook/chapter12/C_navigable_set/Priority

```
public enum Priority { HIGH, MEDIUM, LOW }
```

Чтобы сравнить две PriorityTask, компаратор priorityTaskCmpr сначала сравнивает их приоритеты: если приоритеты не равны, то задача с более высоким приоритетом считается большей. Если приоритеты равны, то компаратор переходит к использованию естественного порядка на задачах. Таким образом, два объекта считаются равными, только если они действительно равны, т. е. естественный порядок согласован с equals, и проблемы, которые мы обсуждали

в начале этой главы, не возникают. А могли бы они возникнуть, если бы мы определили компаратор так, что все задачи с равным приоритетом считаются равными. При таком определении `NavigableSet` могло бы содержать не более одной задачи каждого приоритета.

Пример 12.1. *Запись* `PriorityTask`

org/jgcbook/chapter12/C_navigable_set/PriorityTask

```
public record PriorityTask(Task task, Priority priority)
  implements Comparable<PriorityTask> {

    static Comparator<PriorityTask> priorityTaskCmpr =
      Comparator.comparing(PriorityTask::priority)
          .thenComparing(PriorityTask::task);

    public int compareTo(PriorityTask pt) {
      return priorityTaskCmpr.compare(this, pt);
  }
}
```

В следующем далее коде показано, как `NavigableSet` работает с множеством `PriorityTask`:

org/jgcbook/chapter12/C_navigable_set/Program_1

```
NavigableSet<PriorityTask> priorityTasks = new TreeSet<PriorityTask>();

priorityTasks.add(new PriorityTask(mikePhone, Priority.MEDIUM));
priorityTasks.add(new PriorityTask(paulPhone, Priority.HIGH));
priorityTasks.add(new PriorityTask(databaseCode, Priority.MEDIUM));
priorityTasks.add(new PriorityTask(guiCode, Priority.LOW));

assert priorityTasks.toString().equals("""
    [PriorityTask[task=PhoneTask[name=Paul, number=123 4567], priority=HIGH],
    PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
   PriorityTask[task=PhoneTask[name=Mike, number=987 6543], priority=MEDIUM],
    PriorityTask[task=CodingTask[spec=gui], priority=LOW]]""");
```

Методы NavigableSet

Методы, объявленные в интерфейсе `NavigableSet`, можно отнести к шести группам.

Получение компаратора

`Comparator<? super E> comparator()`	возвращает `Comparator`, если в `NavigableSet` таковой имеется; в противном случае возвращает `null`

Этот метод получает компаратор множества, если тот был задан при конструировании. Если в множестве используется естественный порядок элементов, то метод возвращает `null`. В качестве параметра-типа `Comparator` используется ограниченный тип, для того чтобы `NavigableSet`, параметризованный `E`, мог применять для упорядочения компаратор, определенный для любого супертипа `E`. Например, в разделе «Фруктовый пример» главы 3 `Comparator<Fruit>` можно было использовать для `NavigableSet<Apple>`.

Получение первого и последнего элемента

`E first()`	возвращает первый элемент множества
`E last()`	возвращает последний элемент множества

Это методы для получения элементов с обеих сторон `SequencedSet` (см. табл. 12.1 в разделе «NavigableSet теперь расширяет SequencedSet» ниже в этой главе, где описано соответствие с методами `SequencedSet`). Если множество пусто, то эти операции возбуждают исключение `NoSuchElementException`.

Удаление первого и последнего элемента

`E pollFirst()`	получает и удаляет первый (самый маленький) элемент или возвращает `null`, если множество пусто
`E pollLast()`	получает и удаляет последний (самый большой) элемент или возвращает `null`, если множество пусто

Это методы для удаления элементов с обеих сторон `SequencedSet` (см. табл. 12.1, где описано соответствие с методами `SequencedSet`). Они аналогичны одноименным методам `Deque` (см. раздел «Deque» главы 13) и служат для поддержки использования `NavigableSet` в приложениях, где требуются очереди. Например, в версии менеджера задач из этого раздела мы могли бы сначала получать из списка высокоприоритетные задачи, готовые к выполнению:

org/jgcbook/chapter12/C_navigable_set/Program_2

```
PriorityTask nextTask = priorityTasks.pollFirst();
assert nextTask.toString().equals(
   "PriorityTask[task=PhoneTask[name=Paul, number=123 4567], priority=HIGH]");
```

Получение диапазонных представлений

Когда возникает необходимость работать с упорядоченным множеством значений, полезно рассматривать их как *диапазон*. Например, если имеется множество событий с временными метками, то может возникнуть желание посмотреть только те, которые имели место в определенный промежуток времени. В случае задач типа `PriorityTask` мы могли бы обрабатывать только те, для которых приоритеты принадлежат заданному диапазону, скажем, с высоким и средним приоритетом. Диапазон всегда можно извлечь, обойдя множество и собрав элементы, попадающие в диапазон, но тип данных `NavigableSet` может существенно упростить эту задачу, предоставив методы, извлекающие диапазоны за одну операцию.

Интервал, подобный диапазонному представлению, может быть *открытым*, *полуоткрытым* и *замкнутым* – в зависимости от того, сколько конечных точек он содержит. Например, диапазон чисел x, для которых $0 \leqslant x \leqslant 1$, замкнут, потому что содержит обе конечные точки 0 и 1. Диапазоны $0 \leqslant x < 1$ и $0 < x \leqslant 1$ полуоткрыты, потому что содержат только одну конечную точку, а диапазон $0 < x < 1$ открыт, потому что не содержит конечных точек.

Это предварительное пояснение необходимо, потому что для каждого метода в этой группе имеется два перегруженных варианта: один унаследован от `SortedSet` и возвращает полуоткрытое представление `SortedSet`, а второй опре-

делен в NavigableSet и возвращает представление NavigableSet, которое может быть открытым, полуоткрытым или замкнутым в зависимости от выбора пользователя.

`SortedSet<E> subSet(E fromValue, E toValue)`	возвращает представление части множества от `fromValue` (включая) до `toValue` (не включая)
`SortedSet<E> headSet(E toValue)`	возвращает представление части множества до `toValue` (не включая)
`SortedSet<E> tailSet(E fromValue)`	возвращает представление части множества, элементы которого больше или равны `fromValue`

Метод SortedSet возвращает представление элементов множества от fromValue – включая его, если он присутствует, – до toValue, исключая последний. Заметим, что, хотя в документации этих методов аргументы операций называются «элементами» – fromElement и toElement, – это не совсем так: на самом деле они сами не обязаны быть членами множества.

`NavigableSet<E> subSet(E fromValue, boolean fromInclusive, E toValue, boolean toInclusive)`	возвращает представление части множества от `fromValue` до `toValue`
`NavigableSet<E> headSet(E toValue, boolean inclusive)`	возвращает представление части множества до `toValue`
`NavigableSet<E> tailSet(E fromValue, boolean inclusive)`	возвращает представление части множества от `fromValue`

Методы NavigableSet похожи, но позволяют для каждой границы определить, включать ее или нет. В нашем примере эти методы могли бы быть полезны для создания различных представлений множества priorityTask. Например, мы можем использовать headSet, чтобы получить представление, состоящее из задач с высоким и средним приоритетом. Для этого потребуется специальная задача, предшествующая всем прочим в порядке следования задач; по счастью, мы определили класс EmptyTask специально для этой цели в разделе «Использование методов коллекций» главы 10. С его помощью легко извлечь все задачи, предшествующие любой низкоприоритетной:

org/jgcbook/chapter12/C_navigable_set/Program_3

```
PriorityTask firstLowPriorityTask = new PriorityTask(new EmptyTask(), Priority.
LOW);

NavigableSet<PriorityTask> highAndMediumPriorityTasks =
    priorityTasks.headSet(firstLowPriorityTask, false);

assert(highAndMediumPriorityTasks.toString()).equals("""
    [PriorityTask[task=PhoneTask[name=Paul, number=123 4567], priority=HIGH],
    PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
                PriorityTask[task=PhoneTask[name=Mike,    number=987    6543],
priority=MEDIUM]]""");
```

На самом деле, поскольку мы знаем, что реальных задач без каких бы то ни было деталей не бывает, мы можем использовать такую задачу как начальную точку полуоткрытого интервала:

org/jgcbook/chapter12/C_navigable_set/Program_4

```
PriorityTask firstMediumPriorityTask =
    new PriorityTask(new EmptyTask(), Priority.MEDIUM);

NavigableSet<PriorityTask> mediumPriorityTasks =
     priorityTasks.subSet(firstMediumPriorityTask, true, firstLowPriorityTask,
false);

assert (mediumPriorityTasks.toString()).equals("""
    [PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
                PriorityTask[task=PhoneTask[name=Mike,    number=987    6543],
priority=MEDIUM]]""");
```

Все изменения, произведенные в базовом множестве, отражаются в представлении:

org/jgcbook/chapter12/C_navigable_set/Program_5

```
PriorityTask logicCodeMedium = new PriorityTask(logicCode, Priority.MEDIUM);
priorityTasks.add(logicCodeMedium);
assert mediumPriorityTasks.toString().equals("""
    [PriorityTask[task=CodingTask[spec=db], priority=MEDIUM],
    PriorityTask[task=CodingTask[spec=logic], priority=MEDIUM],
                PriorityTask[task=PhoneTask[name=Mike,    number=987    6543],
priority=MEDIUM]]""");
```

Чтобы понять, как это работает, представьте, что все возможные значения лежат в каком-то порядке на прямой, как числа на числовой оси. Диапазон определяется как фиксированный отрезок этой прямой независимо от того, каковы значения в оригинальном множестве. Поэтому подмножество `NavigableSet` и диапазон позволят работать с теми элементами `NavigableSet`, которые в данный момент попали в диапазон.

Обратное также верно; изменения в представлении, в том числе структурные, отражаются в базовом множестве:

```
assert priorityTasks.size() == 5;
mediumPriorityTasks.remove(logicCodeMedium);
assert priorityTasks.size() == 4;
```

Получение ближайших соседей

`E ceiling(E e)`	возвращает наименьший элемент `x` в множестве такой, что $x \geq e$, или `null`, если такого элемента не существует
`E floor(E e)`	возвращает наибольший элемент `x` в множестве такой, что $x \leq e$, или `null`, если такого элемента не существует
`E higher(E e)`	возвращает наименьший элемент `x` в множестве такой, что $x > e$, или `null`, если такого элемента не существует
`E lower(E e)`	возвращает наибольший элемент `x` в множестве такой, что $x < e$, или `null`, если такого элемента не существует

Эти методы полезны для навигации на небольшие расстояния. Например, предположим, что имеется отсортированное множество строк, и мы хотим найти последние три строки его подмножества, ограниченного сверху строкой

"x-ray", включая саму эту строку, если она присутствует в множестве. Методы NavigableSet позволяют легко решить эту задачу.

org/jgcbook/chapter12/C_navigable_set/Program_7

```
NavigableSet<String> stringSet = new TreeSet<>();
Collections.addAll(stringSet, "abc", "cde", "x-ray" ,"zed");
Optional<String> last = Optional.ofNullable(stringSet.floor("x-ray"));
assert last.equals(Optional.of("x-ray"));
Optional<String> secondToLast = last.map(stringSet::lower);
assert secondToLast.equals(Optional.of("cde"));
Optional<String> thirdToLast = secondToLast.map(stringSet::lower);
assert thirdToLast.equals(Optional.of("abc"));
```

Заметим, что в полном согласии с общим дизайном каркаса коллекций NavigableSet возвращает значения null, когда нужно обозначить отсутствие элементов, тогда как методы first и last интерфейса SortedSet возбудили бы в этом случае исключение NoSuchElementException. По этой причине следует избегать элементов null в множествах типа NavigableSet, и на самом деле более новая реализация, ConcurrentSkipListSet, их вообще не допускает (хотя TreeSet вынуждена допускать null ради обратной совместимости).

Обход множества в обратном порядке

`NavigableSet<E> descendingSet()`	возвращает представление этого множества, в котором элементы следуют в обратном порядке
`Iterator<E> descendingIterator()`	возвращает итератор в обратном порядке

Методы этой группы позволяют обходить NavigableSet в порядке убывания (т. е. обратном) с той же легкостью, что и в порядке возрастания. В качестве простой иллюстрации обобщим предыдущий пример, в котором используются методы поиска ближайшего соседа. Предположим, что вместо нахождения трех последних строк отсортированного множества, ограниченных сверху строкой "x-ray", мы хотим обойти все строки множества в порядке их убывания.

org/jgcbook/chapter12/C_navigable_set/Program_8

```
NavigableSet<String> headSet = stringSet.headSet(last.get(), true);
NavigableSet<String> reverseHeadSet = headSet.descendingSet();
assert reverseHeadSet.toString().equals("[x-ray, cde, abc]");
String conc = " ";
for (String s : reverseHeadSet) {
  conc += s + " ";
}
assert conc.equals(" x-ray cde abc ");
```

Если итеративная обработка включает внесение структурных изменений в множество и в качестве реализации используется TreeSet (итераторы которого обладают свойством быстрого отказа), то мы должны будем использовать явный итератор, чтобы избежать исключения ConcurrentModificationException:

org/jgcbook/chapter12/C_navigable_set/Program_9

```
for (Iterator<String> itr = headSet.descendingIterator(); itr.hasNext(); ) {
  itr.next(); itr.remove();
}
assert headSet.isEmpty();
```

NavigableSet теперь расширяет SequencedSet

Хотя в новой версии интерфейс `NavigableSet` стал расширять `SequencedSet`, ни один из новых методов не предоставляет какой-то другой функциональности; все они просто переименованные версии существующих методов. В табл. 12.1 показано соответствие между новыми и старыми методами.

Таблица 12.1. Сравнение методов интерфейса `SequencedSet` с эквивалентными методами `NavigableSet`

SequencedSet	NavigableSet
`getFirst`	`first` (унаследован от `SortedSet`)
`getLast`	`last` (унаследован от `SortedSet`)
`removeFirst`	`pollFirst`
`removeLast`	`pollLast`
`addFirst`	Метод не поддерживается для внутренне упорядоченных коллекций
`addLast`	Метод не поддерживается для внутренне упорядоченных коллекций
`reversed`	`descendingSet`

Причина пересечения методов `SequencedSet` и `NavigableSet` в том, что первые шесть методов `SequencedSet` в табл. 12.1 были скопированы из `Deque`. До появления `SequencedSet` в Java 21 интерфейсы `SortedSet` и `NavigableSet` имели несколько методов, похожих на методы `Deque`, но с другими именами и немного отличающимся поведением:

- `NavigableSet::first` и `NavigableSet::last`, унаследованные от `SortedSet`, – то же самое, что `SequencedSet::getFirst` и `SequencedSet::getLast`, и возбуждают исключение `NoSuchElementException`, если коллекция пуста;
- `NavigableSet::pollFirst` и `NavigableSet::pollLast` удаляют и возвращают соответственно первый и последний элемент. Однако они отличаются от `SequencedSet::removeFirst` и `SequencedSet::removeLast` тем, что для пустой коллекции возвращают `null`, а не возбуждают исключение `NoSuchElementException`.

TreeSet

Потратим немного времени на рассмотрение производительности деревьев в сравнении с другими типами реализаций, встречающимися в каркасе коллекций.

Дерево – структура данных, которую имеет смысл брать, если приложению необходима быстрая вставка и получение отдельных элементов, и, кроме того, элементы должны выбираться в отсортированном порядке.

Например, предположим, что вы хотите найти в множестве все слова с заданным префиксом – типичное требование в графических приложениях, когда выпадающий список в идеале должен показывать все возможные элементы с префиксом, введенным пользователем. Хеш-таблица не умеет возвращать элементы в отсортированным порядке, а список не может быстро находить элементы по их содержимому. Но дерево может и то и другое.

В информатике дерево – это ветвящаяся структура для представления иерархии. Древовидные структуры данных заимствуют значительную часть терминологии у генеалогических деревьев, хотя есть и некоторые различия; самым важным является то, что в информатике каждый узел дерева имеет только одного родителя (кроме корня, у которого родителей нет вообще). Важным классом являются *двоичные* деревья, каждый узел которых может иметь не более двух родителей. На рис. 12.4 показан пример двоичного дерева, содержащего слова предложения «figure shows an example of a binary tree containing the words of this sentence in alphabetical order», расположенные в алфавитном порядке.

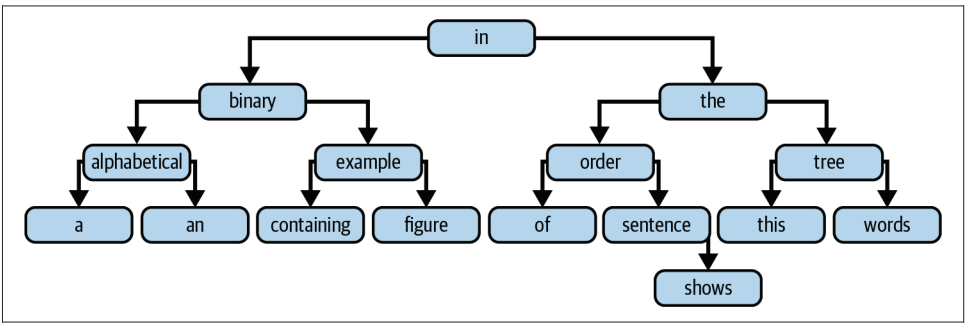


Рис. 12.4. Упорядоченное сбалансированное двоичное дерево

Самое важное свойство этого дерева можно увидеть, взглянув на любой нелистовой узел – скажем, содержащий слово *the*: все узлы, находящиеся ниже него слева, содержат слова, предшествующие *the* в алфавитном порядке, а все узлы, находящиеся ниже него справа, содержат слова, следующие за *the*. Чтобы найти слово, мы начинаем с корня и спускаемся вниз, производя сравнение на каждом уровне, поэтому стоимость поиска или вставки элемента пропорциональна глубине дерева.

Какова же глубина дерева, содержащего N элементов? Полное двоичное дерево с двумя уровнями содержит три элемента (т. е. $2^2 - 1$), а дерево с тремя уровнями – семь элементов ($2^3 - 1$). В общем случае двоичное дерево с N полными уровнями будет содержать $2^N - 1$ элементов, а глубина дерева, содержащего N элементов, не превышает $\log N$ (поскольку $2^{\log N} = N$). Как N растет гораздо медленнее, чем 2^N, так $\log N$ растет гораздо медленнее N, поэтому метод `contains` для большого дерева работает гораздо быстрее, чем для списка с тем же самым числом элементов. Конечно, не так быстро, как для хеш-таблицы, операции которой в идеале должны занимать постоянное время, но у дерева есть то важное преимущество перед хеш-таблицей, что его итератор возвращает элементы в отсортированном порядке.

Но не все двоичные деревья демонстрируют такую замечательную производительность. На рис. 12.4 показано *сбалансированное* двоичное дерево, каждый узел которого имеет в точности или почти одинаковое количество потомков с каждой стороны. У несбалансированного дерева производительность может быть намного хуже – в худшем случае такая же низкая, как у связанного списка (см. рис. 12.5). В `TreeSet` используется тип данных, называемый *красно-черным деревом* (см., например, Sedgewick and Wayne 2011), обладающий тем преимуществом, что если в результате вставки или уда-

ления элемента дерево оказывается несбалансированным, то может быть перебалансировано за время $O(\log N)$.

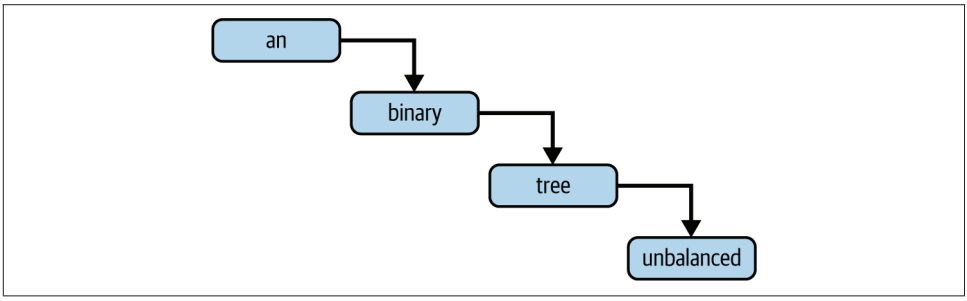


Рис. 12.5. Несбалансированное двоичное дерево

Помимо стандартных конструкторов, у `TreeSet` есть конструктор, который позволяет задать компаратор (см. раздел «Интерфейс Comparator» главы 3), и конструктор, создающий `TreeSet` из `SortedSet`:

`TreeSet(Comparator<? super E> c)`	конструирует пустое множество, которое будет сортироваться с помощью заданного компаратора
`TreeSet(SortedSet<E> s)`	возвращает новое множество, содержащее элементы переданного, отсортированные в том же порядке

Объявление второго из них довольно близко к стандартному «преобразующему конструктору» (см. раздел «Конструкторы коллекций» главы 10):

`TreeSet(Collection<? extends E> c)`	конструирует новое множество, содержащее элементы переданного, отсортированные в естественном порядке

Как объяснил Джошуа Блох в книге «Effective Java» (2017, совет 52), вызов одного из двух перегруженных конструкторов или методов, которые принимают параметры родственных типов, может дать сбивающие с толку результаты. Дело в том, что в Java вызовы перегруженных конструкторов и методов разрешаются во время компиляции на основе статического типа аргумента, поэтому применение операции приведения к аргументу может оказать существенное влияние на результат вызова, как показывает следующий код:

org/jgcbook/chapter12/C_navigable_set/Program_10

```
// сконструировать и заполнить NavigableSet, итератор которого возвращает
// элементы в порядке, обратном естественному:
NavigableSet<String> base = new TreeSet<>(Comparator.reverseOrder());
Collections.addAll(base, "b", "a", "c");

// вызвать два разных конструктора TreeSet, передав им только что
// сконструированное множество, но с разными статическими типами
NavigableSet<String> sortedSet1 = new TreeSet<>((Set<String>)base);
```

```
NavigableSet<String> sortedSet2 = new TreeSet<>(base);
// и теперь оба множества обходятся в разном порядке
List<String> forward = new ArrayList<>(sortedSet1);
List<String> backward = new ArrayList<>(sortedSet2);
assert !forward.equals(backward);
assert forward.reversed().equals(backward);
```

Эта проблема свойственна конструкторам всех отсортированных коллекций в каркасе (`TreeSet`, `TreeMap`, `ConcurrentSkipListSet` и `ConcurrentSkipListMap`). Чтобы не столкнуться с ней в собственных классах, выбирайте типы параметров различных перегруженных вариантов, так чтобы аргумент типа, подходящего для одного варианта, нельзя было привести к типу, подходящему для другого. Если это невозможно, то оба перегруженных варианта должны быть спроектированы так, чтобы они вели себя одинаково при передаче одного и того же аргумента, каким бы ни был его статический тип. Например, объект `PriorityQueue` (см. раздел «PriorityQueue» главы 13), сконструированный из коллекции, пользуется определенным в ней порядком независимо от того, является статический тип переданного конструктору аргумента типом `PriorityQueue` или `SortedSet`, содержащим `Comparator`, или просто типом `Collection`. Чтобы добиться этой цели, преобразующий конструктор выполняет тест `instanceof`, стремясь определить тип переданной коллекции; если тип совпадает с `SortedSet` или `PriorityQueue`, то извлекается `Comparator`, а если нет, то используется естественный порядок.

Коллекция `TreeSet` не синхронизирована и не потокобезопасна, ее итераторы обладают свойством быстрого отказа.

ConcurrentSkipListSet

`ConcurrentSkipListSet` стал первой конкурентной реализацией множества. В его основе лежит *список с пропусками*, современная альтернатива двоичным деревьям, описанным в предыдущем разделе. Список с пропусками для множества – это последовательность *связанных списков*, каждый из которых является цепочкой звеньев, состоящих из двух полей: в одном хранится значение, а в другом ссылка на следующее звено. Элементы вставляются и удаляются из связанного списка за постоянное время с помощью манипуляций указателями, как показано на рис. 12.6a и b соответственно.

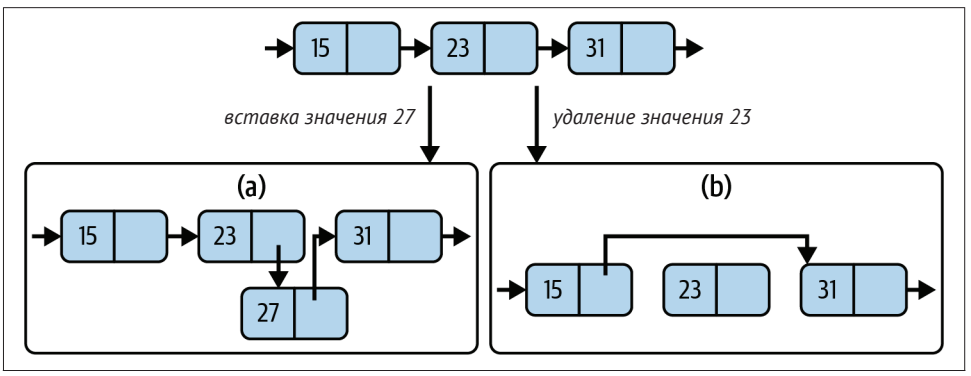


Рис. 12.6. Модификация связанного списка

На рис. 12.7 показан список с пропусками, состоящий из трех связанных списков, помеченных «Уровень 1», «Уровень 2» и «Уровень 3». Первый связанный список коллекции (уровня 0 на рисунке) содержит элементы множества, отсортированные в естественном порядке или компаратором множества. Каждый список уровня больше 0 содержит подмножество списка под ним, выбранное случайно с некоторой фиксированной вероятностью. Для этого примера предположим, что вероятность равна 0.5; в среднем каждый список будет содержать половину элементов списка, расположенного под ним. Для навигации по ссылкам требуется фиксированное время, поэтому самый быстрый способ найти элемент – начать с начала верхнего списка (левого конца) и продвигаться так далеко, как возможно, прежде чем опуститься на уровень ниже.

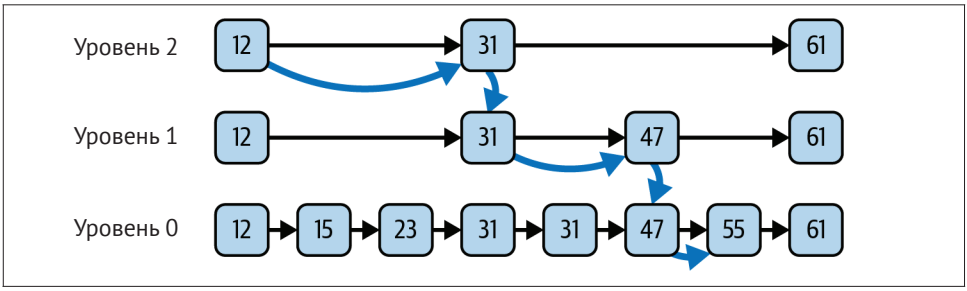


Рис. 12.7. Поиск в списке с пропусками

Изогнутые стрелки на рис. 12.7 показывают ход поиска элемента 55. Поиск начинается с элемента 12 с левого конца верхнего списка уровня 2, доходит до элемента 31 на этом уровне, затем обнаруживает, что следующий элемент равен 61, т. е. больше искомого. Поэтому поиск опускается на уровень ниже, и весь процесс повторяется; элемент 47 еще меньше 55, но 61 снова больше, поэтому мы опускаемся еще на один уровень и на следующем шаге находим искомое значение.

Вставка элемента в список с пропусками всегда включает по меньшей мере его вставку на уровень 0. А после этого нужно ли его вставлять на уровень 1? Если уровень 1 содержит в среднем половину элементов уровня 0, то мы должны подбросить монету (т. е. сделать случайный выбор с вероятностью 0.5), чтобы решить, вставлять ли его также на уровень 1. Если после подбрасывания монеты элемент вставлен на уровень 1, то процесс повторяется для уровня 2 и т. д. При удалении элемента из списка с пропусками он удаляется из каждого уровня, где встречается.

Если все подбрасывания монеты неудачны, то может оказаться, что все списки уровня, большего 0, пусты – или заполнены, что так же плохо. Но вероятность таких исходов очень мала, и анализ показывает, что с очень высокой вероятностью списки с пропусками дают производительность, сравнимую с двоичными деревьями: временная сложность поиска, вставки и удаления равна $O(\log N)$. Но у них есть важное преимущество при конкурентном использовании: безблокировочные алгоритмы вставки и удаления. Для двоичных деревьев алгоритмы с такими свойствами неизвестны.

Итераторы `ConcurrentSkipListSet` слабо согласованы.

Сравнение реализаций множеств

В табл. 12.2 приведено сравнение производительности различных реализаций интерфейса `Set`. При выборе реализации эффективность – лишь один из факторов, влияющих на решение. Некоторые реализации специализированы для особых ситуаций; например, `EnumSet` следует всегда использовать для представления элементов перечисления (и только для этой цели). Аналогично `CopyOnWriteArraySet` следует использовать, только когда размер множества остается сравнительно мал, операций чтения гораздо больше, чем операций записи, необходима потокобезопасность и приемлемы итераторы, допускающие только чтение.

Таблица 12.2. Сравнительная производительность различных реализаций `Set`

	add	contains	Next	Примечания
`HashSet`	*O*(1)	*O*(1)	*O*(*h*/*N*)	*h* – емкость таблицы
`LinkedHashSet`	*O*(1)	*O*(1)	*O*(1)	
`CopyOnWriteArraySet`	*O*(*N*)	*O*(*N*)	*O*(1)	
`EnumSet`	*O*(1)	*O*(1)	*O*(1)[a]	
`TreeSet`	*O*(log *N*)	*O*(log *N*)	*O*(log *N*)	
`ConcurrentSkipListSet`	*O*(log *N*)	*O*(log *N*)	*O*(1)	

[a] В реализации `EnumSet` для типов перечислений, содержащих более 64 значений, сложность `next` в худшем случае равна *O*(log *m*), где *m* – число элементов перечисления.

Это оставляет для общего назначения реализации `HashSet`, `LinkedHashSet`, `TreeSet`, `ConcurrentSkipListSet` и представление `ConcurrentHashMap` в виде множества, полученного методом `newSetFromMap`.

Первые три используются в основном в однопоточных приложениях. Они не являются потокобезопасными, поэтому в многопоточном коде их можно использовать только в сочетании с блокировками на стороне клиента или с обертками `Collection.synchronizedSet` (см. раздел «Синхронизированные коллекции» главы 16). Если не требуется, чтобы множество было отсортированным, то выбирать стоит между `HashSet` и `LinkedHashSet`. Если приложение будет часто обходить множество, то лучше остановиться на `LinkedHashSet`. Если множество должно поддерживать методы интерфейса `NavigableSet`, то используйте `TreeSet`.

В многопоточной среде выбирать приходится между представлением множества, получаемым от `ConcurrentHashMap::newSetFromMap`, и `ConcurrentSkipListSet`. Первый вариант выбирается по умолчанию, из соображений эффективности, а второй поддерживает методы `NavigableSet`.

Заключение

В этой главе мы видели разнообразие способов реализации обманчиво простой абстракции неупорядоченной коллекции без дубликатов. Эти реализации отражают различные требования к множествам, предъявляемые в разных сценариях, встречающихся при практическом программировании.

Предмет следующей главы, интерфейс `Queue`, полезен совсем в других ситуациях, чем `Set` и другие классы коллекций. Очереди – это каналы, используемые объектами для взаимодействия между собой, а не для хранения данных. Они не являются частью состояния других объектов в том смысле, в каком эту роль обычно играют другие коллекции; мы увидим, как эта особенность формирует API интерфейса `Queue`.

Глава 13

Очереди

Очередь – это коллекция, предназначенная для хранения элементов, ожидающих обработки, и извлечения их в том порядке, в котором они должны быть обработаны. Соответствующий интерфейс `Queue<E>` в каркасе коллекций имеет несколько реализаций, представляющих различные порядки извлечения. Во многих реализациях подразумевается, что задачи должны обрабатываться в том порядке, в котором помещались в очередь (первым пришел, первым ушел – *FIFO*), но возможны и другие правила – например, в каркасе коллекций есть классы очередей, для которых порядок обработки зависит от приоритета задачи. Интерфейс `Queue` был добавлен в каркас позднее, отчасти из-за необходимости очередей для средств обеспечения конкурентности, добавленных в то же время. И действительно, беглый взгляд на иерархию реализаций, изображенную на рис. 13.1, показывает, что все реализации `Queue` в каркасе коллекций находятся в пакете `java.util.concurrent`.

Классическое требование к очередям в конкурентных системах возникает, когда несколько задач исполняются несколькими параллельно работающими потоками. Бытовой пример этой ситуации дает одна очередь пассажиров, обслуживаемая несколькими стойками регистрации на рейс. Каждый оператор обслуживает одного пассажира (или группу пассажиров), а остальные ждут в очереди. Вновь прибывшие пассажиры становятся в *конец* очереди и ждут, пока не достигнут ее *начала*, откуда смогут подойти к первому освободившемуся оператору. При реализации подобной очереди нужно предусмотреть много тонких деталей: операторы не должны пытаться одновременно обслуживать одного и того же пассажира, требуется правильно обрабатывать пустые очереди, а в компьютерной системе должен быть способ определения максимального размера, или *границы* очереди. (Последнее требование не часто встретишь на стойках регистрации в аэропортах, но оно может быть весьма полезно в системах, где определено максимальное время ожидания задачи, подлежащей выполнению.) Реализации интерфейса `Queue` в `java.util.concurrent` решают эти проблемы за вас.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter13*.

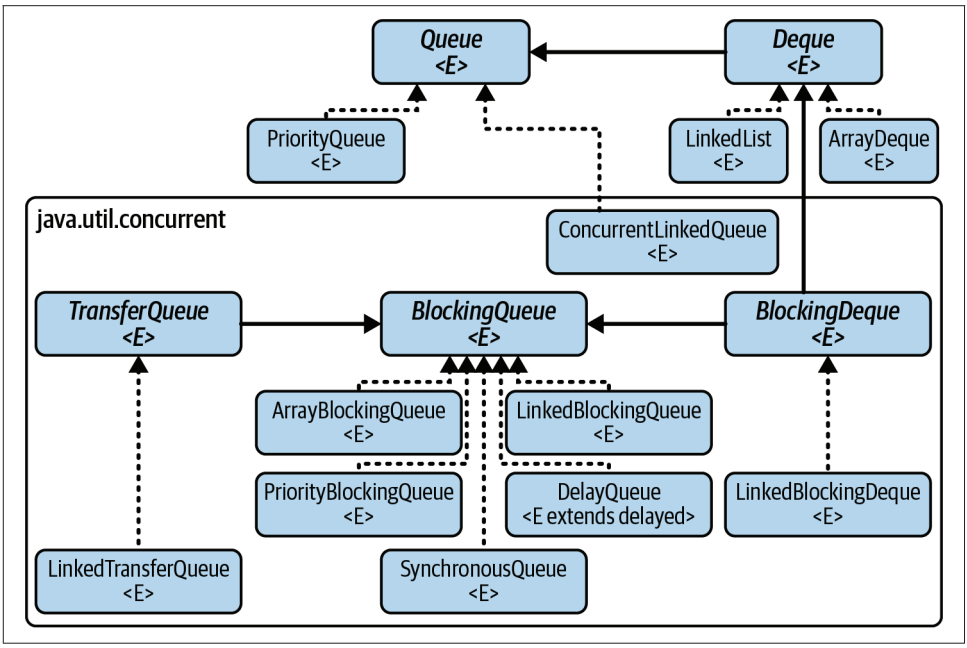


Рис. 13.1. Реализации Queue в каркасе коллекций

Методы интерфейса Queue

В дополнение к операциям, унаследованным от `Collection`, интерфейс `Queue` включает операции добавления в конец очереди, инспектирования элемента в ее начале и удаления начального элемента. Каждая из этих операций имеет два варианта: один для уведомления об ошибке возвращает значение `null`, а другой возбуждает исключение.

Добавление элемента в очередь

`boolean offer(E e)`	вставить заданный элемент, если возможно

Вариантом этой операции с исключением является метод `add`, унаследованный от `Collection`. Хотя `add` возвращает значение типа `boolean`, подтверждающее успешность вставки элемента, это значение нельзя использовать для извещения о том, что ограниченная очередь заполнена. Контракт `add` говорит, что он может возвращать `false`, только если элемент не вставлен, потому что уже присутствует в очереди; в противном случае обязан возбудить (неконтролируемое) исключение. Для ограниченной очереди вариант `offer`, возвращающий значение, обычно предпочтительнее.

Значение, возвращенное `offer`, показывает, был ли элемент успешно вставлен. Отметим, что `offer` все же возбуждает исключение, если элемент по какой-то причине недопустим (например, при попытке вставить `null` в очередь, не принимающую значений `null`). Обычно если `offer` вернул `false`, значит он был вызван для ограниченной очереди, достигшей предела своей емкости.

Извлечение элемента из очереди

В эту группу входят методы `peek` и `element` для инспектирования элемента в начале очереди и методы `poll` и `remove` для удаления его из очереди и возврата его значения.

Исключение в случае пустой очереди возбуждают следующие методы:

`E element()`	извлечь, но не удалять элемент в начале очереди
`E remove()`	извлечь и удалить элемент в начале очереди

Заметим, что этот метод отличается от метода `remove(Object)`, унаследованного от `Collection`.

Значение `null` в случае пустой очереди возвращают следующие методы:

`E peek()`	извлечь, но не удалять элемент в начале очереди
`E poll()`	извлечь и удалить элемент в начале очереди

Поскольку эти методы возвращают `null`, когда очередь пуста, не следует помещать в очередь элементы со значением `null`. Вообще говоря, помещать `null` в очередь запрещает интерфейс `Queue`; в JDK единственной реализацией, допускающей это, является унаследованный класс `LinkedList`.

Использование методов очереди

Рассмотрим несколько примеров использования этих методов. Очереди должны предоставлять удобный способ реализации менеджера задач, поскольку их основная цель – отдавать элементы, например задачи, для обработки. Пока что воспользуемся классом `ArrayDeque`, поскольку это самая быстрая и прямолинейная реализация `Queue` (и `Deque`). Как и раньше, мы ограничимся методами интерфейса, но помните, что, выбирая реализацию `Queue`, вы также выбираете порядок обработки задач. В случае `ArrayDeque` вы получаете порядок FIFO; это хороший выбор для менеджера задач, который в своей первоначальной версии не поддерживает назначения задачам приоритетов. (В последующей версии, в примере 13.1, мы воспользуемся очередью с приоритетами.)

Очередь `ArrayDeque` неограниченная, поэтому мы могли бы использовать как `add`, так и `offer` для заполнения ее новыми задачами:

```
Queue<Task> taskQueue = new ArrayDeque<>();
taskQueue.offer(mikePhone);
taskQueue.add(paulPhone);
```

Как только мы будем готовы выполнить какую-нибудь задачу, мы можем получить ее из начала очереди:

```
Task nextTask = taskQueue.poll();
if (nextTask != null) {
  // обработать nextTask
}
```

Выбор между методами `poll` и `remove` зависит от того, хотим ли мы рассматривать пустоту очереди как исключительное условие. На практике – с учетом

природы приложения – в этом есть смысл, поэтому ниже показан альтернативный подход:

```
try {
Task nextTask = taskQueue.remove();
  // обработать nextTask
} catch (NoSuchElementException e) {
 // но у нас задачи *никогда* не кончаются!
}
```

Эта схема нуждается уточнении, учитывающем природу различных видов задач. Для телефонных задач нужно сравнительно немного времени, тогда как приступать к кодированию, не имея достаточного запаса времени, мы не любим. Поэтому если срок ограничен, – скажем, задачу нужно довести до конца к следующей встрече, – то мы предпочли бы проверить, что очередная задача нужного вида, прежде чем выбирать ее из очереди:

```
Task nextTask = taskQueue.peek();
if (nextTask instanceof PhoneTask) {
  taskQueue.remove();
  // обработать nextTask
}
```

Эти методы инспектирования и удаления – главное достоинство интерфейса `Queue`; в `Collection` нет ничего подобного (хотя в `NavigableSet` есть). Цена, которую мы платим за это преимущество, состоит в том, что методы `Queue` полезны, только если начальный элемент содержит то, что нам нужно. Да, класс `PriorityQueue` позволяет предоставить компаратор, который будет упорядочивать элементы очереди, так чтобы нужный нам оказался в начале, но это может оказаться не самым лучшим способом выражения алгоритма выбора следующей задачи. Например, иногда бывает полезно знать обо *всех* ожидающих задачах, прежде чем выбрать следующую. Или, если время ограничено, то при работе с менеджером задач, основанным строго на очереди, у вас может не остаться другого выхода, как провести время за чашечкой кофе перед началом встречи. В качестве альтернативы можно было бы рассмотреть интерфейс `List`, предоставляющий более гибкие средства доступа к элементам, но у него есть существенный недостаток – гораздо более слабая поддержка многопоточного использования.

Возможно, это звучит излишне пессимистично; в конце концов, `Queue` – подынтерфейс `Collection`, а значит, наследует методы, поддерживающие обход, в частности `iterator`. Но хотя эти методы реализованы, использовать их в обычных ситуациях не рекомендуется. При проектировании классов очередей эффективность обхода была принесена в жертву быстроте методов `Queue`; кроме того, не гарантируется, что итераторы очередей возвращают их элементы в правильном порядке, а для некоторых конкурентных очередей они и вовсе дают ошибку при нормальных условиях (см. раздел «Реализации BlockingQueue» ниже в этой главе).

Реализации Queue

В этом разделе мы рассмотрим две прямые реализации `Queue` в JDK: `PriorityQueue` и `ConcurrentLinkedQueue`. Далее в разделе «BlockingQueue» мы рассмотрим интерфейс `BlockingQueue` и его реализации. Поведение этих классов сильно разли-

чается. Большинство из них потокобезопасны и предлагают средства *блокирования* (т. е. операции, которые ждут наступления подходящих условий для выполнения). Некоторые поддерживают упорядочение по приоритету, одна – `DelayQueue` – хранит элементы, пока не истечет время задержки, а еще одна – `SynchronousQueue` – предназначена исключительно для синхронизации. При выборе реализации `Queue` вы обычно обращаете больше внимания на эти функциональные различия, чем на производительность.

PriorityQueue

`PriorityQueue` – одна из двух не унаследованных реализаций `Queue` (отличных от `LinkedList`), которые не были спроектированы в основном для конкурентного использования (другая – `ArrayDeque`). Она не является потокобезопасной и не предоставляет возможности блокирования. Она отдает свои элементы для обработки в порядке, похожем на используемый в `NavigableSet`, – либо в естественном порядке элементов, если они реализуют интерфейс `Comparable`, либо в порядке, индуцированном компаратором, переданным конструктору `PriorityQueue`. Поэтому она могла бы стать альтернативным выбором (очевидным, в силу своего названия) для разработки менеджера задач, основанного на приоритетах, который был описан в разделе «NavigableSet» главы 12 с использованием интерфейса `NavigableSet`. Какую альтернативу выбрать, определяется вашим приложением: если ему нужно просматривать все множество ожидающих задач, берите `NavigableSet`, а если основное требование – эффективный доступ к следующей задаче, то подойдет `PriorityQueue`.

Выбор `PriorityQueue` позволяет пересмотреть упорядочение: поскольку этот класс допускает дубликаты, он не требует – в отличие от `NavigableSet`, – чтобы порядок был согласован с `equals`. Чтобы подчеркнуть этот момент, мы определим для нашего менеджера задач новый порядок, зависящий только от приоритетов. В противовес возможным ожиданиям `PriorityQueue` не дает гарантий относительно того, как представляются несколько элементов с одинаковым значением. Поэтому в нашем примере, если в очереди есть несколько задач с наивысшим приоритетом, то какой именно будет находиться в начале, предсказать невозможно.

Ниже перечислены конструкторы `PriorityQueue`:

`PriorityQueue()`	естественный порядок, начальная емкость по умолчанию
`PriorityQueue(Collection<? extends E> c)`	если `c` имеет тип `PriorityQueue` или `SortedSet`, то упорядочить в порядке `c`, иначе использовать естественный порядок на элементах `c`
`PriorityQueue(int initialCapacity)`	естественный порядок, заданная начальная емкость
`PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`	порядок, определяемый компаратором, заданная начальная емкость
`PriorityQueue(PriorityQueue<? extends E> c)`	порядок и элементы копируются из `c`
`PriorityQueue(SortedSet<? extends E> c)`	порядок и элементы копируются из `c`

Обратите внимание, как второй конструктор избегает проблему перегруженного конструктора `TreeSet`, которую мы обсуждали в разделе «TreeSet» главы 12. Применение порядка, заимствованного у переданной коллекции `PriorityQueue` или `SortedSet`, позволяет этому конструктору воспользоваться реализацией двоичной кучи, чтобы выполнить массовое копирование содержимого переданной коллекции, избегая необходимости строить кучу поэлементно. В случае, когда конструктор получает `PriorityQueue`, это работает, потому что источник уже содержит правильно организованную кучу. Если же конструктор получил `SortedSet`, то сначала он вызывает метод `SortedSet::toArray`, который порождает массив, уже являющийся правильно организованной кучей.

Мы можем использовать `PriorityQueue` для простой реализации менеджера задач с помощью класса `PriorityTask`, определенного в разделе «NavigableSet» главы 12, и нового компаратора, учитывающего только приоритеты задач.

org/jgcbook/chapter13/B_implementing_queue/Program_1

```
final int INITIAL_CAPACITY = 10;
Comparator<PriorityTask> priorityComp = Comparator.comparing(PriorityTask::p
riority);
Queue<PriorityTask> priorityQueue =
    new PriorityQueue<>(INITIAL_CAPACITY, priorityComp);
priorityQueue.add(new PriorityTask(mikePhone, Priority.MEDIUM));
priorityQueue.add(new PriorityTask(paulPhone, Priority.HIGH));
PriorityTask nextTask = priorityQueue.poll();
System.out.println(nextTask);
...
nextTask = priorityQueue.poll();
```

Очереди с приоритетами обычно эффективно реализуются с помощью *приоритетных куч*. Приоритетная куча – это двоичное дерево, напоминающее те, что мы видели при реализации `TreeSet` в разделе «TreeSet» главы 12, но с двумя отличиями. Во-первых, единственное ограничение на порядок заключается в том, что каждый узел дерева должен быть упорядочен относительно своих дочерних узлов: либо меньше в случае *минимальной кучи* (в Java она подразумевается по умолчанию для естественно упорядоченных элементов), либо больше в случае *максимальной кучи*. Во-вторых, дерево должно быть полным на всех уровнях, кроме, быть может, самого нижнего; если нижний уровень неполон, то составляющие его узлы должны быть сгруппированы в левой части. На рис. 13.2a изображена небольшая минимальная куча, в каждом узле которой показано только поле, содержащее приоритет. Чтобы добавить новый элемент в приоритетную кучу, он сначала помещается в самую левую свободную позицию – показанную более темным цветом на рис. 13.2b. Затем он повторно обменивается местами со своим родителем, пока не достигнет родителя с более высоким приоритетом, т. е. с меньшим значением приоритета. На рисунке для этого понадобился только один обмен нового элемента с родителем, после чего куча приняла вид, показанный на рис. 13.2c. (Узлы более темного цвета на рис. 13.2 и 13.3 только что поменялись позициями.)

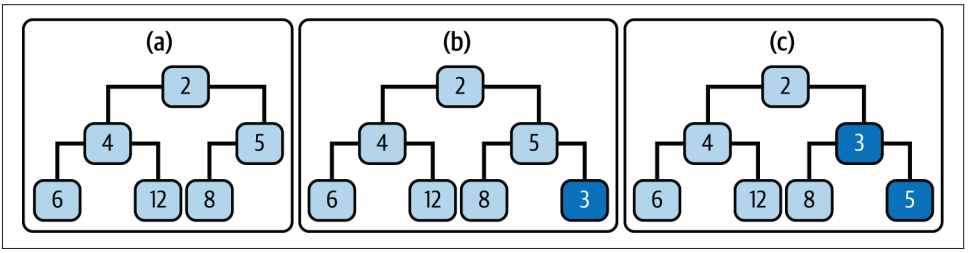


Рис. 13.2. Добавление элемента в `PriorityQueue`

Получить элемент с наивысшим приоритетом из приоритетной кучи тривиально: он находится в корне дерева. Но после его удаления два получившихся дерева должно быть объединены в приоритетную кучу. Для этого мы сначала помещаем самый правый элемент из нижней строки в позицию корня. Затем выполняется процедура, противоположная добавлению элементу, – мы повторно меняем его местами с меньшим из дочерних элементов, пока не окажется, что его приоритет выше, чем приоритеты обоих потомков, или пока он не станет листовым узлом. Весь процесс, начиная с кучи, образовавшейся на рис. 13.2c после удаления корня, показан на рис. 13.3 – снова понадобился только один обмен.

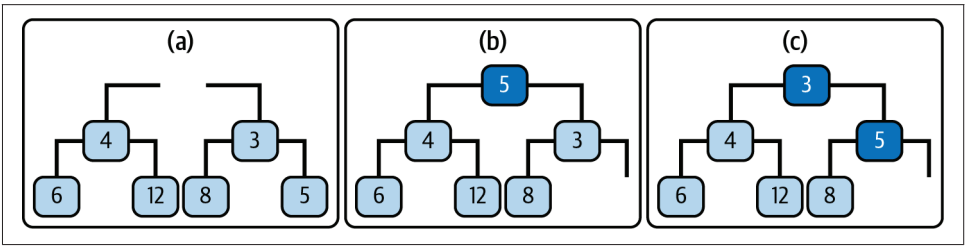


Рис. 13.3. Удаление элемента из начала `PriorityQueue`

Помимо постоянных накладных расходов, количество операций, потребных для добавления и удаления элементов, пропорционально высоте дерева. Поэтому временная сложность методов `offer`, `poll`, `remove` без параметров и `add` класса `PriorityQueue` равна $O(\log N)$. Методы `remove(Object)` и `contains` могут потребовать обхода всего дерева, поэтому их сложность равна $O(N)$. Методы `peek` и `element`, который просто получает корень дерева, не удаляя его, занимают постоянное время, как и метод `size`, которые читает постоянно обновляемое поле объекта.

Класс `PriorityQueue` не предназначен для конкурентного использования. Для этой цели каркас предлагает версию `PriorityBlockingQueue` (см. раздел «PriorityBlockingQueue» ниже в этой главе). Итераторы `PriorityQueue` обладают свойством быстрого отказа. Результаты обхода могут показаться неожиданными, пока не познакомишься с реализацией: элементы хранятся в массиве построчно слева направо. Итератор просто должен обойти массив, поэтому, как видно из рисунков, порядок предоставления им элемен-

тов не связан с порядком, индуцированным методом сравнения. Порядок, определяемый приоритетами, становится виден, только когда элементы удаляются по одному.

ConcurrentLinkedQueue

Как показано на рис. 13.1, большинство реализаций очереди блокирующие, и их мы рассмотрим в следующем разделе. Исключения составляют `PriorityQueue`, рассмотренная выше, и `ConcurrentLinkedQueue`. Это неограниченная, потокобезопасная очередь, обслуживаемая в порядке FIFO. В ней используется связанная структура, похожая на те, что мы видели в разделе «ConcurrentSkipListSet» главы 12, где она использовалась в качестве основы для списков с пропусками, и в разделе «HashSet» той же главы, где она нашла применение в цепочках переполнения хеш-таблиц. Там мы отметили, что одно из основных достоинств связанных структур, заключается в том, что операции вставки и удаления, реализованные посредством манипулирования указателями, требуют постоянного времени. Это делает их особенно полезными для реализации FIFO-очереди, где эти операции всегда применяются к элементам на концах структуры, которые не нужно искать с применением медленного последовательного поиска в списке.

В `ConcurrentLinkedQueue` используется основанный на команде CAS (см. раздел «Конкурентные коллекции» главы 9) алгоритм *без ожидания*, гарантирующий, что каждый поток не простаивает независимо от состояния других потоков, обращающихся к очереди. Он выполняет операции вставки и удаления из очереди за постоянное время, но требует линейного времени для выполнения `size`. Это связано с тем, что алгоритм, который опирается на кооперацию потоков для вставки и удаления, не хранит размер очереди и вынужден обходить всю очередь для его вычисления по запросу.

Класс `ConcurrentLinkedQueue` имеет два стандартных конструктора, о которых шла речь в разделе «Конструкторы коллекций» главы 10. Его итераторы слабо согласованы.

BlockingQueue

Классы в каркасе коллекций, предназначенные для поддержки конкурентных приложений, – в большинстве своем реализации подынтерфейса `BlockingQueue<E>` интерфейса `Queue`, и спроектированы они прежде всего для использования в сценариях производитель–потребитель.

Типичный пример использования очередей типа производитель–потребитель встречается в системах спулинга печати: клиентские процессы добавляют задания печати в очередь спулера, а затем они обрабатываются одним или несколькими процессами обслуживания печати, каждый из которых забирает задачу, находящуюся в начале очереди. Основные услуги, которые `BlockingQueue` предоставляет таким системам, – методы помещения в очередь и извлечения из очереди, которые не возвращают управление, пока не завершатся успешно. Так, в нашем пример сервер печати не обязан беспрерывно опрашивать очередь, чтобы узнать, есть ли в ней ожидающие задания; ему нужно только вызвать

метод `poll` с указанием тайм-аута, а система приостановит его выполнение до тех пор, пока в очереди не появится задача или не истечет тайм-аут.

Методы BlockingQueue

В интерфейсе `BlockingQueue` объявлено семь новых методов, которые можно отнести к трем группам.

Добавление элемента

`boolean offer (E e, long timeout, TimeUnit unit)`	вставить `e`, но ждать не дольше, чем заданный тайм-аут
`void put(E e)`	добавить `e`, ожидая столько, сколько понадобится

Методы, унаследованные от `Queue`, – `add` и `offer`, – немедленно завершаются с ошибкой, если вызваны для ограниченной очереди, которая уже заполнена; при этом `add` возбуждает исключение, а `offer` возвращает `false`. Описанные же здесь блокирующие методы более терпеливы: перегруженный вариант `offer` ждет в течение указанного количества единиц времени, определенных в перечислении `java.util.concurrent.TimeUnit` (это позволяет задавать тайм-аут, скажем, в секундах или миллисекундах), а `put` может находиться в состоянии ожидания неопределенно долго.

Удаление элемента

`E poll(long timeout, TimeUnit unit)`	извлечь и удалить элемент, находящийся в начале очереди, но ждать не дольше, чем заданный тайм-аут
`E take()`	извлечь и удалить элемент, находящийся в начале очереди, ожидая столько, сколько понадобится

Вместе с унаследованными от `Queue` методы удаления элемента демонстрируют те же четыре поведения в случае ошибки, что и методы добавления элемента. Как правило, ошибка вызвана тем, что очередь в данный момент не содержит ни одного элемента. В этой ситуации унаследованный метод `poll` возвращает `null`, тогда как блокирующий метод `poll` возвращает `null` только в случае истечения тайм-аута; унаследованный метод `remove` возбуждает исключение немедленно, а `take` блокирует выполнение, пока не завершится успешно.

Хотя поведение различных методов `BlockingQueue` изменяется систематически, разобраться в их названиях может быть нелегко. Наиболее четкую картину дает документация Java (табл. 13.1).

Таблица 13.1. Методы BlockingQueue

	Возбуждает исключение	Специальное значение	Блокирует	Тайм-аут
Вставка	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Удаление	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Осмотр	`element()`	`peek()`	Неприменимо	Неприменимо

Извлечение из очереди или опрос ее содержимого

`int drainTo(Collection<? super E> c)`	очистить очередь, переместив все элементы в `c`
`int drainTo(Collection<? super E> c, int maxElements)`	очистить не более заданного числа элементов, переместив их в `c`
`int remainingCapacity()`	вернуть число элементов, которые очередь могла бы принять без блокировки, или `Integer.MAX_VALUE`, если очередь неограниченная

Методы `drainTo` выполняются атомарно и эффективно, поэтому второй перегруженный вариант полезен в ситуациях, когда вы точно знаете, что обрабатывающих мощностей достаточно для немедленной обработки определенного числа элементов, а первый – когда, например, все потоки-производители перестали работать. Эти методы возвращают количество переданных элементов. Метод `remainingCapacity` возвращает количество свободного места в очереди, хотя, как всегда в многопоточных контекстах, его результат не следует рассматривать как часть последовательности «проверь, потом действуй», потому что между проверкой (вызов `remainingCapacity`) и действием (добавление элемента в очередь) в одном потоке другой поток мог бы вмешаться и добавить иди удалить элементы.

`BlockingQueue` гарантирует, что операции во всех его реализациях будут потокобезопасными и атомарными. Но эта гарантия не распространяется на массовые операции, унаследованные от `Collection`, – `addAll`, `containsAll`, `retainAll` и `removeAll`, – если только ее не дает конкретная реализация. Так, например, `addAll` вполне может возбудить исключение, добавив в коллекцию только часть заказанных элементов.

Использование методов BlockingQueue

Менеджер задач, обслуживающий только одного пользователя в каждый момент времени, крайне ограничен; нам необходимо кооперативное решение – которое позволило бы конкурентно порождать и обрабатывать задачи. На рис. 13.1 показан класс `StoppableTaskQueue`, простая версия конкурентного менеджера задач, основанная на `PriorityBlockingQueue`, которая позволяет пользователям (нам) независимо добавлять задачи в очередь, когда мы чувствуем в этом необходимость, а также забирать их для обработки, когда у нас появится время. В классе `StoppableTaskQueue` определены методы `addTasks`, `getFirstTask`, `shutDown` и вспомогательный метод `addTask`. Объект `StoppableTaskQueue` либо работает, либо остановлен. Метод `addTasks` возвращает булево значение, показывающее, были ли задачи добавлены успешно; это значение равно `true`, если только `StoppableTaskQueue` не остановлен. Метод `getFirstTask` возвращает задачу, находящуюся в начале очереди. Если задач нет, он не блокирует выполнение, а возвращает `null`. Метод `shutDown` останавливает `StoppableTaskQueue`, ждет завершения всех выполняющихся операций `addTasks`, а затем опустошает `StoppableTaskQueue` и возвращает ее содержимое.

Пример 13.1. *Конкурентный менеджер задач на основе очереди*

org/jgcbook/chapter13/C_blocking_queue/StoppableTaskQueue

```
public class StoppableTaskQueue {
  private final int MAXIMUM_PENDING_OFFERS = Integer.MAX_VALUE;
       private    final    BlockingQueue<PriorityTask>    taskQueue    =    new
PriorityBlockingQueue<>();
  private final Semaphore semaphore = new Semaphore(MAXIMUM_PENDING_OFFERS);
  private volatile boolean isStopping;

  // вернуть true, если задача была успешно помещена в очередь, или false, если
  // очередь находится в процессе остановки
  public boolean addTask(PriorityTask task) {
    return addTasks(List.of(task));
  }

  // вернуть true, если задачи в переданной коллекции были успешно помещены в
  // очередь, или false, если очередь находится в процессе остановки
  public boolean addTasks(Collection<PriorityTask> tasks) {
    if (isStopping) return false;
    if (! semaphore.tryAcquire()) {
      return false;
    } else {
      taskQueue.addAll(tasks);
      semaphore.release();
      return true;
    }
  }

  // вернуть задачу, находящуюся в начале очереди, или null, если задач нет
  public PriorityTask getFirstTask() {
    return taskQueue.poll();
  }

  // остановить очередь, дождаться завершения всех производителей, затем
  // вернуть ее содержимое
  public Collection<PriorityTask> shutDown() {
    isStopping = true;
    // блокирует выполнение до тех пор, пока все находящиеся в процессе
    // выполнения вызовы addTasks() не завершатся
    semaphore.acquireUninterruptibly(MAXIMUM_PENDING_OFFERS);
    List<PriorityTask> returnCollection = new ArrayList<>();
    taskQueue.drainTo(returnCollection);
    return returnCollection;
  }
}
```

В этом примере, как и в большинстве коллекций из пакета `java.util.concurrent`, коллекция сама решает вопросы, возникающие из-за взаимодействия различных потоков при добавлении или удалении элементов из очереди. А большая часть кода в примере 13.1 занята решением другой задачи: механизмом упорядоченного завершения работы. Причина такого внимания к этому моменту заключается в том, что, когда мы захотим использовать класс `StoppableTaskQueue` в качестве компонента более крупной системы, понадобится возможность останавливать задачи без потери информации.

Организация упорядоченного завершения работы часто оказывается проблемой в конкурентных системах; дополнительные сведения см. в главе 7 книги Goetz et al. (2006).

Более крупная система будет моделировать задачи, запланированные на каждый день будущего года, позволяя потребителям обрабатывать задачи из очереди на день. В примере из этого раздела неявно предполагается, что если не осталось задач, запланированных на текущий день, то потребитель не станет ждать появления задачи, а сразу перейдет к очереди задач на следующий день. (В действительности мы в этом случае скорее пойдем домой или, что более вероятно, отметим нежданный праздник.) Это предположение упрощает пример, поскольку нам не нужно вызывать блокирующие методы `PriorityBlockingQueue`, хотя мы все же пользуемся одним методом, `drainTo`, интерфейса `BlockingQueue`.

Есть несколько подобных способов остановить очередь типа производитель–потребитель; в выбранном нами менеджер предоставляет метод `shutdown`, который может быть вызван потоком «супервизора», чтобы прекратить запись в очередь производителями, затем опустошить ее и вернуть невыполненные задачи. Трудность в том, чтобы обеспечить атомарное выполнение метода `addtasks,` – он должен либо добавить все переданные ему задачи, либо ни одной, – а также в том, чтобы гарантировать, что после того, как метод `shutdown` остановил очередь и занялся ее опустошением или уже опустошил, никакие потоки не могут добавить в очередь новые задачи.

В примере 13.1 это достигается с помощью *семафора* – потокобезопасного объекта, который хранит фиксированное число *допусков*. Обычно семафоры применяются для регулирования доступа к конечному множеству объектов, например, пулу подключений к базе данных. Допуски, имеющиеся у семафора в каждый момент времени, представляют ресурсы, которые в данный момент не используются. Поток, нуждающийся в ресурсе, получает допуск от семафора и возвращает его семафору вместе с освобождением ресурса. Если все ресурсы заняты, то у семафора нет свободных допусков; в этот момент поток, пытающийся получить допуск, будет заблокирован, пока какой-нибудь другой поток не вернет свой допуск.

В этом примере семафор используется иначе. Мы не хотим препятствовать потокам-производителям записывать в очередь – ведь это конкурентная очередь, вполне способная справиться с многопоточным доступом и без нашей помощи. Мы только хотим, чтобы метод `shutdown` мог сказать, есть ли в данный момент незавершенные операции записи. Поэтому мы создаем семафор с максимально возможным количеством допусков, которые на практике никогда не будут востребованы одновременно. Метод производителя `addTasks` сначала проверяет волатильный булев флаг `isStopping`, чтобы убедиться, что процедура завершения работы еще не начата, а затем вызывает метод семафора `tryAcquire`, который немедленно возвращает `false`, если свободных допусков нет (это означает, что процедура остановки выполняется или уже закончилась). Если `tryAcquire` вернула `true`, то `addTasks` добавляет свои задачи в очередь, а затем освобождает допуск.

Метод `shutdown` сначала устанавливает флаг `isStopping`, чтобы не могло начаться новое выполнение метода `addTasks`. Затем он должен дождаться возврата всех

ранее выданных допусков. Для этого он вызывает метод `acquireUninterruptibly`, указывая, что нужны *все* допуски; этот вызов блокирует выполнение, пока все допуски не будут освобождены потоками-производителями, после чего процедуру остановки можно завершить, вернув необработанные задачи.

Реализации BlockingQueue

Каркас коллекций предлагает пять реализаций интерфейса `BlockingQueue`.

LinkedBlockingQueue

Эта реализация – потокобезопасная FIFO-очередь на основе связанной структуры. Оба стандартных конструктора создают потокобезопасную блокирующую очередь с емкостью `Integer.MAX_VALUE`. Меньшую емкость можно задать с помощью третьего конструктора:

```
LinkedBlockingQueue(int capacity)
```

`LinkedBlockingQueue` – FIFO-очередь. Вставка и удаление элементов выполняются за постоянное время; у таких операций, как `contains`, требующих обхода массива, сложность линейная. Итераторы слабо согласованы.

ArrayBlockingQueue

Эта реализация основана на *циклическом массиве* – линейной структуре, в которой первый и последний элементы логически являются соседними. Идея показана на рис. 13.4. Позиция «head» обозначает начало очереди; всякий раз, как элемент удаляется из очереди, индекс начала продвигается вперед. Аналогично новые элементы добавляются в конец очереди (обозначен «tail»), что приводит к продвижению соответствующего индекса. Если любой из индексов переходит через правую границу массива, то ему присваивается значение 0. Если оба индекса одинаковы, значит, очередь либо пуста, либо заполнена, и, чтобы различить эти два случая, реализация должна отдельно хранить счетчик элементов в очереди.

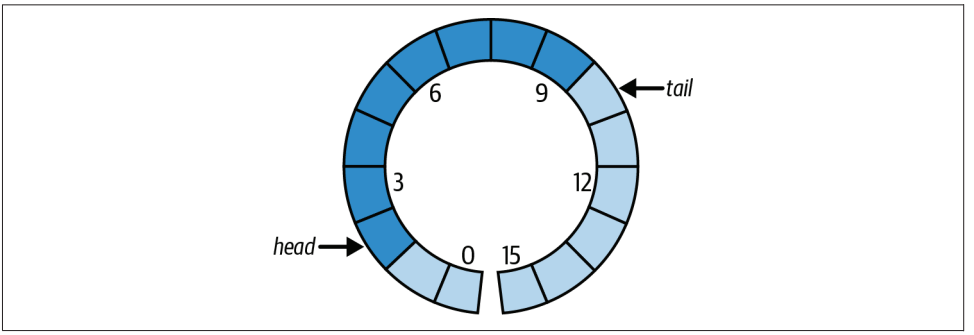


Рис. 13.4. Циклический массив

Конструкторы классов коллекций на базе массива обычно имеют всего один конфигурационный параметр – начальную длину массива. Для классов фиксированного размера, в частности `ArrayBlockingQueue`, этот параметр обязате-

лен, так как определяет емкость коллекции. (Для классов переменного размера, например `ArrayList`, можно использовать начальную емкость по умолчанию, поэтому имеются конструкторы, которым емкость не передается.) Для `ArrayBlockingQueue` такие конструкторы имеют вид:

```
ArrayBlockingQueue(int capacity)
ArrayBlockingQueue(int capacity, boolean fair)
ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c)
```

Параметр типа `Collection` последнего конструктора позволяет инициализировать `ArrayBlockingQueue` элементами заданной коллекции, которые обходятся в порядке, определяемом ее итератором. Заданная в этом конструкторе емкость должна быть не меньше размера указанной коллекции, а если эта коллекция пуста, то емкость должна быть как минимум 1. Помимо параметра, конфигурирующего базовый массив, последние два конструктора требуют также аргумента типа `boolean`, управляющего тем, как очередь обрабатывает несколько блокированных запросов. Такое бывает, когда несколько потоков пытаются удалить элементы из пустой очереди или добавить новые в полную. Когда очередь сможет обслужить один из таких запросов, какой именно она должна выбрать? Альтернативы таковы: выбирать запрос, который ждет дольше всех (такая политика планирования называется *справедливой*), или позволить реализации выбрать запрос по своему усмотрению. Справедливая политика кажется лучшей альтернативой, потому что гарантирует, что неудачливый поток не будет задержан на неопределенно долгое время, но на практике преимущества, которые она сулит, редко бывают настолько важны, чтобы оправдать значительные накладные расходы, сопровождающие операции очереди. Если справедливое планирование не задано, то `ArrayBlockingQueue` обычно аппроксимирует справедливое обслуживание, но без гарантий.

`ArrayBlockingQueue` обслуживает запросы в порядке FIFO. Вставка и удаление элементов выполняются за постоянное время; у таких операций, как `contains`, требующих обхода массива, сложность линейная. Итераторы слабо согласованы.

PriorityBlockingQueue

Эта реализация является потокобезопасной блокирующей версией `PriorityQueue` (см. раздел «PriorityQueue» выше) со схожими характеристиками упорядочения и производительности. Ее итераторы обладают свойством быстрого отказа, поэтому возбуждают исключение `ConcurrentModificationException` при многопоточном доступе; успешно они работают, только если к очереди осуществляет доступ только один поток. Для безопасного обхода `PriorityBlockingQueue` переместите ее элементы в массив и обходите его.

DelayQueue

Это специализированная очередь с приоритетами, упорядочение которой основано на *времени задержки* каждого элемента – времени до того момента, когда элемент можно будет забрать из очереди. Если у всех элементов время задержки положительно, т. е. ни один тайм-аут еще не истек, то метод очереди `poll` вернет `null`. Если у одного или нескольких элементов время задержки истекло, то тот, у которого это время было наибольшим, окажется в начале очереди. Элементы `DelayQueue` принадлежат классу, реализующему интерфейс `java.util.concurrent.Delayed`:

```
interface Delayed extends Comparable<Delayed> {
  long getDelay(TimeUnit unit);
}
```

Метод `getDelay` объекта типа `Delayed` возвращает оставшееся время задержки. (Этот интерфейс появился раньше, чем API `java.time`, но теперь очевидным улучшением было бы добавить метод по умолчанию `getDelay`, возвращающий значение типа `Duration`.) Метод `compareTo` интерфейса `Comparable` (см. раздел «Интерфейс Comparable» главы 3) должен быть определен так, чтобы его результаты были согласованы с задержками сравниваемых объектов. Это означает, что он редко бывает совместим с `equals`, так что объекты `Delayed` не годятся для использования совместно с реализациями `SortedSet` и `SortedMap`.

Например, в нашем менеджере задач нам, вероятно, понадобятся задачи-напоминания, следящие за тем, чтобы сообщения электронной почты и телефонные звонки не оставались без ответа. Мы могли бы определить новый класс `DelayedTask`, как показано в примере 13.2, и использовать его для реализации очереди напоминаний.

org/jgcbook/chapter13/C_blocking_queue/Program_1

```
BlockingQueue<DelayedTask> reminderQueue = new DelayQueue<DelayedTask>();
reminderQueue.offer(new DelayedTask(databaseCode, 1));
reminderQueue.offer(new DelayedTask(guiCode, 2));
...
// теперь получить первую задачу-напоминание, готовую к обработке
DelayedTask t1 = reminderQueue.poll();
if (t1 == null) {
  // готовых напоминаний еще нет
} else {
  // обработать t1
}
```

Пример 13.2. *Класс* `DelayedTask`

org/jgcbook/chapter13/C_blocking_queue/DelayedTask

```
public class DelayedTask implements Delayed {

  private final LocalDateTime endTime;
  private final Task task;
  private static Comparator<Delayed> dtComparator =
      Comparator.comparing(dt -> dt.getDelay(TimeUnit.MILLISECONDS));

  public DelayedTask(Task t, int daysDelay) {
    task = t;
    endTime = LocalDateTime.now().plusDays(daysDelay);
  }

  @Override
  public long getDelay(TimeUnit unit) {
    return unit.convert(Duration.between(LocalDateTime.now(), endTime));
  }

  @Override
  public int compareTo(Delayed d) {
    return dtComparator.compare(this, d);
  }
```

```
    public Task getTask() {
      return task;
    }
  }
```

Большинство операций очереди знают о задержках и обращаются с очередью, в которой нет элементов с истекшими тайм-аутами, как с пустой. Исключения составляют методы `peek` и `remove`, которые, быть может неожиданно, позволяют просматривать элемент в начале очереди `DelayQueue`, даже если его задержка еще не истекла. И, подобно им, но в отличие от других методов `Queue`, операции интерфейса `Collection` применительно к `DelayQueue` не обращают внимания на задержки. Например, ниже приведено два способа скопировать элементы `reminderQueue` в множество:

```
Set<DelayedTask> delayedTaskSet1 = new HashSet<DelayedTask>();
delayedTaskSet1.addAll(reminderQueue);

Set<DelayedTask> delayedTaskSet2 = new HashSet<DelayedTask>();
reminderQueue.drainTo(delayedTaskSet2);
```

Множество `delayedTaskSet1` будет содержать все напоминания в очереди, а множество `delayedTaskSet2` – только те, что готовы к использованию.

Характеристики производительности `DelayQueue` такие же, как у `PriorityQueue`, на которой она основана, и, подобно `PriorityQueue`, ее итераторы обладают свойством быстрого отказа. Замечания по поводу итераторов `PriorityBlockingQueue` (см. раздел «PriorityBlockingQueue» выше) применимы и к итераторам `DelayQueue`.

SynchronousQueue

На первый взгляд кажется, что в очереди, не имеющей внутренней емкости, смысла не много, а именно так можно кратко описать класс `SynchronousQueue`. Но на самом деле она может оказаться очень полезной; поток, который хочет добавить элемент в `SynchronousQueue`, должен подождать, пока не появится другой поток, готовый тут же забрать элемент, и то же самое – только наоборот – верно для потока, который хочет забрать элемент из очереди. Поэтому `SynchronousQueue` выполняет функцию, на которую намекает ее название: *рандеву* – механизм синхронизации двух потоков. (Не путайте эту идею синхронизации потоков, что позволяет им работать сообща и обмениваться данными, с ключевым словом Java `synchronized`, которое предотвращает одновременное выполнение кода разными потоками.) В классе `SynchronousQueue` определено два конструктора:

```
SynchronousQueue()
SynchronousQueue(boolean fair)
```

Класс `SynchronousQueue` обычно находит применение в системах организации совместной работы, когда дизайн обеспечивает наличие достаточного числа потоков-потребителей, чтобы потоки-производители могли поставлять задачи без необходимости ждать. В такой ситуации он позволяет безопасно передавать данные задач между потоками без свойственных `BlockingQueue` накладных расходов на добавление в очередь и последующее извлечение из нее при передаче каждой задачи.

С точки зрения методов `Collection` `SynchronousQueue` ведет себя как пустая коллекция; методы `Queue` и `BlockingQueue` ведут себя так, как и должны для очереди нулевой емкости, которая всегда пуста. Метод `iterator` возвращает пустой итератор, метод `hasNext` которого всегда возвращает `false`.

TransferQueue

Интерфейс `TransferQueue<E>` предлагает производителям способ выбрать между синхронной и асинхронной постановкой данных в очередь. Это полезно, например, в системах обмена сообщениями, допускающих как синхронные, так и асинхронные сообщения. Будучи расширением `BlockingQueue`, он предоставляет систему, обладающую способностью притормаживать порождение данных блокирующими производителями. (Эта возможность на самом деле не используется в единственной платформенной реализации, `LinkedTransferQueue`, которая всегда неограниченна; в этом классе `put` всегда реализует асинхронную постановку в очередь.) Однако в интерфейсе объявлен дополнительный метод, `transfer`, который производитель может вызвать, если хочет блокировать выполнение, пока помещенный в очередь элемент не будет извлечен каким-то потребителем, – это вариация на тему синхронного обмена данными, похожего на предоставляемый классом `SynchronousQueue`. Метод `transfer` имеет три перегруженных варианта.

`void transfer(E e)`	передать элемент потребителю, ожидая столько времени, сколько понадобится
`boolean tryTransfer(E e)`	передать элемент потребителю, если это возможно немедленно
`boolean tryTransfer(E e, long timeout, TimeUnit unit)`	передать элемент потребителю, ожидая до истечения тайм-аута

В интерфейсе также объявлено два вспомогательных метода, предоставляющих грубую оценку счетчика ожидающих потребителей.

`boolean hasWaitingConsumer()`	вернуть `true`, если есть хотя бы один ожидающий потребитель
`int getWaitingConsumerCount()`	вернуть оценку числа ожидающих потребителей

Как и все методы, обобщающие картину состояния конкурентной коллекции, эти дают в лучшем случае моментальный снимок, который может измениться к моменту обработки результатов.

JDK предоставляет одну реализацию `TransferQueue` – класс `LinkedTransferQueue`. Это неограниченная FIFO-очередь, обладающая некоторыми интересными свойствами: она безблокировочная, как `ConcurrentLinkedQueue` но имеет блокирующие методы, которые у последней отсутствуют; она поддерживает методы `transfer` реализуемого интерфейса с помощью «двойственной очереди», узлы которой могут представлять либо находящиеся в очереди данные, либо еще не выполненные запросы на извлечение из очереди. И, что необычно для конкурентных классов, она обеспечечивает справедливость без снижения производительности. На самом деле она превосходит по производительности даже «несправедливый» режим `SynchronousQueue`.

Добавление в очередь и извлечение из нее имеют сложность *O*(1). Итератор слабо согласован.

Deque

Дек – это двусторонняя очередь. В отличие от очереди, в которую элементы добавляются только в конец, а удаляются только из начала, дек принимает и отдает (для просмотра или удаления) элементы с обеих сторон. Кроме того, в отличие от `Queue`, контракт интерфейса `Deque<E>` оговаривает порядок представления элементов: это линейная структура, в которой элементы, добавленные в конец, отдаются в том же порядке из начала. Таким образом, будучи использован в качестве очереди, дек всегда является FIFO-структурой; контракт не допускает, например, существования деков с приоритетом. Если элементы удаляются с той же стороны (начала или конца), с которой добавлялись, то дек ведет себя как стек или LIFO-структура с дисциплиной обслуживания «последним пришел, первым ушел».

В быстрой реализации `ArrayDeque` интерфейса `Deque` используется циклический массив (см. раздел «Реализации BlockingQueue» выше); это предпочтительная реализация для стеков и очередей. Конкурентные деки играют особую роль в распараллеливании, которая обсуждается в разделе «BlockingDeque» ниже).

Методы Deque

Интерфейс `Deque` расширяет `Queue` методами, в которых начало и конец равноправны. Для ясности методам `Queue`, которые неявно ссылаются на одну из сторон очереди, сопоставляется синоним, делающий их поведение явным. Например, методы `peek` и `offer`, унаследованные от `Queue`, эквивалентны методам `peekFirst` и `offerLast`. (Слова «First» и «Last» относятся соответственно к началу и концу очереди; в документации `Deque` встречаются также слова «front» и «end».)

Методы, подобные унаследованным от Collection

`void addFirst(E e)`	вставить `e` в начало, если места достаточно
`void addLast(E e)`	вставить `e` в конец, если места достаточно
`void push(E e)`	вставить `e` в начало, если места достаточно
`boolean removeFirstOccurrence(Object o)`	удалить первое вхождение `o`
`boolean removeLastOccurrence(Object o)`	удалить последнее вхождение `o`
`Iterator<E> descendingIterator()`	получить итератор, возвращающий элементы дека в обратном порядке

Контракты методов `addFirst` и `addLast` похожи на контракт метода `add` интерфейса `Collection`, но дополнительно оговаривают, куда именно следует поместить добавляемый элемент и что если элемент не удалось добавить, то должно быть возбуждено исключение `IllegalStateException`. Как и в случае ограниченных очередей, пользователи ограниченных деков должны избегать этих методов, отдавая предпочтение методам `offerFirst` и `offerLast`, которые могут сообщить о «нормальной» ошибке с помощью возврата булева значения.

Имя метода `push` – синоним `addFirst`, удобный, когда `Deque` используется как стек. Методы `removeFirstOccurrence` и `removeLastOccurrence` – аналоги `Collection::remove`, но дополнительно указывают, какое вхождение элемента следует удалить. Возвращенное значение показывает, был ли элемент удален в результате вызова.

Методы, подобные унаследованным от Queue

`Boolean offerFirst(E e)`	вставить `e` в начало, если в деке есть место
`boolean offerLast(E e)`	вставить `e` в конец, если в деке есть место

Имя метода `offerLast` – синоним эквивалентного метода `offer` интерфейса `Queue`.

Следующие методы возвращают `null` для пустого дека:

`E peekFirst()`	извлечь первый элемент, но не удалять его
`E peekLast()`	извлечь последний элемент, но не удалять его
`E pollFirst()`	извлечь и удалить первый элемент
`E pollLast()`	извлечь и удалить последний элемент

Методы `peekFirst` и `pollFirst` – синонимы эквивалентных методов `peek` и `poll` интерфейса `Queue`.

Следующие методы возбуждают исключение для пустого дека:

`E getFirst()`	извлечь первый элемент, но не удалять его
`E getLast()`	извлечь последний элемент, но не удалять его
`E removeFirst()`	извлечь и удалить первый элемент
`E removeLast()`	извлечь и удалить последний элемент
`E pop()`	извлечь и удалить первый элемент

Методы `getFirst` и `removeFirst` – синонимы эквивалентных методов `element` и `remove` интерфейса `Queue`. Метод `pop` – синоним `removeFirst`, удобный, когда `Deque` используется как стек.

Методы, унаследованные от SequencedCollection

Из семи методов `SequencedCollection` шесть на самом деле введены для нужд `Deque`. А оставшийся – единственный, предоставляющий представление `Deque`, – это метод `reversed`, ковариантное переопределение метода `SequencedCollection`, возвращающее `Deque`:

`Deque<E> reversed()`	вернуть представление `Deque` в обратном порядке

Реализации Deque

`Deque` – это двусторонняя очередь, которая может принимать и отдавать элементы с любой стороны. `Deque`, как и `Queue`, можно использовать в качестве канала для передачи информации между потоком-производителем и потоком-

потребителем. Его преимущество в этом контексте – способность реализовать заимствование работ, технику балансировки нагрузки, при которой простаивающие потоки заимствуют задачи из конца очередей работ занятых потоков с целью максимизировать степень параллелизма. В разделе «BlockingDeque» ниже эта техника обсуждается подробнее.

ArrayDeque

Появлению интерфейса `Deque` сопутствовала весьма эффективная реализация `ArrayDeque` на основе циклического массива, похожая на `ArrayBlockingQueue` (см. раздел «Реализации BlockingQueue» выше). Она заполняет разрыв в классах, реализующих `Queue`; раньше если вы хотели использовать FIFO-очередь в однопоточной среде, то должны были либо обратиться к классу `LinkedList` (который мы ниже рассмотрим, но которого следует избегать в качестве реализации очереди общего назначения), либо нести накладные расходы на ненужную вам потокобезопасность, сопряженные с использованием классов `ArrayBlockingQueue` или `LinkedBlockingQueue`. Что же касается `ArrayDeque`, то это реализация общего назначения, пригодная как для деков, так и для FIFO-очередей. Она имеет такие же характеристики производительности, как циклический массив: добавление или удаление элемента в начало или в конец очереди занимает постоянное время. Итераторы обладают свойством быстрого отказа.

Циклический массив, начало и конец которого могут непрерывно перемещаться вперед, лучше подходит для реализации дека, чем нециклический (например, стандартная реализация `ArrayList`, которую мы рассмотрим в разделе «Реализации интерфейса List» главы 14), когда удаление элемента из начала очереди требует изменения позиций всех остальных элементов, чтобы новый начальный элемент оказался в позиции 0. Заметим, впрочем, что за постоянное время можно вставить элементы только в начало и в конец дека. Если требуется удалить элемент из середины, что для деков позволяет сделать метод `Iterator::remove`, то все элементы с одной стороны следует сдвинуть, чтобы сохранить компактное представление. В результате вставка и удаление элементов из середины дека имеют сложность $O(N)$. На рис. 13.5 показано удаление элемента с индексом 6 из дека.

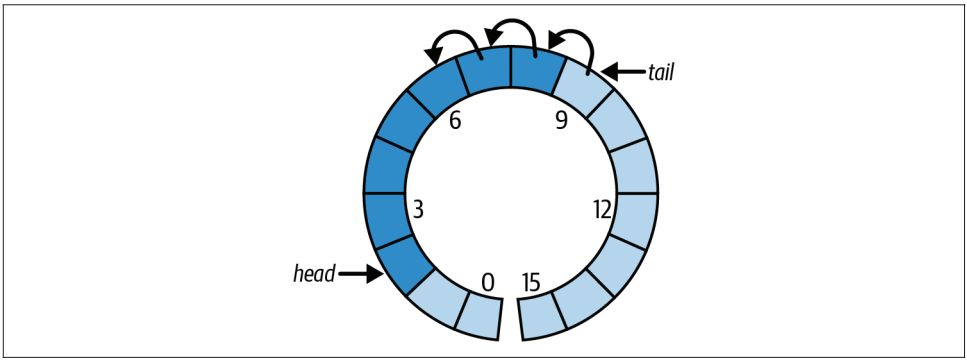


Рис. 13.5. Удаление элемента из циклического массива

LinkedList

Среди реализаций `Deque` класс `LinkedList` представляет собой диковинку; например, только он допускает элементы `null`, которые запрещены интерфейсом `Queue` из-за общепринятого использования `null` в качестве специального значения. Он присутствовал в каркасе коллекций с самого начала – первоначально как одна из стандартных реализаций интерфейса `List` (см. раздел «Реализации интерфейса List» главы 14). Но затем он был пополнен методами интерфейса `Queue`, когда последний был включен в каркас, а еще позже – методами `Deque`. В его основе лежит структура связанного списка, похожая на ту, что мы видели в разделе «ConcurrentSkipListSet» главы 12 при рассмотрении списков с пропусками, но с дополнительным полем в каждом узле, которое указывает на предыдущий элемент (см. рис. 13.6). Эти указатели позволяют проходить список в обратном направлении – например, для обратного итерирования – или удалять элемент из конца списка.

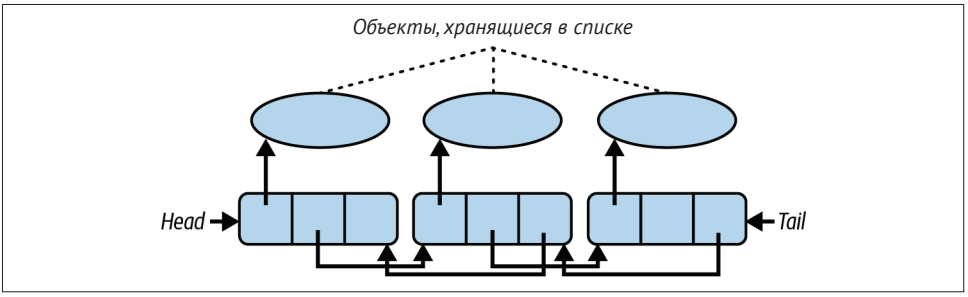


Рис. 13.6. Дважды связанный список

В качестве реализации `Deque` класс `LinkedList` не пользуется популярностью. Его основное преимущество, добавление в очередь и удаление из нее за постоянное время, оспаривается и для очередей, и для деков классом `ArrayDeque`, который во всех остальных отношениях превосходит `LinkedList`. Единственная причина использовать `LinkedList` в качестве очереди или дека проявляется, когда нужно добавлять и удалять элементы не только с обеих сторон, но и в середине списка – но это весьма необычное требование. Но даже это в `LinkedList` обходится дорого; чтобы добраться до нужного элемента, необходим линейный проход с временной сложностью $O(N)$. В разделе «Избегайте LinkedList» главы 17 объясняется, почему этого класса в общем случае лучше избегать.

Единственный метод `LinkedList`, не унаследованный от двух его интерфейсов, `Deque` и `List`, – ковариантное переопределение `reversed`:

`LinkedList<E> reversed()`	вернуть представление `LinkedList` в обратном порядке

Класс `LinkedList` имеет только стандартные конструкторы, описанные в разделе «Конструкторы коллекций» главы 10. Его итераторы обладают свойством быстрого отказа.

BlockingDeque

В разделе «BlockingQueue» выше мы видели, что `BlockingQueue` добавляет четыре метода сверх интерфейса `Queue`, которые позволяют удалять или добавлять элемент в очереди, ожидая неопределенно долго или в течение заданного тайм-аута. `BlockingDeque` предоставляет по два новых метода для каждого из этих четырех, чтобы операцию можно было выполнить в начале или в конце дека. Так, например, вдобавок к методу `BlockingQueue`

`void put(E e)`	добавить `e` в `Queue`, ожидая столько, сколько понадобится

`BlockingDeque` добавляет:

`void putFirst(E e)`	добавить `e` в начало `Deque`, ожидая столько, сколько понадобится
`void putLast(E e)`	добавить `e` в конец `Deque`, ожидая столько, сколько понадобится

Очевидно, что `put` и `putLast` – синонимы, равно как `take` и `takeFirst` и т. д.; имена продублированы, чтобы было понятно, как методы используются Аналогичные новые методы определены для каждого из трех других методов `BlockingQueue`: `offer`, `poll` и `take`.

Хорошие алгоритмы балансировки нагрузки приобрели особую важность с тех пор, как многоядерные и многопроцессорные архитектуры стали стандартом. Конкурентные деки легли в основу одного из лучших методов балансировки нагрузки, *заимствования работ*. Чтобы понять, как он работает, представьте себе алгоритм балансировки нагрузки, который распределяет задачи – например, циклически – между рядом очередей, и с каждой из них связан поток-потребитель, который забирает задачу из начала очереди, обрабатывает ее и возвращается за следующей. Хотя эта схема дает ускорение благодаря распараллеливанию, у нее есть существенный недостаток: можно представить себе две очереди, одна из которых содержит журнал долгих задач, с которыми напряженно стремится совладать поток-потребитель, а другая пуста, и простаивающий потребитель томится в ожидании работы. Очевидно, что пропускную способность удалось бы увеличить, если бы мы разрешили простаивающему потоку забрать задачу из начала другой очереди.

Заимствование работ – развитие этой идеи; заметив, что заимствование работы из начала чужой очереди простаивающим потоком создает риск конкуренции за начальный элемент, мы заменяем очереди деками и просим простаивающие потоки брать задачи из *конца* чужого дека. Этот механизм оказался весьма эффективным и получил широкое распространение.

Реализация BlockingDeque

У интерфейса `BlockingDeque` есть всего одна реализация в JDK: класс `LinkedBlockingDeque`. Этот класс имеет структуру дважды связанного списка, как `LinkedList`. Дек может быть ограниченным, поэтому, помимо двух стандартных конструкторов, предлагается третий, задающий емкость:

```
LinkedBlockingDeque(int capacity)
```

Этот класс имеет такие же характеристики производительности, как `LinkedBlockingQueue`: добавление элемента в очередь или удаление из нее занимает постоянное время, а такие операции, как `contains`, требующие обхода очереди, – линейное время. Итераторы слабо согласованы.

Сравнение реализаций очереди

В табл. 13.2 приведены данные о производительности без учета накладных расходов на блокировки и команду CAS для выборочных операций рассмотренных выше реализаций `Deque` и `Queue`. Эти результаты могут быть вам интересны с точки зрения понимания выбранной вами реализации, но, как было сказано в начале этой главы, маловероятно, что именно это станет решающим фактором. Скорее всего, выбор будет продиктован требованиями приложения к функционалу и конкурентности реализации.

Таблица 13.2. Сравнительная производительность различных реализаций `Queue` и `Deque`

	offer	peek	poll	size
`PriorityQueue`	*O*(log *N*)	*O*(1)	*O*(log *N*)	*O*(1)
`ConcurrentLinkedQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(*N*)
`ArrayBlockingQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`LinkedBlockingQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`PriorityBlockingQueue`	*O*(log *N*)	*O*(1)	*O*(log *N*)	*O*(1)
`DelayQueue`	*O*(log *N*)	*O*(1)	*O*(log *N*)	*O*(1)
`LinkedTransferQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(*N*)
`LinkedList`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`ArrayDeque`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`LinkedBlockingDeque`	*O*(1)	*O*(1)	*O*(1)	*O*(1)

При выборе реализации `Queue` сразу же возникает вопрос, должна ли она поддерживать конкурентный доступ. Если нет, то выбор очевиден: если нужна FIFO-очередь, берите `ArrayDeque`, а если очередь с приоритетами, то `PriorityQueue`.

Если же приложению требуется потокобезопасность, то далее нужно рассмотреть упорядочение. Если нужна очередь с приоритетами или с задержкой, то выбирайте `PriorityBlockingQueue` или `DelayQueue` соответственно. Если же порядок FIFO приемлем, то третий вопрос – нужны ли блокирующие методы, как обычно бывает в задачах типа производитель–потребитель (потому ли, что потребители должны ждать, когда очередь пуста, или потому что вы хотите ограничить спрос на них, сделав очередь ограниченной и тем самым заставив иногда ждать производителей). Если ни блокирующие методы, ни ограничение размера очереди не нужны, выбирайте эффективную и свободную от ожидания реализацию `ConcurrentLinkedQueue`.

Если вам нужна блокирующая очередь, потому что приложению необходима поддержка взаимодействия между производителем и потребителем, то подумайте, действительно ли вам нужно буферизовать данные или достаточно просто безопасного обмена данными между потоками. Если вы можете обойтись без буферизации (обычно потому, что уверены, что потребителей всегда будет достаточно, чтобы избежать накопления данных), то `SynchronousQueue` – эффективная альтернатива другим реализациям блокирующей FIFO-очереди, `LinkedBlockingQueue` и `ArrayBlockingQueue`.

В противном случае остается выбор между этими двумя вариантами. Если вы не можете зафиксировать реалистичную верхнюю границу размера очереди, то должны выбрать `LinkedBlockingQueue`, потому что `ArrayBlockingQueue` всегда ограничена. Если требуется ограниченная очередь, то выбирать одну из двух следует исходя из соображений производительности. В табл. 13.2 их характеристики производительности одинаковы, но это формулы лишь для последовательного доступа; как они поведут себя в конкурентной среде – другой вопрос. На относительную производительность влияет ряд факторов:

- наличие отдельных блокировок на начало и конец очереди `LinkedBlockingQueue` означает, что производитель и потребитель не будут конкурировать между собой. Класс `ArrayBlockingQueue` пользуется одной блокировкой;
- у ограниченности `ArrayBlockingQueue` есть тот плюс, что потребление ей памяти предсказуемо, она никогда не выделяет дополнительную память, в отличие от `LinkedBlockingQueue`. С другой стороны, предварительное выделение означает, что очередь, возможно, будет потреблять больше памяти, чем необходимо, – в отличие от `LinkedBlockingQueue`, которая выделяет примерно столько памяти, каков фактический размер очереди;
- но при этом `ArrayBlockingQueue` не выделяет память для новых объектов при каждом добавлении в очередь, в отличие от `LinkedBlockingQueue`;
- для связанных структур данных характерно гораздо худшее поведение кеша, чем для основанных на массиве. Как мы видели в разделе «Память» главы 9, непопадание в кеш может оказаться фактором, определяющим производительность алгоритма.

Решая, какую очередь использовать – ограниченную или неограниченную, – иногда полезно учитывать другие факторы, не связанные напрямую с производительностью. Неограниченная очередь может в течение кратких промежутков времени сильно вырастать, чтобы буферизовать пики в работе производителей, с которыми потребители не справляются. Это может быть полезно, но есть и потенциальные недостатки: увеличенная задержка, неспособность блокировать производителей, оказав на них противодавление, и даже опасность, что очередь займет всю доступную память.

Эти факторы по-разному сочетаются в разных приложениях. Если производительность очереди критична для нормальной работы приложения, то следует измерить обе реализации на самом авторитетном для вас тесте: самом приложении.

Заключение

Как показано в этой главе, роль очередей в системах отличается от других типов коллекций. Поскольку они используются в основном для передачи данных, а не являются частью состояния объекта, в их API упор сделан на средствах, необходимых для взаимодействия: почти во всех случаях это блокирующие методы.

В следующей главе мы вернемся к типам коллекций, в большей степени ориентированных на хранение, и изучим ту, которая, пожалуй, встречается чаще всего: `List`.

Глава 14

Списки

Списки – пожалуй, самая широко используемая на практике из коллекций Java. Список (в отличие от множества) может содержать дубликаты и (в отличие от очереди) позволяет пользователю видеть и контролировать порядок своих элементов. В каркасе коллекций спискам соответствует интерфейс `List<E>`.

В контракте `List` метод `equals` переопределен; он говорит, что один список равен другому списку тогда и только тогда, когда они содержат одинаковые элементы, следующие в одном и том же порядке. Метод `hashCode` также переопределен, как и должно быть, когда переопределен метод `equals` (см. «Хеш-таблицы» в разделе «Реализации» главы 9). В документации интерфейса `List` описан алгоритм, который следует использовать для вычисления хеш-кода списка по хеш-кодам его элементов.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter14*.

Методы интерфейса List

В интерфейсе `List` объявлены методы для позиционного доступа, для поиска заданного значения, для генерирования представлений и для создания итераторов `ListIterator` – подтипа `Iterator` с дополнительными возможностями вследствие последовательной природы списка. Кроме того, методы, унаследованные от `SequencedCollection`, дают удобные более короткие версии типичных вызовов для позиционного доступа. Наконец, имеются статические фабричные методы для создания немодифицируемых списков разной длины.

Методы позиционного доступа

`void add(int index, E e)`	вставить элемент `e` в в позицию с индексом `index`
`boolean addAll(int index, Collection<? extends E> c)`	вставить содержимое `c`, начиная с позиции с индексом `index`
`E get(int index)`	вернуть элемент в позиции с индексом `index`
`E remove(int index)`	удалить элемент в позиции с индексом `index`
`E set(int index, E e)`	заменить элемент в позиции с индексом `index`

Эти методы получают доступ к элементу в позиции списка с заданным числовым индексом. К сожалению, объявление `remove` конфликтует с унаследованным методом `Collection::remove` в случае `List<Integer>`. И хотя правила разрешения перегрузки ясно говорят, что вызов вида `list.remove(0)` относится к элементу с индексом 0, для программистов это является постоянным источником путаницы, а значит, и ошибок.

Методы поиска

`int indexOf(Object o)`	вернуть индекс первого вхождения `o`
`int lastIndexOf(Object o)`	вернуть индекс последнего вхождения `o`

Эти методы ищут указанный объект в списке и возвращают его числовую позицию или –1, если объект не найден.

Методы генерирования представлений

`List<E> subList(int fromIndex, int toIndex)`	вернуть представление части списка
`List<E> reversed()`	вернуть представление исходной коллекции в обратном порядке

Эти методы дают различные представления (см. раздел «Представления» главы 9) списка. Метод `subList` работает так же, как метод `subSet` интерфейса `SortedSet` (см. раздел «NavigableSet» главы 12), но с использованием позиций элементов, а не их значений: возвращенный список содержит элементы в позициях с индексами от `fromIndex` до `toIndex`, не включая последний. Возвращенный список не существует сам по себе; это просто представление части списка, из которого он был получен, а все произведенные в нем изменения отражаются в исходном списке. Однако имеется важное отличие от `subSet`: изменения, произведенные в подсписке, попадают в исходный список, но обратное верно только для неструктурных изменений. Если в исходном списке были произведены операции вставки или удаления, минуя подсписок, то последующие попытки использовать подсписок приведут к исключению `ConcurrentModificationException`.

Список, возвращенный методом `reversed`, допускает любые модификации, разрешенные в исходном списке.

Методы обхода списка

`ListIterator<E> listIterator()`	вернуть `ListIterator` для этого списка, первоначально позиционированный на элементе с индексом 0
`ListIterator<E> listIterator(int idx)`	вернуть `ListIterator` для этого списка, первоначально позиционированный на элементе с индексом `idx`

Эти методы возвращают `ListIterator`, представляющий собой `Iterator` с расширенной семантикой, учитывающей последовательную природу списка. Методы, добавленные `ListIterator`, поддерживают обход списка в обратном

направлении, изменение элементов списка и добавление новых, а также получение текущей позиции итератора. Текущая позиция `ListIterator` всегда находится между двумя элементами, так что для списка длины *N* существует *N* + 1 возможных позиций итератора: от 0 (перед первым элементом) до *N* (после последнего элемента). Вызов первого перегруженного варианта возвращает `listIterator`, установленный в позицию 0. Второй перегруженный вариант использует переданное значение, чтобы установить начальную позицию возвращенного `listIterator`.

Дополнительно к методам `Iterator` `hasNext`, `next` и `remove` интерфейс `ListIterator` объявляет следующие методы:

`void add(E e)`	вставить указанный элемент в список
`boolean hasPrevious()`	вернуть `true`, если для этого итератора списка еще имеются элементы в обратном направлении
`int nextIndex()`	вернуть индекс элемента, который был бы возвращен следующим вызовом `next`
`E previous()`	вернуть предыдущий элемент списка
`int previousIndex()`	вернуть индекс элемента, который был бы возвращен следующим вызовом `previous`
`void set(E e)`	заменить последний элемент, возвращенный `next` или `previous`, заданным элементом

На рис. 14.1 изображен список из трех элементов. Рассмотрим итератор в позиции 2, который был получен либо переходом из какого-то другого места, либо в результате вызова `listIterator(2)`. Последствия большинства операций этого итератора интуитивно понятны: `add` вставляет элементы в текущую позицию итератора (между элементами с индексами 1 и 2); `hasPrevious` и `hasNext` возвращают `true`; `previous` и `next` возвращают элементы с индексами 1 и 2 соответственно, а `previousIndex` и `nextIndex` возвращают сами эти индексы. В крайних позициях списка – на рисунке это 0 и 3 – `previousIndex` и `nextIndex` вернули бы –1 и 3 (размер списка) соответственно, а `previous` и `next` возбудили бы исключение `NoSuchElementException`.

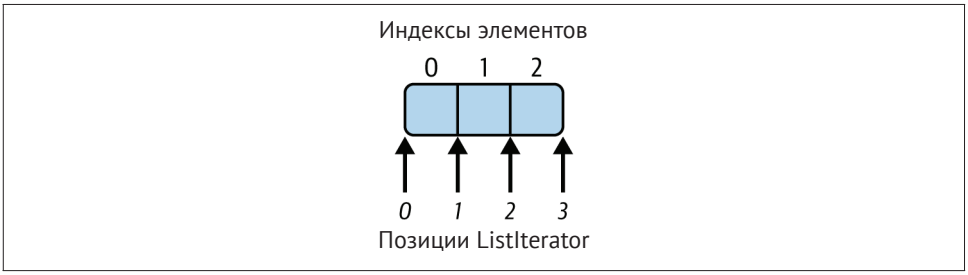


Рис. 14.1. Позиции `ListIterator`

Операции `set` и `remove` работают иначе. Их результат зависит не от текущей позиции итератора, а от его «текущего» элемента, т. е. того, который был

пройден последним в результате вызова метода `next` или `previous`: `set` заменяет текущий элемент, а `remove` удаляет его. Если текущего элемента нет, – потому ли что итератор был только что создан, или потому что текущий элемент был удален, – то эти методы возбуждают исключение `IllegalStateException`.

Методы, унаследованные от SequencedCollection

Если не считать метода `reversed` (см. раздел «Методы генерирования представлений» выше), то унаследованные от `SequencedCollection` методы – это просто удобные версии описанных выше позиционных методов `List`. В табл. 14.1 показано соответствие между методами `SequencedCollection` и методами позиционного доступа, объявленными в `List`.

Таблица 14.1. Вызовы методов `SequencedCollection` и их эквиваленты в `List`

Вызов метода `SequencedCollection`	Вызов метода позиционного доступа `List`
`addFirst(el)`	`add(0, el)`
`addLast(el)`	`add(el)`
`getFirst()`	`get(0)`
`getLast()`	`get(size() - 1)`
`removeFirst()`	`remove(0)`
`removeLast()`	`remove(size() - 1)`

Фабричные методы

Эти методы, перегруженные версии `List::of`, создают немодифицируемые объекты `List`. Мы обсудим их в разделе «UnmodifiableList» ниже.

Использование методов List

Посмотрим, как можно использовать некоторые из этих методов в менеджере задач. В предыдущей главе мы рассматривали представление задач на один день классом на основе очереди с возможностью завершения работы. Один из полезных способов расширить приложение – завести несколько объектов такого типа, каждый из которых представляет задачи, запланированные на какой-то день в будущем. Мы можем хранить ссылки на эти задачи в списке `List`, который (чтобы не усложнять и не связываться с деталями `java.time`) будет индексирован номером будущего дня. Следовательно, очередь задач, запланированных на сегодня, будет храниться в элементе с индексом 0, на завтра – в элементе с индексом 1 и т. д. В примере 14.1 показан код планировщика.

Пример 14.1. *Планировщик задач на базе списка*

org/jgcbook/chapter14/B_using_the_methods_of_list/DailyTaskScheduler

```
public class DailyTaskScheduler {
  private final List<StoppableTaskQueue> schedule;
  private final int FORWARD_PLANNING_DAYS = 365;
```

```
    public DailyTaskScheduler() {
      schedule = Stream.generate(StoppableTaskQueue::new)
          .limit(FORWARD_PLANNING_DAYS)
          .collect(Collectors.toCollection(CopyOnWriteArrayList::new));
    }

    public Optional<PriorityTask> getTopTask() {
      return schedule.stream()
          .map(StoppableTaskQueue::getFirstTask)
          .filter(Objects::nonNull)
          .findFirst();
    }

    // Этот метод вызывается в полночь и создает очередь на новый день на
    // горизонте планирования. Он удаляет и останавливает очередь для дня 0 и
    // перепланирует задачи из нее на новый день 0.
    public synchronized void rollOver() {
      schedule.add(new StoppableTaskQueue());
      StoppableTaskQueue oldDay = schedule.removeFirst();
      schedule.getFirst().addTasks(oldDay.shutDown());
    }

    public void addTask(PriorityTask task, int day) {
      addTasks(List.of(task), day);
    }

    public void addTasks(Collection<PriorityTask> tasks, int day) {
      if (day < 0 || day >= FORWARD_PLANNING_DAYS)
        throw new IllegalArgumentException("day out of range");

      while (! schedule.get(day).addTasks(tasks)) {
        // Если addTasks() завершается неудачно, значит, была инициирована
        // остановка этой очереди, которая, следовательно, должна быть очередью
        // на день 0, уже удаленной из плана. Поэтому правильное действие –
        // вызвать addTasks() для очереди на день 0
      }
    }
  }
```

Хотя главная цель этого примера – продемонстрировать использование методов интерфейса `List`, а не исследовать конкретную реализацию, мы не можем не выбрать какую-то. Так как основным фактором будут требования приложения к конкурентности, рассмотрим их прямо сейчас. Они просты: клиенты, потребляющие или производящие задачи, только читают список, представляющий план, поэтому после конструирования запись в него производится лишь в конце каждого дня. В этот момент очередь на текущий день удаляется из плана, а новая добавляется в его конец (на «горизонте планирования», в качестве которого мы выбрали год). Нам не нужно запрещать клиентам использовать очередь на текущий день, перед тем как это произойдет, потому что дизайн класса `StoppableTaskQueue` из примера 13.1 гарантирует, что они смогут упорядоченно завершиться на этапе остановки очереди. Поэтому единственное необходимое исключение – гарантировать, что клиенты не попытаются читать сам план в то время, когда процедура перехода через сутки меняет его значения.

Вспомнив обсуждение класса `CopyOnWriteArrayList` в разделе «Конкурентные коллекции» главы 9, вы согласитесь, что он отлично отвечает этим требовани-

ям. Он оптимизирует доступ для чтения, в полном согласии с одним из наших требований. А в случае операции записи, он синхронизируется только на время, достаточное для создания новой копии своего внутреннего массива, и не более, а значит, отвечает другому требованию – предотвращению интерференции между операциями чтения и записи.

После выбора реализации нетрудно понять код конструктора в примере 14.1; записывать в список дорого, поэтому разумно использовать преобразующий конструктор, чтобы заполнить его очередями задач на год вперед за одну операцию.

Метод `getTask` не вызывает трудностей – мы просто обходим очереди задач, начиная с сегодняшней, и ищем запланированную задачу. Если метод не находит невыполненных задач, то возвращает `null` – и если день, свободный от задач, стоило отметить, то как мы отпразднуем год без единой задачи?

Ежедневно в полночь система будет вызывать метод `rollOver`, который реализует печальный ритуал закрытия очереди задач на сегодняшний день с переносом оставшихся в очередь на следующий день. Последовательность действий здесь важна. Сначала `rollOver` удаляет очередь из списка, и в этот момент производители и потребители, возможно, еще готовы вставлять или удалять элементы. Затем он вызывает метод `StoppableTaskQueue::shutDown`, который, как мы видели в примере 13.1, возвращает оставшиеся в очереди задачи и гарантирует, что больше ни одной добавлено не будет. В зависимости от того, как далеко продвинулись вызовы `addTask`, они либо завершатся успешно, либо вернут `false`, сообщая, что дойти до конца не удалось, потому что очередь остановлена.

И это довод в обоснование логики `addTask`. Единственная ситуация, когда метод `addTask` класса `StoppableTaskQueue` может вернуть `false`, – если очередь, для которой он вызван, уже остановлена. Поскольку это возможно только в результате перехода на новые сутки, значение `false`, возвращенное `addTask`, должно быть результатом того, что поток-производитель получил ссылку на очередь на день 0 как раз тогда, когда был готов начаться переход через границу суток. В этом случае текущий элемент `0` списка уже содержит очередь на новый день 0, и можно безопасно попробовать еще раз. Если и вторая попытка оказалась неудачной, то, видимо, поток был приостановлен ровно на 24 часа! В данном случае это маловероятно, но любая программа, которая может столкнуться с таким сценарием, должна учитывать эту возможность. Правильное действие – продолжить попытку совершить переход.

Мы могли бы продемонстрировать, что переход через границу суток работает, добавив задачу, а затем – после перехода – проверив, что она все еще находится в системе. Но более убедительно было бы показать, что в процессе перехода задачи, запланированные на старый день 0, теперь находятся в очереди на новый день 0, так что в новой очереди присутствуют все задачи в правильном порядке приоритетов. Это можно сделать, временно испортив дизайн классов `StoppableTaskQueue` и `DailyTaskScheduler` путем добавления диагностических методов, раскрывающих их состояние. Для `StoppableTaskQueue` мы можем добавить метод, копирующий (мгновенное) содержимое очереди в список. Хотя итераторы очередей с приоритетами не возвращают элементы в каком-то определенном порядке, метод `drainTo` интерфейса `BlockingQueue` учитывает упорядочение очереди, когда перемещает ее в коллекцию. Так как метод `drainTo` деструктивный, применять его следует

к копии очереди. По счастью, копирующий конструктор `PriorityBlockingQueue` также учитывает упорядочение своего аргумента:

org/jgcbook/chapter13/C_blocking_queue/StoppableTaskQueueWithAccessor

```
public List<PriorityTask> getTaskList() {
  BlockingQueue<PriorityTask> copyQueue = new PriorityBlockingQueue<>(taskQu
eue);
  List<PriorityTask> taskList = new ArrayList<>();
  copyQueue.drainTo(taskList);
  return taskList;
}
```

А в класс `DailyTaskScheduler` мы можем добавить метод, который вызывает `getTasks` и возвращает содержимое плана на указанный день:

org/jgcbook/chapter14/B_using_the_methods_of_list/DailyTaskSchedulerWithAccessor

```
public List<PriorityTask> tasksForDay(int day) {
  return schedule.get(day).getTaskList();
}
```

Теперь мы можем продемонстрировать переход через границу суток в действии.

org/jgcbook/chapter14/B_using_the_methods_of_list/DailyTaskSchedulerWithAccessor

```
PriorityTask guiCoding = new PriorityTask(new CodingTask("gui"), Priority.MEDIUM);
PriorityTask logicCoding = new PriorityTask(new CodingTask("logic"), Priority.HIGH);

DailyTaskScheduler scheduler = new DailyTaskScheduler();
scheduler.addTask(guiCoding, 0);
assert scheduler.tasksForDay(0).equals(List.of(guiCoding));
scheduler.addTask(logicCoding, 1);
scheduler.rollOver();
assert scheduler.tasksForDay(0).equals(List.of(logicCoding, guiCoding));
```

Отметим, что метод `rollOver` весьма дорогой, он дважды записывает в план, а, поскольку план представлен объектом `CopyOnWriteArrayList` (см. раздел «CopyOnWriteArrayList» ниже в этой главе), каждая запись приводит к копированию всего базового массива. Аргумент в пользу этой реализации – то, что `rollOver` вызывается редко по сравнению с количеством обращений к методу `getTask`, который обходит план. Альтернативой `CopyOnWriteArrayList` было бы использование реализации `BlockingQueue`, но улучшение, которое оно принесло бы редко используемому методу `rollOver`, было бы оплачено замедлением часто используемого метода `getTask`, потому что итераторы очереди не предназначены для ситуаций, где производительность важна.

Использование методов диапазонного представления и итератора

В примере 14.1 в нескольких местах используются методы, принадлежащие одной из четырех описанных ранее групп – позиционный доступ. Чтобы увидеть, чем могут быть полезны методы диапазонного представления и итератора, рассмотрим, как объект типа `DailyTaskScheduler` мог бы экспортировать свой план или его часть клиенту для модификации. Мы хотим, чтобы клиент

мог увидеть эту часть плана и, быть может, вставить или удалить из нее задачи, но мы категорически требуем запретить вставку и удаление элементов в сам список, поскольку они представляют последовательность дней, на которые запланированы задачи. Стандартный способ решить такую задачу – воспользоваться немодифицируемым списком, предоставленным классом `Collections` (см. раздел «Немодифицируемые коллекции» главы 16). В данном случае возможен альтернативный подход: вернуть итератор списка, поскольку слепковые итераторы для коллекций, копируемых при записи, не поддерживают модификацию базовой коллекции. Итак, мы могли бы определить метод, предоставляющий клиентам «окно планирования»:

```
ListIterator<StoppableTaskQueue> getSubSchedule(int startDay, int endDay) {
  return schedule.subList(startDay, endDay).listIterator();
}
```

Это представление годится для сегодняшнего дня, но нужно не забыть уничтожить его в полночь, когда структурные изменения – удаление и вставка элементов – сделают его недействительным.

Реализации интерфейса List

В каркасе коллекций имеется четыре конкретных реализации интерфейса `List` (см. рис. 14.2), различающиеся скоростью выполнения операций и поведением в условиях конкурентности. Но в отличие от `Set` и `Queue`, `List` не имеет подынтерфейсов для определения различий функционального поведения. В следующих разделах рассматриваются все реализации и приводится сравнение их производительности.

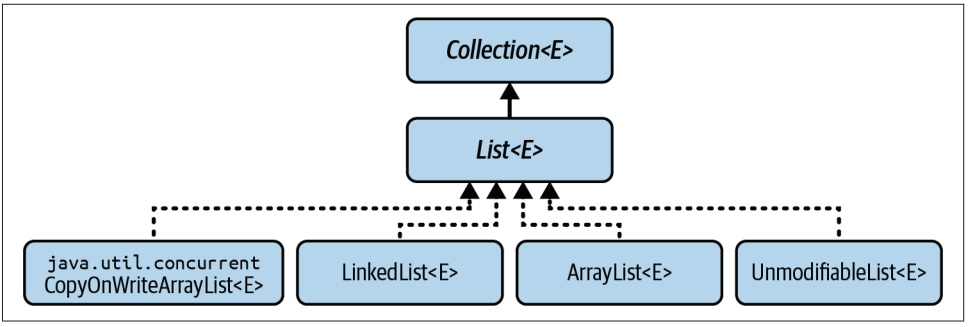


Рис. 14.2. Реализации интерфейса `List`

ArrayList

Массивы – часть языка Java, они имеют очень удобный синтаксис, но их главный недостаток – то, что после создания их размер нельзя изменить, – делает их все менее популярными, чем реализации `List`, которые (если вообще допускают изменение размера) являются бесконечно расширяемыми. На самом деле в основе самой употребительной реализации `List` – `ArrayList` – лежит массив.

В стандартной реализации `ArrayList` элементы списка хранятся в непрерывном массиве, и первый элемент имеет индекс 0. Требуется массив, достаточно

большой для хранения всех элементов (с достаточной *емкостью*), а также способ запоминания количества «занятых» позиций (размер списка). Если `ArrayList` вырос настолько, что размер сравнялся с емкостью, то попытка добавить еще один элемент требует замены базового массива большим, в котором можно было бы сохранить старое содержимое и новый элемент и который имел был запас для дальнейшего расширения (в стандартной реализации, длина нового массива в 1.5 раза больше длины старого). Как было объяснено в разделе «Счетчик команд и нотация О большое» главы 9, амортизированная сложность такой реализации $O(1)$.

Производительность `ArrayList` отражает производительность массива для операций «произвольного доступа»: `set` и `get` занимают постоянное время. Недостаток реализации на основе массива – вставка и удаление элементов в произвольных позициях, потому что для этого может понадобиться перемещение других элементов. Мы уже сталкивались с этой проблемой при рассмотрении метода `remove` итераторов очередей на основе массивов (например, `ArrayBlockingQueue` – см. раздел «Реализации BlockingQueue» главы 13). Но производительность позиционных методов `add` и `remove` для списков гораздо важнее, чем метода `Iterator::remove` для очередей.

Например, на рис. 14.3a показан новый `ArrayList` после добавления трех элементов с помощью следующих предложений:

```
List<Character> charList = new ArrayList<>();
Collections.addAll(charList, 'a', 'b', 'c');
```

Если теперь мы захотим удалить элемент с индексом 1, то реализация должна сохранить порядок остальных элементов и гарантировать, что занятая область массива по-прежнему начинается позицией с индексом 0. Поэтому элемент с индексом 2 следует переместить в позицию 1, элемент с индексом 3 – в позицию 2 и т. д. На рис. 14.3b показан пример `ArrayList` после выполнения этой операции. Поскольку элементы нужно перемещать по очереди, временная сложность этой операции пропорциональна размеру списка (впрочем, эту операцию обычно можно реализовать аппаратно, поэтому коэффициент пропорциональности мал).

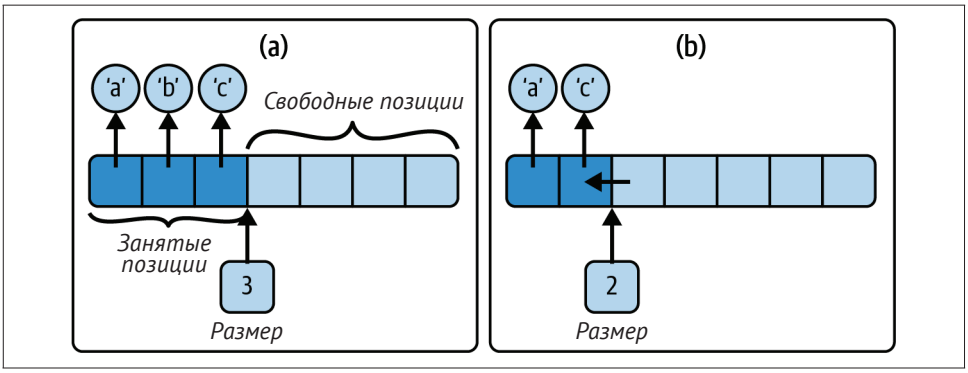


Рис. 14.3. Удаление элемента из `ArrayList`

Но даже если так, вы можете вспомнить обсуждение циклических массивов для реализации `ArrayBlockingQueue` и `ArrayDeque` (см. раздел «Реализации Deque»

главы 13) и задаться вопросом, почему для реализации `ArrayList` не был выбран циклический массив. Да, методы `add` и `remove` циклического массива работают гораздо быстрее, только если вызываются для позиции с индексом 0, но это очень частый случай, а накладные расходы, связанные с использованием циклического массива, настолько малы, что вопрос остается открытым.

На самом деле в статье Heinz Kabutz (2001) в издании «The Java Specialists' Newsletter» даже представлен набросок реализации списка на основе циклического массива. В принципе, еще остается возможность, что `ArrayList` будет переписан таким образом, что, возможно, приведет к значительному выигрышу производительности во многих существующих приложениях Java. Альтернатива, хотя и менее вероятная, – переработка циклической `ArrayDeque` с реализацией методов `List`. А пока, если в вашем приложении используется список, в котором скорость вставки и удаления элемента в начале списка важнее, чем операции произвольного доступа, то подумайте о том, чтобы переписать его под интерфейс `Deque` и воспользоваться всеми преимуществами очень эффективной реализации `ArrayDeque`.

Как было отмечено при обсуждении `ArrayBlockingQueue` в предыдущей главе, классы коллекций переменного размера на основе массивов могут иметь один конфигурационный параметр: начальный размер массива. Поэтому, помимо стандартных для каркаса коллекций конструкторов, `ArrayList` имеет еще один, позволяющий задать начальную емкость, достаточно большую, чтобы вместить элементы коллекций без необходимости часто выполнять операции создания и копирования. В версии Java 21 начальная емкость списка `ArrayList`, созданного конструктором по умолчанию, равна 10, а списка, инициализированного элементами другой коллекции, – размеру этой коллекции.

Итераторы `ArrayList` обладают свойством быстрого отказа.

LinkedList

Класс `LinkedList` мы обсуждали в контексте реализации `Deque` в разделе «Реализации Deque» главы 13. Редко находится основательная причина выбирать его в качестве реализации `List`: чтобы дойти до нужной позиции, приходится последовательно проходить по списку, поэтому позиционные операции `add` и `remove` в среднем имеют линейную временную сложность. Реализацию на основе циклического массива, например `ArrayDeque`, `LinkedList` превосходит только при добавлении первого или последнего элемента, потому что никогда не приходится копировать или изменять размер коллекции, как вынуждены делать реализации на основе массива в случае исчерпания емкости. В принципе, добавление и удаление элементов с помощью итератора занимают постоянное время, если не приходится обходить список, а не линейное время, как в реализациях на основе обычного, не циклического массива. Однако другие недостатки `LinkedList`, описанные в разделе «Избегайте LinkedList» главы 17, означают, что даже в этой ситуации достижение производительности, лучшей, чем у `ArrayList`, маловероятно.

CopyOnWriteArrayList

В разделе «Реализации интерфейса Set» главы 12 мы обсуждали `CopyOnWriteArraySet` – реализацию множества, предназначенную для обеспече-

ния одновременно потокобезопасности и очень быстрого доступа для чтения. `CopyOnWriteArrayList` – это реализация списка с теми же проектными целями. Такое сочетание потокобезопасности с быстрым доступом для чтения полезно в некоторых конкурентных программах, особенно когда коллекция объектов-наблюдателей получает частые уведомления о событиях. Платить за это приходится тем, что массив, лежащий в основе коллекции, должен рассматриваться как неизменяемый, поэтому при внесении любых изменений в коллекцию создается новая копия. Эта плата может быть приемлемой, если множество наблюдателей изменяется редко.

Класс `CopyOnWriteArraySet` на самом деле делегирует все свои операции экземпляру `CopyOnWriteArrayList`, пользуясь атомарными операциями `addIfAbsent` и `addAllAbsent`, предоставляемыми последним, чтобы поддержать методы `add` и `addAll` интерфейса `Set`, которые не должны приводить к появлению дубликатов в множестве. Помимо двух стандартных конструкторов (см. раздел «Конструкторы коллекций» главы 10), в классе `CopyOnWriteArrayList` имеется еще один, который позволяет инициализировать его объект элементами переданного массива. Итераторы класса являются слепковыми и отражают состояние списка в момент создания.

UnmodifiableList

Как и в случае *UnmodifiableSet* (см. раздел «UnmodifiableSet» главы 12), ни в документации, ни в коде каркаса коллекций вы не найдете никаких упоминаний о реализации *UnmodifiableList<E>* интерфейса `List`: это еще одно имя, придуманное авторами книги для семейства закрытых на уровне пакета классов, к которым программист клиентских приложений не может обратиться по имени, но которые важны, потому что предоставляют реализацию немодифицируемых списков, полученных от различных перегруженных вариантов фабричных методов `List::of` и `List::copyOf`. Свойства членов этого семейства описаны в документации `List`:

- они немодифицируемы: элементы нельзя ни добавлять, ни удалять. Вызов любого изменяющего метода возбуждает исключение `UnsupportedOperationException`;
- они не допускают `null`. Попытка поместить в них при создании элементы `null` приводит к исключению `NullPointerException`;
- списки и их представления `subList` реализуют интерфейс `RandomAccess`.

Статические фабричные методы (в следующих таблицах ключевое слово `static` для краткости опущено) можно разбить на те же три группы, что и методы *UnmodifiableSet*. Первая группа содержит только перегруженный вариант метода `of`, который не имеет аргументов и возвращает пустой немодифицируемый список:

`<E> List<E> of()`	вернуть немодифицируемый список, не содержащий ни одного элемента

Вторая группа методов позволяет создавать немодифицируемый список, содержащий до 10 заданных элементов:

`<E> List<E> of(E e1)`	вернуть немодифицируемый список, содержащий один элемент
`<E> List<E> of(E e1, E e2)`	вернуть немодифицируемый список, содержащий два элемента
`<E> List<E> of(E e1, E e2, E e3)`	вернуть немодифицируемый список, содержащий три элемента
дополнительные перегруженные варианты с числом аргументов от 4 до 9	
`<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)`	вернуть немодифицируемый список, содержащий десять элементов

Третья группа методов позволяет создавать немодифицируемый список из коллекции, массива или списка с переменным числом аргументов:

`<E> List<E> copyOf(Collection<? extends E> coll)`	вернуть немодифицируемый список, содержащий произвольное число элементов
`<E> List<E> of(E... elements)`	вернуть немодифицируемый список, содержащий произвольное число элементов

Характеристики производительности классов из семейства *UnmodifiableList* для операций чтения такие же, как у класса `ArrayList`.

Сравнение реализаций списка

В табл. 14.2 приведены данные о сравнительной производительности выборочных операций классов, реализующих интерфейс `List`. И хотя выбор здесь гораздо уже, чем для очередей и даже для множеств, можно применить ту же процедуру исключения. Как и в случае очередей, сначала нужно задать вопрос, требуется ли приложению потокобезопасность. Если да, то следует использовать `CopyOnWriteArrayList`, если это возможно, т. е. если запись в список производится сравнительно редко. Если нет, то придется использовать синхронизированную обертку (см. раздел «Синхронизированные коллекции» главы 16) вокруг `ArrayList` или `LinkedList`.

Таблица 14.2. Сравнительная производительность различных реализаций `List`

	get	add	add(int,e)	contains	iterator.next	remove(0)	iterator.remove
`ArrayList`	*O*(1)	*O*(1)	*O*(*N*)	*O*(*N*)	*O*(1)	*O*(*N*)	*O*(*N*)
`LinkedList`	*O*(*N*)	*O*(1)	*O*(1)[a]	*O*(*N*)	*O*(1)	*O*(1)	*O*(1)[b]
`CopyOnWriteArrayList`	*O*(1)	*O*(*N*)	*O*(*N*)	*O*(*N*)	*O*(1)	*O*(*N*)	–[b]

[a] Сложность *O*(1) для операций `add(int,e)` и `iterator.remove` класса `LinkedList` следует понимать в контексте сложности *O*(*N*) операции поиска места вставки или удаления.

[b] Возбуждает `UnsupportedOperationException`.

Для большинства применений списков выбирать приходится между `ArrayList` и `LinkedList`, синхронизированными или нет. Просматривая табл. 14.2, вы можете подумать, что бывают ситуации, – например, если ваше приложение часто вставляет и удаляет элементы близко к началу списка, – когда лучше остановиться на `LinkedList`. Но на практике это редкий случай; почему – мы узнаем в разделе «Избегайте LinkedList» главы 17. Если вставка и удаление производятся *точно* в начале списка, то однопоточный код лучше написать под интерфейс `Deque` и воспользоваться очень эффективной его реализацией `ArrayDeque`. Если произвольный доступ требуется сравнительно редко, то воспользуйтесь итератором или скопируйте элементы `ArrayDeque` в массив методом `toArray`. Как всегда, если сомневаетесь, измеряйте производительность всех вариантов приложения.

Заключение

Этой главой о списках мы завершаем изучение подтипов `Collection`. Несмотря на свою популярность, `List` – единственный из основных интерфейсов каркаса коллекций, не имеющий широко распространенной конкурентной реализации. Этим он резко отличается от предмета следующей главы, интерфейса `Map`, конкурентные реализации которого незаменимы во многих системах уровня предприятия, написанных на Java.

Глава 15

Отображения

Интерфейс `Map` – последний из крупных интерфейсов в каркасе коллекций и единственный, не наследующий `Collection`. В отображении `Map` хранятся пары ключ–значение, или *записи*, в которых ключи уникальны. Важность этого интерфейса заключается в том, что его реализации предоставляют очень быстрые – в идеале с постоянным временем – операции поиска значения, соответствующего заданному ключу.

Интерфейс `Map` переопределяет методы `equals` и `hashCode` и предоставляет их определения (`hashCode` всегда необходимо переопределять, если переопределен `equals`, как мы видели при обсуждении хеш-таблиц в разделе «Реализации» главы 9). В интересах этих методов `Map` рассматривается как множество `Set` записей отображения (пар ключ–значение). Как и в случае контракта `Set`, два отображения `Map` равны тогда и только тогда, когда они одинакового размера и содержат равные записи. Хеш-код `Map` вычисляется как сумма хеш-кодов его записей. Запись отображения описывается интерфейсом `Map.Entry`. Две записи отображения равны, если равны их ключи и значения, а хеш-код записи вычисляется как результат применения операции ИСКЛЮЧАЮЩЕЕ ИЛИ к хеш-кодам ключа и значения.

Примеры кода к этой главе находятся по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/jgcbook/chapter15*.

Методы интерфейса Map

Методы `Map` можно разбить на группы в соответствии с двумя способами взглянуть на отображение: как на множество записей или как на механизм поиска. С первой точки зрения, операции примерно соответствуют операциям интерфейсов `Iterable` и `Collection`. Со второй точки зрения, важными группами являются операции создания представлений, составные операции и фабричные методы.

В тех случаях, когда операции `Map` принимают лямбда-выражения в качестве параметров, типы параметров этих выражений часто ограничены сверху или снизу типом ключа или значения. Аналогично `Map.Entry` обычно параметризуется ограниченными типами. Однако в листингах этого раздела для краткости используются точные типы.

Операции, подобные Iterable

`void forEach(BiConsumer<K,V> action)`	выполнить действие `action` для каждой записи отображения в порядке обхода множества записей, если он определен

Точно так же как `Iterable::forEach` (см. раздел «Итерируемые объекты и итераторы» главы 9) позволяет коллекции передать каждый из своих элементов по очереди потребителю `Consumer`, этот метод позволяет отображению передать каждую из своих записей ключ–значение потребителю `BiConsumer`.

Операции, подобные Collection

Методы этой группы можно разделить на три подгруппы, которые в общих чертах соответствуют первым трем группам операций `Collection`: для добавления или модификации записей, для удаления записей и для опроса содержимого отображения.

Добавление или замена записей

`V put(K key, V value)`	добавить или заменить пару ключ-значение; вернуть старое значение, если ключ найден, иначе вернуть `null`
`void putAll(Map<K,V> m)`	копировать все пары из отображения `m` получателю
`void replaceAll(BiFunction<K,V,V> remapper)`	заменить каждое значение результатом вызова `remapper`

Удаление записей

`void clear()`	удалить все записи из отображения
`V remove(Object key)`	удалить запись с заданным ключом, если она присутствует; вернуть значение, с которым был ассоциирован ключ, или `null`, если запись не найдена

Опрос содержимого отображения

`V get(Object k)`	вернуть значение, соответствующее `k`, или `null`, если ключ `k` не найден
`boolean containsKey(Object k)`	вернуть `true`, если ключ `k` присутствует
`boolean containsValue(Object v)`	вернуть `true`, если значение `v` присутствует
`int size()`	вернуть число записей
`boolean isEmpty()`	вернуть `true`, если нет ни одной записи

Контракты методов `put`, `remove` и `get` проблематичны, когда отображение допускает значения `null`: возврат `null` для заданного ключа может означать как то, что с этим ключом ассоциировано значение `null`, так и то, что этот ключ отсутствует. Для различения этих ситуаций можно использовать метод

containsKey. Обычно эта проблема возникает только для неконкурентных отображений, потому что конкурентные, как правило, не допускают null. Если вы хотите производить запись в отображение, допускающее null, в зависимости от того, присутствует ключ или отсутствует, то может быть полезен один из составных методов (см. раздел «Составные операции» ниже). Однако в случае неконкурентных отображений нельзя полагаться на их атомарность, поэтому может показаться, что они ведут себя не согласованно, когда применяются к такому отображению в условиях конкурентного доступа. Это проблема TOCTOU, обсуждаемая в разделе «Безопасное использование синхронизированных коллекций» главы 17. Проектное решение допустить значения null в отображениях оценивается в разделе «Значения null» главы 18.

Получение представлений ключей, значений и записей

`Set<Map.Entry<K,V>> entrySet()`	вернуть представление множества записей
`Set<K> keySet()`	вернуть представление множества ключей
`Collection<V> values()`	вернуть представление коллекции значений

За коллекциями, возвращаемыми этими методами под видом представлений, стоит отображение, поэтому произведенные в них изменения отражаются на этом отображении. Связь в обратном направлении более ограничена: удалить элементы из представлений можно, но попытка добавить элемент приведет к исключению UnsupportedOperationException. Удаление ключа из множества ключей приводит к удалению единственной соответствующей ему пары ключ–значение. С другой стороны, удаление значения из коллекции, возвращенной методом values, приводит к удалению только одной из пар, где это значение встречается; то же значение, ассоциированное с другим ключом, может остаться. Итератор по представлению становится неопределенным, если базовое отображение конкурентно модифицировано.

Составные операции

Поскольку отображения очень часто используются в многопоточной среде, в интерфейсе присутствуют разнообразные составные действия. Они объединяют проверку условия – присутствует или отсутствует ключ, возможно, с конкретным значением – либо с переданным значением, либо с действием, представленным лямбда-выражением, для ленивого вычисления значения. Основное назначение этих методов – выполнение условных транзакций с состоянием конкурентного отображения. Некоторые методы из этой группы первоначально входили в состав интерфейса ConcurrentMap, появившегося в пакете java.util.concurrent. Впоследствии добавление методов по умолчанию в Java 8 позволило расширить интерфейс Map, включив их в него, и новые методы были добавлены как Map, так и в ConcurrentMap.

Эти методы можно отнести к двум группам в зависимости от стиля блокировки: пессимистического или оптимистического (см. раздел «Механизмы работы конкурентных коллекций» главы 9).

Пессимистические атомарные операции

Они необходимы в многопоточной среде, чтобы избежать небезопасных последовательностей операций типа «проверить, затем действовать», о которых мы говорили в разделе «Синхронизированные коллекции и итераторы с быстрым отказом» главы 9. Кроме того, они очень полезны при работе с потоконебезопасными отображениями.

`V getOrDefault(Object k, V defaultValue)`	вернуть значение, ассоциированное с ключом `k`, или `defaultValue`, если с этим ключом не ассоциировано никакого значения

Если в отображении значения ассоциированы только с некоторыми ключами, то метод `getOrDefault` позволяет использовать значение по умолчанию для всех остальных ключей, не сохраняя его в отображении.

`V putIfAbsent(K key, V value)`	создать пару ключ–значение, если ключ отсутствует или ассоциирован со значением `null`; в этих случаях вернуть `null`, а иначе – текущее значение

Метод `putIfAbsent` полезен, когда вы хотите записать что-то при первом появлении и больше не изменять. Например, чтобы сохранить временную метку первого появления события определенного вида, можно написать:

```
Map<EventKind,Long> firstOccurrenceMap = ... ;
...
firstOccurrenceMap.putIfAbsent(event.getKind(), System.currentTimeMillis());
```

В этом примере накладные расходы на упаковку временной метки в `Long` приходится нести для каждого события, а не только для первого. Обычно мы хотим избежать таких затрат, воспользовавшись другим составным методом, `computeIfAbsent`, который вычисляет новое значение лениво:

```
firstOccurrenceMap.computeIfAbsent(event.getKind(),
    key -> System.currentTimeMillis());
```

Метод `computeIfAbsent` объявлен следующим образом:

`V computeIfAbsent(K k, Function<K,V> mapper)`	применить функцию `mapper` к `k` и использовать результат для создания пары ключ–значение, если только `k` в данный момент не отображен на значение, отличное от `null`, и результат не равен `null`; вернуть значение, которое теперь ассоциировано с `k`, или `null`, если `k` отсутствует в отображении

В документации `computeIfAbsent` приводится пример наиболее типичного использования – создание отображения ключа на список (или другую совокупность) нескольких значений:

```
map.computeIfAbsent(key, k -> new ArrayList<V>()).add(newValue);
```

Есть и другой сценарий: аккумулирование значений в отображении посредством оператора, например сложения или конкатенации, который комбини-

рует старое и вновь переданное значение для сохранения в записи с данным ключом. Самый полезный метод в этой ситуации – `merge`:

`V merge(K k, V newValue, BiFunction<V,V,V> remapper)`	если ключ `k` отсутствует или отображен на `null`, то ассоциировать с ним `newValue`; в противном случае применить `remapper` к `newValue` и ассоциировать `k` с результатом, если тот отличен от `null`, а иначе удалить `k`

В документации `merge` приводится следующий пример, где значение записи либо инициализируется заданной строкой `msg`, либо эта строка дописывается в конец существующего значения:

```
map.merge(key, msg, String::concat);
```

Следующий метод, `compute`, похож на `merge`, но, вместо того чтобы передавать значение `newValue`, он лениво вычисляет его с использованием ключа:

`V compute(K k, BiFunction<K,V,V> remapper)`	использовать результат, вычисленный `remapper` по `k` и существующему значению (или `null`, если `k` отсутствует), для создания или модификации пары ключ–значение и вернуть результат; если вычисленный результат равен `null`, то удалить существующую запись и вернуть `null`

Например, если мы хотим отобразить каждое слово текста на количество символов во всех его вхождениях в текст, то можем использовать такой код:

```
map.compute(word, (s,i) -> s.length() + (i == null ? 0 : i));
```

Последний член семейства `compute*` – метод `computeIfPresent`:

`V computeIfPresent(K k, BiFunction<K,V,V> remapper)`	если `k` отображен на `null`, то вернуть `null`; в противном случае использовать результат, вычисленный `remapper` по `k` и существующему значению, для замены этого значения и вернуть результат; если вычисленный результат равен `null`, то удалить существующую запись и вернуть `null`

Точно так же как метод `computeIfAbsent` наиболее полезен, когда нужно добавить новый ключ, метод `computeIfPresent` можно использовать для удаления существующего ключа. В примере выше мы видели, как использовать `putIfAbsent`, чтобы создать отображение видов событий на временную метку первого появления события данного вида. Теперь предположим, что далее мы хотим обработать другой этап потока событий, так чтобы первое – и только первое – событие каждого вида инициировало запись в журнал с ранее запомненной временной меткой. Это можно сделать следующим образом:

```
firstOccurrenceMap.computeIfPresent(event.getKind(),(kind, timestamp) -> {
  log.info("first occurrence of event " + kind + " was at " + timestamp);
  return null;
});
```

Возврат `null` из лямбда-выражения гарантирует, что ключ будет удален из отображения, и потому запись сообщения в журнал будет произведена только один раз для события каждого вида.

Оптимистические атомарные операции

`V replace(K k, V newValue)`	заменить существующее значение, ассоциированное с `k`, при условии, что `k` присутствует в отображении; вернуть старое значение или `null`, если `k` отсутствовал
`boolean replace(K k, V oldValue, V newValue)`	заменить существующее значение, ассоциированное с `k`, при условии, что `k` отображен на `oldValue`; вернуть `true`, если `oldValue` было заменено
`boolean remove(Object key, Object value)`	удалить запись с этим ключом, если он отображен на это значение, и вернуть `true` в случае успеха

Эти методы условного обновления поддерживают оптимистический стиль конкурентной блокировки (см. раздел «Механизмы работы конкурентных коллекций» главы 9). Они вряд ли найдут широкое применение в неконкурентных системах, но в некоторых приложениях могут быть удобны. Например, вызывающая сторона может взять значение ключа `A` и работать над задачей исходя из этого значения. По завершении задачи вызывающая сторона хочет обновить значение, сделав его равным `B`. Однако другая конкурентная вызывающая сторона могла также взять значение `A` и начать работать, а в конце заменить его значением `C`; таким образом, налицо гонка между обновлениями.

Чтобы избежать гонки обновлений, вызывающие стороны могут использовать метод `replace(K,V,V)`. Он производит обновление и возвращает `true`, только когда текущее значение в отображении совпадает с первым аргументом. В противном случае метод не делает ничего и возвращает `false`. Предполагается, что вызывающая сторона передаст ожидаемое значение в первом аргументе. Та сторона, которая «выиграла» гонку, увидит первоначальное значение, которое и ожидала увидеть, и выполнит свое обновление. «Проигравшая» сторона увидит другое значение, обновление не будет выполнено, и метод вернет `false`. Ожидается, что в этот момент «проигравшая» сторона повторит свою операцию, быть может, пожертвовав работой, которую, возможно, успела выполнить, исходя из первоначального значения.

В сочетании эти методы позволяют нескольким конкурентным сторонам перемещать пару ключ–значение по конечному автомату, производя оптимистические вычисления, но не чураясь отбрасывать зря вычисленные результаты. Метод `replace(K,V)` производит безусловное изменение состояния, модифицируя существующую запись без риска создать новую. (Его дополнением является `putIfAbsent`, который не модифицирует существующую запись, а только создает новую.) Метод `remove(K,V)` производит условный переход в терминальное состояние, в котором запись удаляется.

Фабричные методы

Эти методы создают немодифицируемые объекты `Map`. Подробно они рассматриваются в разделе «UnmodifiableMap» ниже в этой главе.

Интерфейс Map.Entry

Элементы множества, возвращенного методом `entrySet`, реализуют интерфейс `Map.Entry`, представляющий пару ключ–значение. Этот интерфейс объявляет фабричные методы для создания компараторов по ключу и значению и методы экземпляра для доступа к компонентам записи. Факультативный метод `setValue` можно использовать для изменения значения в записи, если базовое отображение модифицируемо, и если это так, то изменения будут отражены в нем. Согласно документации, объект `Map.Entry`, полученный путем обхода множества, возвращенного `entrySet`, сохраняет связь с ним только на время обхода, но на самом деле это лишь гарантированный минимум: поведение реализации может отличаться, и некоторые сохраняют связь неопределенно долго. Однако современные идиомы обработки отображений в меньшей степени, чем раньше, зависят от долговечных объектов `Map.Entry`: если в прошлом мы могли бы использовать итератор множества `Map.Entry` для удаления некоторых записей, то теперь мы, скорее, вызовем метод `entrySet.removeIf`. Альтернативно и только для случаев, когда мы хотим изменить значения в парах, методы экземпляра `Map`, перечисленные в предыдущем разделе, предлагают различные способы удаления записей или изменения значений в них.

Для создания объектов `Map.Entry` можно использовать фабричный метод `Map.entry`; чаще всего он применяется для создания немодифицируемых отображений, обсуждаемых в разделе «UnmodifiableMap» ниже.

Использование методов Map

Одна из проблем, вызванных использованием очередей с приоритетами в нашем менеджере задач, заключается в том, что такие очереди не способны сохранить порядок добавления пар (разве что он включен в сами приоритеты, например в виде временной метки или порядкового номера). Чтобы избежать этой проблемы, мы могли бы использовать в качестве альтернативной модели ряд FIFO-очередей, каждой из которых назначен один общий приоритет. Для хранения ассоциации между приоритетами и очередями задач можно было бы использовать отображение, например `EnumMap`, очень эффективную реализацию `Map`, специализированную для ключей, являющихся элементами перечисления.

Эта модель опирается на реализацию интерфейса `Queue`, сохраняющую FIFO-порядок. Чтобы сосредоточиться на использовании методов `Map`, ограничимся однопоточным клиентом и будем использовать ряд очередей `ArrayDeque` в качестве реализации.

org/jgcbook/chapter15/C_using_the_methods_of_map/Program_1

```
var taskMap = new EnumMap<Priority,Queue<Task>>(Priority.class);
for (Priority p : Priority.values()) {
  taskMap.put(p, new ArrayDeque<Task>());
}
// заполнить очереди, например:
taskMap.get(Priority.MEDIUM).add(new PhoneTask("Mike", "987 6543"));
taskMap.get(Priority.HIGH).add(new CodingTask("db"));
```

Теперь, чтобы добраться до одной из очередей задач, например с наивысшим приоритетом, мы можем написать:

org/jgcbook/chapter15/C_using_the_methods_of_map/Program_1

```
Queue<Task> highPriorityTaskList = taskMap.get(Priority.HIGH);
```

Выборка из этой очереди будет возвращать задачи с высшим приоритетом в том порядке, в каком они добавлялись в систему.

Чтобы посмотреть, как используются другие методы `Map`, немного расширим пример, допустив, что некоторые задачи могут приносить нам доход, поскольку за их выполнение выставляется счет. Один из способов представить эту идею – определить класс `Client`:

```
class Client {...}
Client acme = new Client("Acme Corp.", ...);
```

и создать отображение задач на клиенты:

```
Map<Task,Client> billingMap = new HashMap<>();
billingMap.put(interfaceCode, acme);
```

Только оплачиваемые задачи попадают в отображение `billingMap`. Теперь обработка задачи `t` могла бы включать такой код:

```
Task t = ...
Client client = billingMap.get(t);
if (client != null) {
  client.bill(t);
}
```

Когда мы, наконец, закончим всю работу, оплаченную клиентом Acme Corp., можно удалить записи, которые связывают задачи с Acme. Сделать это можно разными способами, но, пожалуй, самый понятный – обойти все записи отображения, проверяя для каждой хранящееся в ней значение:

```
billingMap.entrySet().removeIf(e -> e.getValue().equals("acme"))
```

Представление `values` отображения `billingMap` может содержать дубликаты, если с одним клиентом было ассоциировано несколько задач. Если мы хотим найти множество клиентов, которым следует выставить счет, то можем скопировать данные из представления `values`, устранив дубликаты, с помощью конструктора `Set`:

```
Set<Client> billedClients = new HashSet<>(billingMap.values());
```

Развивая пример, предположим, что мы создали множество задач, которые можно выполнить, только находясь на территории клиента:

```
Set<Task> onsiteTasks = ...
```

Теперь мы хотим определить, для каких клиентов имеются оплачиваемые локальные задачи, чтобы можно было запланировать командировку. Конечно, можно было бы построить множество таких клиентов, обойдя множество записей в `billingMap` и оставив только тех клиентов, для которых соответствующая задача встречается в `onsiteTasks`. Однако более краткой и современной идиомой является копирование `billingMap` с применением деструктивной массовой операции `retainAll` к множеству ключей и последующим устранением дубликатов:

```
Map<Task,Client> onsiteMap = new HashMap<>(billingMap);
onsiteMap.keySet().retainAll(onsiteTasks);
Set<Client> clientsToVisit = new HashSet<>(onsiteMap.values());
```

Потоковая альтернатива

Потоковая обработка множества записей – также распространенная идиома. Например, можно было бы создать отображение, которое ассоциирует с каждым клиентом список задач, выполняемых на его территории:

```
Map<Client,List<Task>> tasksByClient = billingMap.entrySet().stream()
    .filter(entry -> onsiteTasks.contains(entry.getKey()))
    .collect(Collectors.groupingBy(Map.Entry::getKey));
```

Реализации интерфейса Map

На рис. 15.1 показаны все девять реализаций интерфейса `Map`, предоставляемых каркасом коллекций. Здесь мы обсудим классы `HashMap`, `WeakHashMap`, `IdentityHashMap`, `EnumMap`, а также семейство *UnmodifiableMap*; класс `LinkedHashMap` и интерфейсы `NavigableMap`, `SequencedMap`, `ConcurrentMap` и `ConcurrentNavigableMap` вместе с их реализациями обсуждаются в последующих разделах.

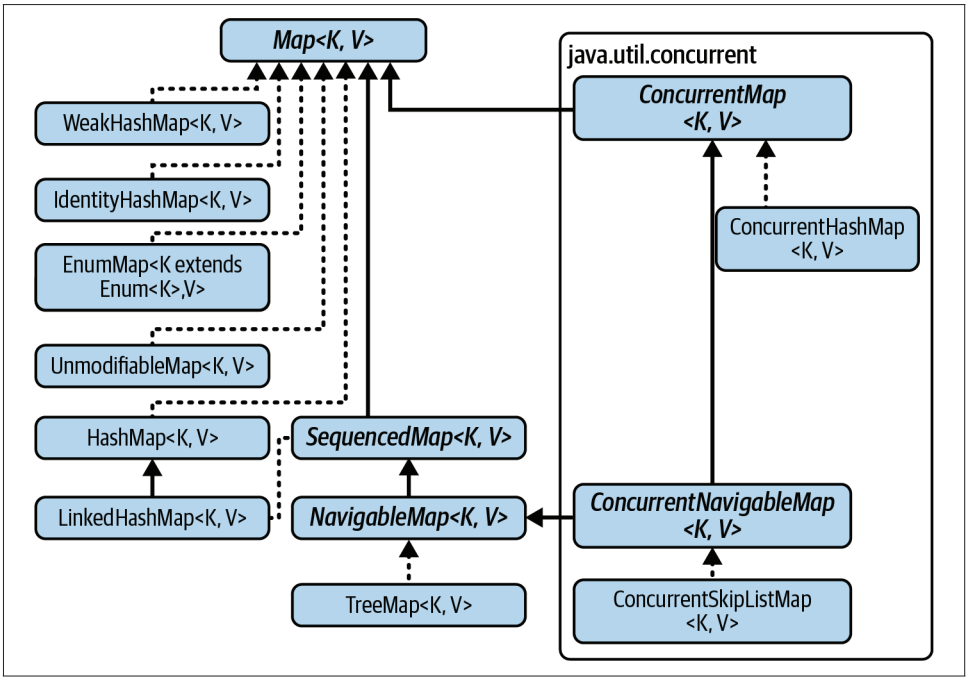


Рис. 15.1. Структура реализаций `Map` в каркасе коллекций

Что касается конструкторов, то общее правило для реализаций `Map` такое же, как для реализаций `Collection` (см. раздел «Конструкторы коллекций» главы 10). В каждой реализации, кроме `EnumMap`, имеется два конструктора. Например, для `HashMap` это:

```
public HashMap()
public HashMap(Map<? extends K,? extends V> m)
```

Первый создает пустое отображение, а второй – отображение, которое содержит пары ключ–значение, содержащиеся в переданном отображении `m`. Ключи и значения отображения `m` должны иметь типы, совпадающие или являющиеся подтипами типов ключей и значений создаваемого отображения. Эффект второго конструктора такой же, как после создания пустого отображения с помощью конструктора по умолчанию и последующего добавления содержимого `m` методом `putAll`. Помимо этих двух конструкторов, в классах `HashMap`, `LinkedHashMap` и `WeakHashMap` имеются другие конструкторы для конфигурирования, но на практике они были заменены фабричными методами по причинам, описанным в разделе «Конструкторы HashSet» главы 12.

HashMap

В разделе «HashSet» главы 12 мы говорили, что класс `HashSet` реализован с помощью специализированного `HashMap`; специализация заключается в том, что для всех ключей хранится одно и то же постоянное значение. Поэтому рис. 15.2a похож на рис. 12.1, но без упрощения, из-за которого все значения были убраны. Обсуждение хеш-таблиц и их производительности в разделе «Реализации интерфейса Set» главы 12 применимо также к `HashMap`. В частности, `HashMap` обеспечивает постоянную временную сложность методов `put` и `get`. Хотя теоретически постоянная сложность достижима только при условии отсутствия коллизий, ее можно хорошо аппроксимировать посредством перехеширования, минимизирующего число коллизий, при увеличении заполненности.

Для обхода коллекции ключей или значений требуется время, пропорциональное сумме емкости отображения и числа хранящихся в нем пар ключ-значение. Итераторы обладают свойством быстрого отказа.

Если вы можете предсказать максимальное число записей, то рекомендуется использовать фабричный метод `newHashMap`:

```
static <K,V> HashMap<K,V> newHashMap(int numElements)
```

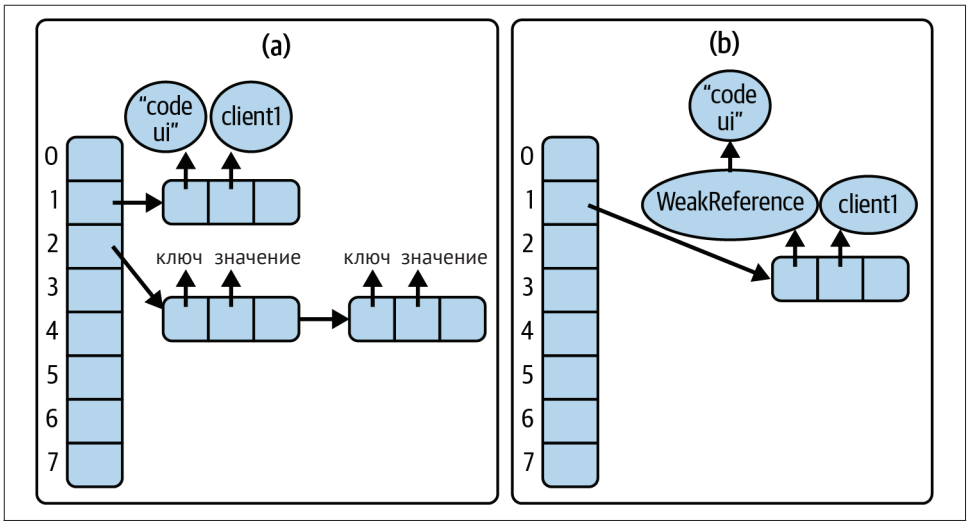


Рис. 15.2. (a) `HashMap` и (b) `WeakHashMap`

WeakHashMap

В большинстве отображений хранятся обыкновенные («сильные») ссылки на все содержащиеся в них объекты. Это означает, что даже когда ключ становится недостижим ни для кого, кроме самого отображения, ассоциированное с ним значение нельзя убрать в мусор. В примере в начале этой главы такая ситуация складывается с парами задача–клиент: они ссылаются на объекты задачи и клиента и оба занимают память, поэтому сохранение их без необходимости потенциально снижает производительность сборки мусора и создает утечку памяти. Идея класса `WeakHashMap` заключается в том, чтобы избежать такой ситуации с помощью объектов класса `java.lang.ref.WeakReference`. В классе `WeakReference` при доступе к объекту используется дополнительный уровень косвенности. Эта ситуация изображена на рис. 15.2b. Слабая ссылка не защищает объект от уборки в мусор; в данном случае если объект задачи «code ui» не достижим больше ниоткуда по нормальной ссылке, то он может стать жертвой сборщика мусора. Отображение замечает это и удаляет запись – со стороны будет казаться, что вся запись внезапно исчезла из отображения.

Для чего хорош класс `WeakHashMap`? Представьте, что ваша программа выделяет по запросу клиента какой-то временный системный ресурс, к примеру буфер. Помимо передачи ссылки на ресурс клиенту, программа, возможно, захочет сохранить информацию о нем локально, например ассоциировать буфер с запросившим его клиентом. Это можно было бы реализовать посредством отображения ресурсов на объекты клиентов. Но если использовать сильную ссылку, то даже после того как клиент освободил ресурс, в отображении будет храниться ссылка, мешающая убрать объект ресурса в мусор. Со временем память забивается ресурсами, которые больше никому не нужны. С другой стороны, если ссылка слабая и удерживается объектом `WeakHashMap`, то сборщик мусора сможет вернуть для последующего использования объекты, на которые не осталось сильных ссылок, и утечка памяти будет предотвращена.

Более общий случай встречается в приложениях, например кешах, где вы не возражаете против пропадания информации в условиях нехватки памяти. Для этой цели `WeakHashMap` подходит неидеально. Один из недостатков заключается в том, что слабые ссылки ведут на ключи отображения, а не на значения, которые обычно занимают гораздо больше памяти; поэтому даже после того как сборщик мусора освободит ключ, реального выигрыша в плане доступной памяти не случится, пока отображение не удалит неактуальную запись. Второй недостаток – тот факт, что слабые ссылки *слишком* слабы: сборщик мусора вправе убрать слабо достижимый объект в любой момент, и программист на это повлиять никак не может. (Родственник `WeakReference`, класс `java.lang.ref.SoftReference`, трактуется иначе: сборщик мусора откладывает уборку до тех пор, пока нехватка памяти не станет очень серьезной. В статье Heinz Kabutz (2004) написано основанное на `SoftReference` отображение, которое в качестве кеша лучше работает.)

По производительности `WeakHashMap` похож на `HashMap`, но чуть медленнее из-за накладных расходов на дополнительный уровень косвенности для ключей. Стоимость вычистки нежелательных пар ключ–значение перед каждой операцией пропорциональна числу пар, подлежащих удалению из-за того, что сборщик мусора убрал ключ. Итераторы по коллекциям ключей и значений, возвращаемые `WeakHashMap`, обладают свойством быстрого отказа.

Если вы можете предсказать максимальное число записей, то рекомендуется использовать фабричный метод `newWeakHashMap`:

```
static <K,V> WeakHashMap<K,V> newWeakHashMap(int numMappings)
```

IdentityHashMap

Отличительная особенность класса `IdentityHashMap` обсуждается в разделе «Определение множества: отношения эквивалентности» главы 12: в качестве отношения эквивалентности на множестве ключей в нем используется тождественное отношение. Иными словами, любой физически отличный объект является ключом.

`IdentityHashMap` отличается от обыкновенного `HashMap` тем, что два ключа считаются равными, если это физически один и тот же объект, т. е. для сравнения ключей используется тождественность, а не метод `equals`. В результате контракт класса `IdentityHashMap` противоречит контракту интерфейса `Map`, который он реализует, утверждающего, что для сравнения ключей следует использовать равенство. Проблема здесь аналогична той, с которой мы сталкивались при рассмотрении `Set`: в документации `Map` предполагается, что отношение эквивалентности всегда определяется методом `equals`, тогда как в `IdentityHashMap` фактически используется другое отношение, как и во всех реализациях `NavigableMap` (см. раздел «NavigableMap» ниже в этой главе).

Основная цель `IdentityHashMap` – поддержать операции, нуждающиеся в обходе графа и сохранении информации о каждой его вершине. Одна из таких операций – сериализация. Алгоритм, применяемый для обхода графа, должен уметь для каждой посещенной вершины проверять, встречалась ли эта вершина раньше, в противном случае он может бесконечно обходить один и от же цикл графа. Для циклических графов мы должны при проверке одинаковости вершин использовать тождественность, а не равенство. Вычисление равенства двух вершин графа требует вычисления равенства полей представляющих их объектов, что, в свою очередь, означает вычисление всех последующих вершин, – и мы возвращается к проблеме, с которой начали. Напротив, `IdentityHashMap` сообщает о том, что узел уже встречался, только если тот же самый узел ранее был помещен в отображение.

Стандартная реализация `IdentityHashMap` обрабатывает коллизии иначе, чем метод цепочек, показанный на рис. 12.1 и используемый во всех остальных вариантах `HashSet` и `HashMap`. В этой реализации применяется техника *линейного зондирования*, когда ссылки на ключ и значение хранятся в соседних ячейках самой таблицы, а не в позициях списка, на который ячейка ссылается. При обработке коллизии просто ищется свободная пара ячеек. На самом деле это по-прежнему цепочки, но физически они выглядят иначе. На рис. 15.3 показаны три этапа заполнения `IdentityHashMap` емкости 8. На рисунке (a) мы сохраняем пару ключ–значение, ключ которой хешируется в 0, а на рисунке (b) – пару, ключ которой хешируется в 4. Третий ключ, добавленный на рисунке (c), тоже хешируется в 4, поэтому алгоритм проходит по таблице, пока не найдет неиспользуемую ячейку. В данном случае первая же ячейка, с индексом 6, свободна, и ее можно использовать.

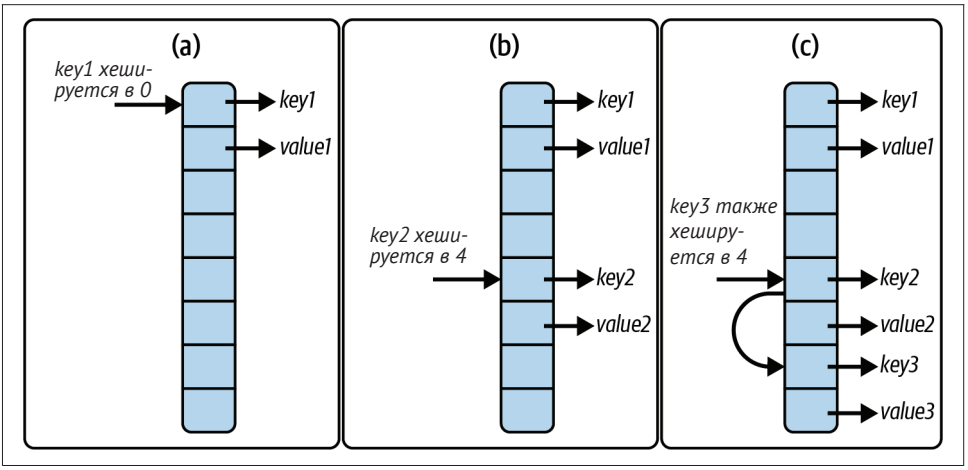


Рис. 15.3. Разрешение коллизий методом линейного зондирования

Удаление сложнее, чем в случае цепочек; если бы `key2` и `value2` были удалены из таблицы на рис. 15.3, то `key3` и `value3` нужно было бы переместить на их место. Для этого требуется копирование; альтернатива, при которой этого удается избежать, – заменить удаленную запись фиктивным элементом – *надгробным маркером*, который сохраняет цепочку, но может быть пропущен во время поиска и перезаписан для последующей вставки.

Почему в `IdentityHashMap` используется линейное зондирование, а `HashMap` и `ConcurrentHash` связанные цепочки? У каждой техники есть свои преимущества. Доступ к массиву по индексу, вообще говоря, быстрее поиска в связанном поиске, а его пространственная локальность лучше (см. раздел «Память» главы 9). Связанные цепочки более устойчивы к отказам в случае, когда несколько записей оказываются в одной ячейке, обычно из-за неудачно спроектированной функции хеширования. Этого преимущества лишен класс `IdentityHashMap`, поскольку используемый в нем хеш-код реализован в классе `Object` и не может быть изменен пользователем. Второе преимущество связанных цепочек в том, что звенья цепочки можно определить так, что в них будет храниться хеш-код полезной нагрузки, указатель на следующий элемент и сама полезная нагрузка. Так сделано из соображений эффективности: операция `get` должна проверить, что нашла правильный ключ, а, поскольку сравнение на равенство в общем случае дорогая операция, имеет смысл сначала проверить, а правилен ли хотя бы хеш-код. Но `IdentityHashCode` и этого преимущества лишен, потому что проверяет тождественность, а не равенство объектов, и потому хранить в таблице хеш-коды объектов нет нужды.

В классе `IdentityHashMap` три конструктора:

```
public IdentityHashMap()
public IdentityHashMap(Map<? extends K,? extends V> m)
public IdentityHashMap(int expectedMaxSize)
```

Первые два – стандартные конструкторы, присутствующие в любой реализации `Map` общего назначения. Третий занимает место двух конструкторов,

которые в других коллекциях на основе хеш-таблиц позволяют пользователю управлять начальной емкостью таблицы и коэффициентом заполненности, при котором ее пора перехешировать. `IdentityHashMap` этого не позволяет, а фиксирует его (в стандартной реализации он равен 0.67), чтобы оградить пользователя от последствий неправильного задания коэффициента. Дело в том, что если в таблице с цепочками переполнения стоимость нахождения ключа пропорциональна коэффициенту заполненности *l*, то в таблице с линейным зондированием она пропорциональна $1/(1 - l)$. Поэтому важно избегать больших коэффициентов заполненности – слишком важно, чтобы оставлять это на усмотрение программиста! Это решение согласуется с вышеупомянутой политикой более не предоставлять конфигурационных параметров в новых классах каркаса коллекций (см. раздел «Конструкторы HashSet» главы 12).

EnumMap

Реализация отображения типа перечисления прямолинейна и очень эффективна по причинам, схожим с описанными для класса `EnumSet` (см. раздел «EnumSet» главы 12). В реализации на основе массива порядковые номера элементов перечисления могут служить индексами соответствующих значений. Базовые операции `get` и `put` могут быть реализованы как доступ к массиву, требующий постоянного времени. Итератор по множеству ключей занимает время, пропорциональное количеству элементов перечисления, и возвращает ключи в их естественном порядке (в котором объявлены элементы). Итераторы по представлениям этого класса слабо согласованы.

В `EnumMap` определено три открытых конструктора:

```
EnumMap(Class<K> keyType)
EnumMap(EnumMap<K,? extends V> m)
EnumMap(Map<K,? extends V> m)
```

Первый создает пустое отображение, второй – отображение с таким же типом ключа и элементами, как у `m`. Третий конструктор создает `EnumMap`, используя элементы переданного `Map`, которое (если только это не `EnumMap`) должно содержать по меньшей мере одну запись.

Типобезопасность на этапе компиляции требует, чтобы `K` был типом перечисления. Это гарантируется объявлением класса `EnumMap`:

```
Class EnumMap<K extends Enum<K>,V>
```

Применение того же паттерна рекурсивного типа, что для класса `Enum` (см. раздел «Типы перечислений» главы 3) гарантирует, что ключ имеет правильный тип.

`EnumMap` содержит сбереженный тип ключа, который используется на этапе выполнения для проверки допустимости добавляемых в отображение записей. Этот тип поставляется тремя конструкторами по-разному. Первый поставляет его в виде объекта класса (см. раздел «Классная альтернатива» главы 5); во втором он копируется из заданного `EnumMap`. В третьем конструкторе есть две возможности: либо заданное `Map` в действительности является `EnumMap`, и тогда он ведет себя так же, как второй конструктор, либо используется класс первого ключа заданного `Map` (именно поэтому переданное отображение `Map` не может быть пустым).

UnmodifiableMap

Семейство реализаций, которое мы называем *UnmodifiableMap<K,V>*, – последние немодифицируемые коллекции, которые мы встретим в этой книге. У него много общего с родственными семействами, *UnmodifiableSet* и *UnmodifiableList*: ключи и значения нельзя добавлять, удалять и изменять, а значения `null` запрещены. Свойства членов этого семейства описаны в документации `Map`:

- они немодифицируемы: ключи и значения нельзя ни добавлять, ни удалять. Вызов любого изменяющего метод возбуждает исключение `UnsupportedOperationException`;
- они не допускают `null`. Попытка поместить в них при создании ключи или значения, равные `null`, приводит к исключению `NullPointerException`;
- они отвергают дубликаты ключей в момент создания. Если передать фабричному методу повторяющиеся ключи, то он возбудит исключение `IllegalArgumentException`.

Фабричные методы следуют образцу *UnmodifiableSet* и *UnmodifiableList*. Метод `of` имеет 11 перегруженных вариантов для создания немодифицируемых отображений, содержащих до 10 записей:

`<K,V> Map<K,V> of(K k1, V v1)`	возвращает немодифицируемое отображение, содержащее одну запись
`<K,V> Map<K,V> of(K k1, V v1, K k2, V v2)`	возвращает немодифицируемое отображение, содержащее две записи
`<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)`	возвращает немодифицируемое отображение, содержащее три записи
дополнительные перегруженные варианты с числом аргументов от 4 до 9	
`<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)`	возвращает немодифицируемое отображение, содержащее десять записей

Еще два метода поддерживают создание немодифицируемых отображений по любому отображению или по массиву либо списку записей переменной длины:

`<K,V> Map<K,V> ofEntries(Map.Entry<K,V>... entries)`	возвращает немодифицируемое отображение, ключи и значения которого получены из `entries`
`<K,V> Map<K,V> copyOf(Map<K,V> map)`	возвращает немодифицируемое отображение, содержащие записи заданного отображения `map`

Поскольку все vararg-аргументы должны быть одного типа, тогда как типы ключей и значений могут различаться, метод создания немодифицируемых отображений, получающий список аргументов переменной длины, должен принимать не ключи и значения, а элементы типа `Map.Entry`. Их можно создать статическим вспомогательным методом `Map::entry`.

Как и другие немодифицируемые коллекции, немодифицируемые отображения основаны на массивах фиксированной длины. Их преимущества перед

хешированными коллекциями – более низкое потребление памяти, ускоренное итерирование и улучшенная пространственная локальность. Если функция `hashCode` дает хорошее распределение, то поиск занимает время *O*(1). Как и для *UnmodifiableSet* и по той же причине, порядок обхода множества ключей или значений определен случайно для каждого экземпляра виртуальной машины.

SequencedMap

`SequencedMap` – это `Map`, записи которого расположены в определенном порядке. Положение `SequencedMap` в иерархии типов `Map` показано на рис. 15.4.

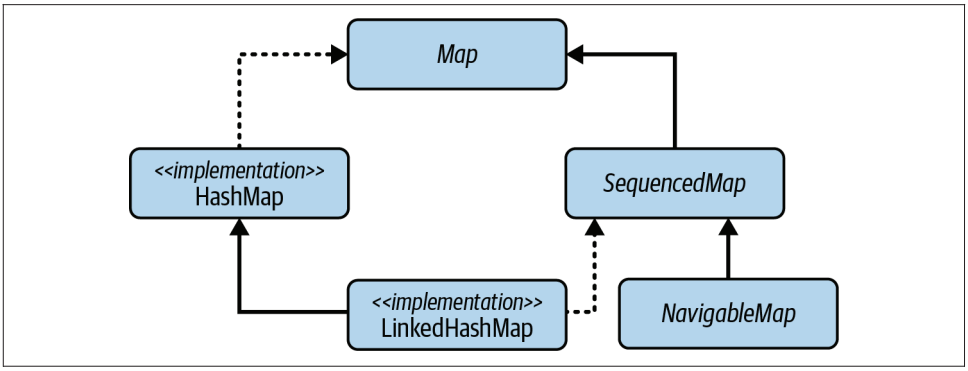


Рис. 15.4. `SequencedMap` и родственные типы

Методы SequencedMap

Этот интерфейс объявляет методы добавления и инспектирования первого и последнего ключа или значения, а также удаления первой или последней записи отображения.

Добавление и обновление записей

`V putFirst(K k, V v)`	вставить запись или обновить ее, если она уже присутствует
`V putLast(K k, V v)`	вставить запись или обновить ее, если она уже присутствует

Эти методы добавляют запись в начало или в конец `SequencedMap` соответственно или обновляют запись, если она уже присутствует. В случае внутренне упорядоченной реализации, например `TreeMap`, оба метода возбуждают исключение `UnsupportedOperationException`.

Контракты этих методов оговаривают, что после успешного выполнения отображение должно содержать переданную запись, которая должна быть первой (или последней) в порядке вхождения.

Инспектирование записей

`Map.Entry<K,V> firstEntry()`	вернуть первую запись или `null`, если отображение пусто
`Map.Entry<K,V> lastEntry()`	вернуть последнюю запись или `null`, если отображение пусто

Эти методы предназначены для инспектирования записей в начале или в конце `SequencedMap`.

Удаление записей

`Map.Entry<K,V> pollFirstEntry()`	вернуть первую запись или `null`, если отображение пусто
`Map.Entry<K,V> pollLastEntry()`	вернуть последнюю запись или `null`, если отображение пусто

Эти методы предназначены для удаления записей из начала или из конца `SequencedMap`.

Методы генерирования представлений

`SequencedMap<K,V> reversed()`	вернуть представление отображения в обратном порядке
`SequencedSet<Map.Entry<K,V>> sequencedEntrySet()`	вернуть представление `SequencedSet` множества записей отображения
`SequencedSet<K> sequencedkeySet()`	вернуть представление `SequencedSet` множества ключей отображения
`SequencedCollection<V> sequencedValues()`	вернуть представление `SequencedSet` коллекции значений отображения

Эти методы предоставляют различные представления отображения.

LinkedHashMap

Как и `LinkedHashSet` (см. раздел «LinkedHashSet» главы 12), класс `LinkedHashMap` уточняет контракт своего родительского класса, `HashMap`, гарантируя порядок возврата элементов итераторами. И так же, как `LinkedHashSet`, он реализует упорядоченный подынтерфейс (`SequencedMap`) своего основного интерфейса. Но, в отличие от `LinkedHashSet`, класс `LinkedHashMap` дает возможность выбрать порядок итерирования; элементы могут быть возвращены либо в том порядке, в каком вставлялись в отображение, – по умолчанию, – или в порядке, в каком к ним осуществлялся доступ (от самого недавнего к самому давнему). Объект `LinkedHashMap`, упорядоченный по времени доступа, создается путем передачи аргумента `true` в качестве последнего параметра конструктора:

```
public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
```

Если этот аргумент равен `false`, то создается отображение, упорядоченное по времени вставки. Другие конструкторы – такие же, как у `HashMap`, – тоже порождают отображения, упорядоченные по времени вставки. Как и в случае `LinkedHashSet`, обход `LinkedHashMap` занимает время, пропорциональное числу элементов в отображении, а не его емкости.

Отображения, упорядоченные по времени доступа, особенно полезны для конструирования кешей *с вытеснением по давности использования* (LRU). Кеш – это область памяти, в которой программа хранит особо востребованные данные для быстрого доступа. При проектировании кеша ключевой вопрос – выбор алгоритма, который будет решать, какие данные вытеснить (удалить из кеша),

чтобы ограничить занятую им память. Когда нужно найти кешированные данные, они сначала ищутся в кеше. Обычно если данные не найдены в кеше, то они извлекаются из основного хранилища и помещаются в кеш. Но кеш не может расти бесконечно, поэтому нужна стратегия удаления из кеша наименее полезного элемента при добавлении нового. Если такой стратегией является LRU, то удаляется элемент, к которому дольше всего не было обращений. Эта простая стратегия приемлема в ситуациях, когда доступ к элементу повышает вероятность повторного доступа к нему в ближайшем будущем. Простота и скорость работы сделали ее самой популярной стратегией кеширования. В классе `LinkedHashMap` имеется метод, специально предназначенный для того, чтобы упростить его использование в качестве LRU-кеша:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
```

Имя метода вводит в заблуждение. Обычно он сам не используется для модификации отображения, а вызывается из методов `put` или `putAll` при каждом добавлении элемента. Возвращенное им значение сообщает вызывающей стороне, должна ли она удалить из отображения первую запись, т. е. ту, к которой дольше всего не обращались (или, если к каким-то записям вообще никогда не обращались, ту из них, которая была помещена в кеш раньше других).

Реализация в `LinkedHashMap` просто возвращает `false` – указание вызывающей стороне, что никакое действие не требуется. Но вы можете создать подкласс `LinkedHashMap` и переопределить `removeEldestEntry`, так что при определенных обстоятельствах он будет возвращать `true`. В документации приведен пример отображения, которое не должно расти сверх заданного размера.

org/jgcbook/chapter15/F_linkedhashmap/BoundedSizeMap

```
class BoundedSizeMap<K,V> extends LinkedHashMap<K,V> {
  private final int maxEntries;
  public BoundedSizeMap(int maxEntries) {
    super(16, 0.75f, true);
    this.maxEntries = maxEntries;
  }
  protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return size() > maxEntries;
  }
}
```

Поскольку в упорядоченном по времени вставки отображении `LinkedHashMap` самой старой записью будет та, которая была добавлена раньше других, показанное выше переопределение `removeEldestEntry` реализует стратегию FIFO. FIFO-кешированию часто отдавали предпочтение перед LRU, поскольку его гораздо проще реализовать в отображениях, не предлагающих упорядочение по времени доступа. Однако LRU обычно эффективнее FIFO, потому что уменьшение затрат на обновление кеша перевешивает накладные расходы на поддержание упорядочения по времени доступа.

Если вы хотите модифицировать весьма специфическое действие, для которого специализирован метод `removeEldestEntry`, то можете просто рассматривать его как обратный вызов, добавив нужное вам действие и вернув `false`. Например, если бы метод `isReservedKey` возвращал `true` для элементов, которые никогда не должны вытесняться из кеша, то `removeEldestEntry` можно было бы реализовать так:

org/jgcbook/chapter15/F_linkedhashmap/BoundedSizeMap_1

```
private boolean isReservedKey(K keyToRetain) { ... }
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
  if (size() > maxEntries) {
    for (var itr = entrySet().iterator() ; itr.hasNext() ; ) {
      if (!isReservedKey(itr.next().getKey())) {
        itr.remove();
        return false;
      }
    }
  }
  return false;
}
```

Начиная с Java 21, `LinkedHashMap` реализует интерфейс `SequencedMap`, что открывает гораздо больше возможностей для выбора политики кеширования. Например, в некоторых приложениях факт недавнего доступа к записи понижает, а не повышает вероятность повторного обращения к ней в ближайшем будущем. В таком случае наилучшей является стратегия *с вытеснением последнего использованного* (MRU), когда удаляется запись, использовавшаяся позже других. Ее очень просто реализовать в отображении `SequencedMap`, поскольку оно предлагает метод `pollLastEntry`, аналогичный `SequencedSet.removeLast`:

org/jgcbook/chapter15/F_linkedhashmap/BoundedSizeMap_2

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
  if (size() > maxEntries) {
    pollLastEntry();
  }
  return false;
}
```

Продолжая развитие этого примера, мы можем объединить оба предыдущих варианта в ограниченный MRU-кеш зарезервированных имен, воспользовавшись методом `sequencedKeySet`, который возвращает представление множества ключей в виде `SequencedSet` (см. раздел «SequencedSet» главы 12):

org/jgcbook/chapter15/F_linkedhashmap/BoundedSizeMap_3

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
  if (size() > maxEntries) {
     for (var itr = sequencedKeySet().reversed().iterator() ; itr.hasNext() ;
) {
      if (!isReservedKey(itr.next())) {
        itr.remove();
        return false;
      }
    }
  }
  return false;
}
```

Эти примеры можно считать реалистичными разве что в самых простых случаях; если вы захотите создать подкласс `LinkedHashMap` в реальном приложении, то, скорее всего, потребуется больше гибкости, чем может дать переопределение `removeEldestEntry`. В таком случае было бы разумнее переопределить `put`

и putAll или предоставить другие методы, позволяющие пользователю модифицировать кеш, например, с помощью методов putFirst или putLast интерфейса SequencedMap с целью переместить записи в начало или в конец отображения.

Временная сложность обхода коллекции ключей или значений, возвращенной LinkedHashMap, линейно зависит от числа элементов. Итераторы таких коллекций обладают свойством быстрого отказа.

Если вы можете предсказать максимальное число записей, то рекомендуется использовать фабричный метод newLinkedHashMap:

```
static <K,V> LinkedHashMap<K,V> newLinkedHashMap(int numMappings)
```

NavigableMap

Интерфейс NavigableMap добавляет SequencedMap гарантию того, что итератор будет обходить отображение в порядке возрастания ключей и, как и NavigableSet, добавляет дополнительные методы для нахождения записей, соседних с целевым значением ключа. И, подобно NavigableSet, NavigableMap расширяет и, по существу, заменяет прежний интерфейс SortedMap, который индуцирует отношение порядка на своих ключах: либо естественного, либо определенного компаратором. Отношение эквивалентности на ключах NavigableMap также определяется отношением порядка; два равных ключа, т. е. таких, что метод сравнения возвращает для них 0, с точки зрения NavigableMap являются дубликатами (см. раздел «Определение множества: отношения эквивалентности» главы 12).

Дополнительные методы, объявленные в интерфейсе NavigableMap, можно отнести к четырем группам.

Получение компаратора

`Comparator<? super K> comparator()`	вернуть компаратор ключей отображения, если он был предоставлен взамен естественного порядка ключей; в противном случае вернуть null

Этот метод может быть полезен при переносе данных из одной отсортированной коллекции в другую; если компараторы, использованные для их сортировки, одинаковы, то никакой пересортировки не требуется.

Получение диапазонных представлений

`SortedMap<K,V> subMap(K fromKey, K toKey)`	вернуть представление части отображения, для которой ключи находятся в диапазоне от fromKey (включая) до toKey (не включая)
`SortedMap<K,V> headMap(K toKey)`	вернуть представление части отображения, для которой ключи строго меньше toKey
`SortedMap<K,V> tailMap(K fromKey)`	вернуть представление части отображения, для которой ключи больше или равны fromKey
`NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)`	вернуть представление части отображения, для которой ключи находятся в диапазоне от fromKey до toKey

`NavigableMap<K,V> headMap(K toKey, boolean inclusive)`	вернуть представление части отображения, для которой ключи меньше (или равны, если `inclusive` равно `true`) `toKey`
`NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)`	вернуть представление части отображения, для которой ключи больше (или равны, если `inclusive` равно `true`) `fromKey`

Как и в случае `NavigableSet`, для каждого метода этой группы имеется два перегруженных варианта: первый унаследован от `SortedMap` и возвращает полуоткрытое представление типа `SortedMap`, а второй определен в `NavigableMap` и возвращает представление типа `NavigableSet`, которое может быть открытым, полуоткрытым или замкнутым по выбору пользователя. Эти операции работают так же, как соответствующие им в `SortedSet`: аргументы `key` не обязаны присутствовать в отображении, а возвращенные множества включают `fromKey` (если он присутствует), но не включают `toKey`.

Как и методы `NavigableSet`, методы `NavigableMap` предлагают большую гибкость, чем методы диапазонных представлений `SortedMap`. Вместо того чтобы всегда возвращать полуоткрытый интервал, они принимают булевы параметры, которые определяют, включать ли в интервал ключ или ключи, его определяющие.

Получение ближайших соседей

`Map.Entry<K,V> ceilingEntry(K Key)`	вернуть пару ключ–значение, ассоциированную с наименьшим ключом, большим или равным заданному, или `null`, если такого ключа не существует
`K ceilingKey(K Key)`	вернуть наименьший ключ, больший или равный заданному, или `null`, если такого ключа не существует
`Map.Entry<K,V> floorEntry(K Key)`	вернуть пару ключ–значение, ассоциированную с наибольшим ключом, меньшим или равным заданному, или `null`, если такого ключа не существует
`K floorKey(K Key)`	вернуть наибольший ключ, меньший или равный заданному, или `null`, если такого ключа не существует
`Map.Entry<K,V> higherEntry(K Key)`	вернуть пару ключ–значение, ассоциированную с наименьшим ключом, строго большим заданного, или `null`, если такого ключа не существует
`K higherKey(K Key)`	вернуть наименьший ключ, строго больший заданного, или `null`, если такого ключа не существует
`Map.Entry<K,V> lowerEntry(K Key)`	вернуть пару ключ–значение, ассоциированную с наибольшим ключом, строго меньшим заданного, или `null`, если такого ключа не существует
`K lowerKey(K Key)`	вернуть наибольший ключ, строго меньший заданного, или `null`, если такого ключа не существует

Эти методы похожи на соответствующие методы `NavigableSet`, но возвращают объекты типа `Map.Entry`. Если вам нужен ключ, принадлежащий одной из

этих записей, воспользуйтесь соответствующим методом получения ключа – при этом вы сэкономите на создании ненужного объекта `Map.Entry`.

Другие представления

`NavigableMap<K,V> descendingMap()`	вернуть представление отображения в обратном порядке
`NavigableSet<K> descendingKeySet()`	вернуть множество ключей в обратном порядке
`NavigableSet<K> navigableKeySet()`	вернуть множество ключей в прямом порядке

Метод `reversed` интерфейса `SequencedMap` унаследован `NavigableMap`, но ковариантная перегрузка позволяет вернуть `NavigableMap`. Это синоним метода `descendingMap`, который остался по историческим причинам, поскольку существовал еще до того, как `NavigableMap` был задним числом переделан, чтобы наследовать `SequencedMap`. Аналогично метод `descendingKeySet` делает то же самое, что вызов `reversed` для множества ключей `SequencedMap`, но опять-таки возвращает более точный тип `NavigableSet`.

Может возникнуть вопрос, почему метод `keySet`, унаследованный от `Map`, нельзя было просто переопределить, так чтобы он ковариантно возвращал `NavigableSet`. На самом деле платформенные реализации `NavigableMap::keySet` действительно возвращают `NavigableSet`. Но имеется проблема совместимости: если бы `TreeMap::keySet` должен был изменить тип своего возвращаемого значения с `Set` на `NavigableSet`, то все существующие подклассы `TreeMap`, в которых этот метод переопределен, перестали бы компилироваться, пока их возвращаемый тип не изменят. (Эта проблема похожа на обсуждаемые в статье «Maintain Binary Compatibility» по адресу *https://mauricenaftalin.github.io/JGC_2e_Book_Code/ maintain_binary_compatibility.html*.)

TreeMap

Интерфейс `NavigableMap` в каркасе коллекций реализован классом `TreeMap`. Мы встречались с деревьями как структурой данных для хранения элементов в определенном порядке, когда обсуждали класс `TreeSet` (см. раздел «TreeSet» главы 12). На самом деле внутреннее представление `TreeSet` – это как раз `TreeMap`, в котором со всеми ключами ассоциировано одно и то же стандартное значение, поэтому приведенное там объяснение механизма и производительности красно-черных деревьев применимо и здесь.

Помимо стандартных, `TreeMap` имеет конструктор, который позволяет передать `Comparator`, и еще один, позволяющий создать `TreeMap` из другого объекта типа `NavigableMap` (строго говоря, типа `SortedMap`), сохранив его компаратор и все записи:

```
public TreeMap(Comparator<? super K> comparator)
public TreeMap(SortedMap<K, ? extends V> m)
```

Отметим, что второй из этих конструкторов, определенный так, чтобы новое отображение перенимало порядок переданного, страдает той же бедой, что соответствующий конструктор `TreeSet`: стандартный преобразующий конструктор – тот, что принимает `Map` в качестве аргумента, – всегда пользуется

естественным порядком ключей, поэтому если передать ссылку на `SortedMap` преобразующему конструктору `TreeMap`, то порядок сконструированного отображения будет зависеть от статического типа этой ссылки.

Характеристики производительности `TreeMap` схожи с `TreeSet`: базовые операции (`get`, `put` и `remove`) выполняются за время $O(\log N)$. Итераторы представлений коллекции обладают свойствами быстрого отказа.

ConcurrentMap

Отображения часто используются в высокопроизводительных серверных приложениях – например, в качестве реализаций кеша, – поэтому потокобезопасная реализация отображения с высокой пропускной способностью – неотъемлемая часть платформы Java. Этому требованию не могут удовлетворить синхронизированные отображения, подобные `Collections::synchronizedMap`, потому что при полной синхронизации каждая операция должна получать монопольную блокировку всего отображения, так что доступ к нему, по существу, сериализуется. Блокировка только части коллекции – *расслоение блокировки* (lock striping) – может дать очень большой выигрыш в производительности, как мы вскоре убедимся на примере `ConcurrentHashMap`. Но поскольку при этом не существует единственной блокировки, которую клиент мог бы захватить для получения монопольного доступа, блокировка на стороне клиента больше не работает, и клиенты нуждаются в помощи со стороны самой коллекции для выполнения атомарных действий.

В этом и состояла цель интерфейса `ConcurrentMap`, который появился вместе с другими конкурентными типами в Java 5. Он объявлял четыре метода: `putIfAbsent`, `remove` и два перегруженных варианта `replace`, которые выполняют составные операции атомарно. В версии Java 8 были введены новые составные операции (см. раздел «Составные операции» выше в этой главе), и в `ConcurrentMap` были включены для них реализации по умолчанию. Однако все четыре существующих составных метода были перенесены в интерфейс `Map`, так что `ConcurrentMap` больше не привносит никакой новой функциональности.

ConcurrentHashMap

Класс `ConcurrentHashMap` предоставляет реализацию `ConcurrentMap` и предлагает эффективное решение проблемы примирения пропускной способности с потокобезопасностью. Он оптимизирован для чтения, поэтому поиск больше не блокирует выполнение, даже если таблица одновременно обновляется. Чтобы это было возможно, в контракте оговорено, что результаты поиска отражают последние операции обновления, завершившиеся до начала поиска. Конкурентные обновления можно безопасно производить, даже когда таблица находится в процессе изменения размера.

Конструкторы `ConcurrentHashMap` похожи на конструкторы `HashMap`, но есть один дополнительный, которые позволяет программисту подсказать реализации ожидаемое число конкурентно обновляемых потоков (*уровень конкурентности*):

```
ConcurrentHashMap()
ConcurrentHashMap(int initialCapacity)
ConcurrentHashMap(int initialCapacity, float loadFactor)
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)
ConcurrentHashMap(Map<? extends K,? extends V> m)
```

`ConcurrentHashMap` – полезная реализация интерфейса `Map` в любом конкурентном приложении, где нет необходимости блокировать всю таблицу; это как раз та возможность синхронизированных отображений (см. раздел «Синхронизированные коллекции» главы 16), которую он не поддерживает. Это влияет на результаты обобщающих методов состояния, таких как `size`, `isEmpty` и `containsValue`. Отображение нельзя заблокировать целиком, поэтому если оно подвергается конкурентному обновлению, то будет изменяться, пока эти методы работают. В таком случае их результаты не будут отражать никакое согласованное состояние, и их следует рассматривать как аппроксимации.

Как и для других отображений, статический метод `Collections::newSetFromMap` можно использовать для создания конкурентного множества из `ConcurrentHashMap` с тем же порядком, свойствами конкурентности и производительностью (см. раздел «Представления отображений в виде множеств» главы 12). Кроме того, в `ConcurrentHashMap` имеется внутренний класс `ConcurrentHashMap.KeySetView`, который не только предоставляет операции над множествами, но и раскрывает методы, дающие доступ к базовому отображению, что позволяет управлять им через представление. Два из таких методов – перегруженные варианты `Map::keySet` с ковариантно переопределенным типом возвращаемого значения `KeySetView`. Еще два – перегруженные варианты фабричного метода `newKeySet`, создающего новое пустое отображение, которым можно управлять – даже добавлять пары ключ–значение – через возвращенное `KeySetView`.

Если не принимать во внимание накладные расходы на блокировку, то временная сложность операций `ConcurrentHashMap` такая же, как у `HashMap`. Представления коллекций возвращают слабо согласованные итераторы.

ConcurrentNavigableMap

Интерфейс `ConcurrentNavigableMap` наследует `ConcurrentMap` и `NavigableMap`. Он содержит только методы этих двух интерфейсов, в которые внесено несколько изменений, чтобы сделать возвращаемые типы более точными. Во-первых, методы диапазонных представлений, унаследованные от `SortedMap` и `NavigableMap`, теперь возвращают представления типа `ConcurrentNavigableMap`. Проблемы совместимости, не позволявшие `NavigableMap` переопределить методы `SortedMap`, отсутствуют при переопределении методов диапазонных представлений `NavigableMap` или `SortedMap`; поскольку ни один из них не имеет реализаций, перенесенных в новый интерфейс, то и опасности «поломать» подклассы реализации не возникает. По той же причине теперь можно переопределить `keySet`, так чтобы он возвращал `NavigableSet`.

ConcurrentSkipListMap

Связь между `ConcurrentSkipListMap` и `ConcurrentSkipListSet` такая же, как между `TreeMap` и `TreeSet`. `ConcurrentSkipListSet` реализован с помощью `ConcurrentSkipListMap`,

в котором с каждым ключом ассоциировано одно и то же стандартное значение, поэтому механизм работы и производительность реализации на основе списка с пропусками, приведенной в разделе «ConcurrentSkipListSet» главы 12, остаются такими же и здесь: базовые операции (`get`, `put` и `remove`) имеют сложность $O(\log N)$, а итераторы по представлениям коллекциям выполняют метод `next` за постоянное время. Эти итераторы слабо согласованы.

Сравнение реализаций отображения

В табл. 15.1 приведены данные о сравнительной производительности различных платформенных реализаций `Map` (в столбце «next» показана стоимость операции `next` итераторов по множеству ключей). Как и в случае реализаций очереди, на выбор класса отображения, скорее всего, в большей степени влияют функциональные требования приложения и свойства конкурентности, чем соображения производительности.

Таблица 15.1. Сравнительная производительность различных реализаций `Map`

	get	containsKey	next	Примечания
`HashMap`	$O(1)$	$O(1)$	$O(h/N)$	h – емкость таблицы
`WeakHashMap`	$O(1)$	$O(1)$	$O(h/N)$	h – емкость таблицы
`LinkedHashMap`	$O(1)$	$O(1)$	$O(1)$	
`IdentityHashMap`	$O(1)$	$O(1)$	$O(h/N)$	h – емкость таблицы
`EnumMap`	$O(1)$	$O(1)$	$O(1)$	
`TreeMap`	$O(\log N)$	$O(\log N)$	$O(\log N)$	
`ConcurrentHashMap`	$O(1)$	$O(1)$	$O(h/N)$	h – емкость таблицы
`ConcurrentSkipListMap`	$O(\log N)$	$O(\log N)$	$O(1)$	

Некоторые специальные ситуации диктуют конкретную реализацию. Для отображений перечисления всегда следует использовать класс `EnumMap` и только его. Такие задачи, как обход графов, описанный в разделе «IdentityHashMap» выше, требуют класса `IdentityHashMap`. Если нужно отсортированное отображение, то используйте `TreeMap`, когда потокобезопасность не требуется, и `ConcurrentSkipListMap` в противном случае.

Остается выбор реализации для отображений общего назначения. Для конкурентных приложений единственный вариант – `ConcurrentHashMap`. В остальных случаях отдавайте предпочтение `LinkedHashMap` перед `HashMap` (смиряясь с чуть более низкой производительностью), если собираетесь воспользоваться порядком отображения по времени вставки или доступа, например использовать его в качестве кеша.

Заключение

В этой главе мы изучили интерфейс `Map` и его подынтерфейс `ConcurrentMap`, который предоставляет лучше всего проработанный API конкурентных операций из имеющихся в каркасе коллекций. Реализации `ConcurrentMap` находят

широкое применение в высокопроизводительных приложениях масштаба предприятия.

На этом мы завершаем обзор основных интерфейсов и классов каркаса коллекций. Но это еще не все: многие из наиболее полезных алгоритмов каркаса реализованы статическими методами служебного класса `Collections`. По большей части они появились еще до того, как в Java 8 были включены статические методы в интерфейсах, но их сосредоточение в классе `Collections` удобно, потому что так их проще найти; это именно то место, куда нужно идти, когда нужна какая-то утилита для коллекций. Мы рассмотрим эту тему в следующей главе.

Глава 16

Класс Collections

Класс `java.util.Collections` содержит исключительно статические методы, которые обрабатывают или возвращают коллекции. Есть три основные категории: обобщенные алгоритмы, методы, возвращающие пустые или предварительно заполненные коллекции, и методы, создающие обертки. Мы обсудим их по очереди, а затем рассмотрим ряд других методов, которые невозможно отнести ни к одной категории.

Все методы класса `Collections` открытые и статические, поэтому для краткости мы будем опускать эти модификаторы в объявлениях.

Обобщенные алгоритмы

Обобщенные алгоритмы можно отнести к одной из четырех крупных категорий: изменению порядка элементов в списке, изменению содержимого списка, нахождению экстремальных значений в коллекции и нахождению конкретных значений в списке. Они представляют повторно используемую функциональность в том смысле, что применимы к спискам `List` (а в некоторых случаях к `Collection`) любого типа. Обобщение типов этих методов привело к довольно сложным объявлениям, поэтому в каждом разделе после описания методов кратко обсуждаются объявления.

Выбор конкретного алгоритма, применяемого к спискам, часто зависит от того, реализует ли обрабатываемый список маркерный интерфейс `RandomAccess`. Классы реализуют этот интерфейс, для того чтобы указать обобщенным методам, что длинный список этого класса будет эффективнее обработать методом `get`, чем с помощью итератора. Класс `ArrayList` реализует `RandomAccess`, а класс `LinkedList` – нет.

Изменение порядка элементов списка

`void reverse(List<?> list)`	изменить порядок элементов на противоположный
`void rotate(List<?> list, int distance)`	циклически переставить элементы списка; элемент с индексом `i` перемещается в позицию `(distance + i) % list.size()`
`void shuffle(List<?> list)`	случайным образом переставить элементы списка
`void shuffle(List<?> list, Random rnd)`	случайным образом переставить элементы списка, пользуясь источником случайности `rnd`

`void shuffle(List<?> list, RandomGenerator rndGen)`	случайным образом переставить элементы списка, пользуясь генератором случайности `rndGen`
`<T extends Comparable<? super T>> void sort(List<T> list)`	отсортировать список в естественном порядке
`<T> void sort(List<T> list, Comparator<? super T> c)`	отсортировать список в указанном порядке
`void swap(List<?> list, int i, int j)`	обменять местами элементы в указанных позициях

Самый простой из методов переупорядочения списка – `swap`, он переставляет местами два элемента и если `List` реализует интерфейс `RandomAccess`, то выполняется за постоянное время. Самый сложный метод – `sort`, который перемещает элементы списка в массив, сортирует их за время $O(N \log N)$ в худшем случае, а затем возвращает их в список. Все остальные методы выполняются за время $O(N)$.

Для каждого метода (кроме `sort` и `swap`) имеется два алгоритма: в одном используется `ListIterator`, а в другом `get` и `set`. Метод `sort` делегирует работу методу `List::sort`, реализация которого по умолчанию перемещает элементы списка в массив, хотя реализация в `ArrayList` переопределяет ее и сортирует список на месте. Затем к этому массиву применяется алгоритм стабильной сортировки – в случае JDK алгоритм timsort (*https://angelikalanger.com/GenericsFAQ/FAQSections/ TypeParameters.html#FAQ107*), который в худшем случае занимает время $O(N \log N)$. В методе `swap` всегда используется произвольный доступ. Стандартные реализации прочих методов из этого раздела пользуются либо итерированием, либо произвольным доступом в зависимости от того, реализует ли список интерфейс `RandomAccess`. Если да, то реализация выбирает алгоритм с произвольным доступом; но даже если список не реализует `RandomAccess`, но его размер ниже порогового, определенного для каждого метода в отдельности по результатам тестирования производительности, то используются алгоритмы с произвольным доступом.

Изменение содержимого списка

`<T> void copy(List<? super T> dest, List<? extends T> src)`	скопировать все элементы из одного списка в другой
`<T> void fill(List<? super T> list, T obj)`	заменить все элементы `list` значением `obj`
`<T> boolean replaceAll(List<T> list, T oldVal, T newVal)`	заменить все вхождения `oldVal` значением `newVal`

Эти методы изменяют некоторые или все элементы списка. Метод `copy` перемещает элементы из исходного списка в начальный подсписок конечного списка (который должен быть достаточно длинным, чтобы вместить их), а прочие элементы конечного списка оставляет неизменными. Метод `fill` заменяет все элементы списка указанным объектом, а `replaceAll` заменяет все вхожде-

ния одного значения другим (и новое, и старое значение могут быть равны null) и возвращает true, если была произведена хотя бы одна замена.

Сигнатуры этих методов можно объяснить с помощью принципа получения и вставки (см. раздел «Принцип получения и вставки» главы 2). Сигнатура copy обсуждалась в разделе «Джокеры с super» главы 2. Он получает элементы из исходного списка и помещает их в конечный, так что типы этих списков равны соответственно ? extends T и ? super T. Это согласуется с интуитивным представлением о том, что тип элементов исходного списка должен быть подтипом конечного списка. И хотя существуют более простые альтернативы сигнатуре copy, обсуждение в главе 2 доказывает, что использование джокеров всюду, где возможно, расширяет диапазон возможных применений.

Что касается fill, принцип получения и вставки настаивает, чтобы мы использовали super, когда вставляем значения в параметризованную коллекцию, а для replaceAll он говорит, что если мы помещаем и извлекаем значения из одной и той же структуры, то не должны использовать джокеров вовсе.

Эти методы не часто используются на практике. fill и copy ожидают, что конечный список с нужным числом элементов уже существует. Если вам нужно создать новый список для передачи в него содержимого, то имеется удобная альтернатива copy:

```
var dest = new ArrayList<>(src);
```

а для fill можно написать:

```
var dest = new ArrayList<>(Collections.nCopies(N, obj));
```

Метод replaceAll в значительной степени замещен методом по умолчанию List::replaceAll.

Нахождение экстремальных значений в коллекции

`<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)`	вернуть максимальный элемент в естественном порядке
`<T> T max(Collection<? extends T> coll, Comparator<? super T> comp)`	вернуть максимальный элемент в порядке, определяемом переданным компаратором
`<T extends Object & Comparable<? super T>> min(Collection<? extends T> coll)`	вернуть минимальный элемент в естественном порядке
`<T> T min(Collection<? extends T> coll, Comparator<? super T> comp)`	вернуть минимальный элемент в порядке, определяемом переданным компаратором

У каждого из методов min и max по два перегруженных варианта: один из них пользуется естественным порядком элементов, а другой принимает компаратор, определяющий порядок. Если переданная коллекция пуста, то они возбуждают исключение NoSuchElementException. Оба требуют линейного времени.

Эти методы и их типы обсуждаются в разделе «Множественные границы» и в разделе «Обеспечение двоичной совместимости» приложения (*https://mauricenaftalin.github.io/JGC_2e_Book_Code/appendix.html*).

Нахождение конкретных значений в списке

`<T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`	искать ключ `key` с помощью двоичного поиска
`<T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`	искать ключ `key` с помощью двоичного поиска
`int indexOfSubList(List<?> source, List<?> target)`	найти первый подсписок `source`, совпадающий с `target`
`int lastIndexOfSubList(List<?> source, List<?> target)`	найти последний подсписок `source`, совпадающий с `target`

Методы из этой группы ищут элементы или группы элементов в списке, опять-таки выбирая алгоритм в зависимости от размера списка и того, реализует ли он интерфейс `RandomAccess`.

Сигнатура первого перегруженного варианта `binarySearch` означает, что его можно использовать для поиска ключа типа `T` в списке объектов, которые могут иметь любой тип, допускающий сравнение с объектами типа `T`. Второй вариант похож на перегруженные варианты `min` и `max`, принимающие компаратор, но только в данном случае параметр-тип `Collection` должен быть подтипом типа ключа, который, в свою очередь, должен быть подтипом параметра-типа `Comparator`.

Для применимости двоичного поиска список должен быть отсортирован. В начале поиска диапазон индексов, в котором может находиться искомое значение, совпадает со всем списком. Алгоритм выбирает элемент в середине этого диапазона и использует его, чтобы определить новый диапазон: слева или справа от выбранного элемента. Может, конечно, случиться, что выбранный элемент равен искомому, и тогда поиск завершается. Поскольку на каждом шаге размер диапазона уменьшается вдвое, то для поиска в списке длины 2^m необходимо m шагов, и временная сложность поиска в списке длины N, реализующем интерфейс `RandomAccess`, составляет $O(\log N)$. Если же список не реализует `RandomAccess`, то метод `binarySearch` использует итерирование, и тогда его сложность линейна.

Методы `indexOfSubList` и `lastIndexOfSubList` не требуют, чтобы список был отсортирован. Их сигнатуры допускают элементы любого типа в исходном и конечном списках (напомним, что вместо двух джокеров могут быть подставлены разные типы). Это проектное решение продиктовано теми же соображениями, что для методов `containsAll`, `retainAll` и `removeAll` интерфейса `Collection` (см. раздел «Ограниченные или неограниченные?» главы 2).

Фабрики коллекций

`<T> List<T> emptyList()`	вернуть пустой список `List`
`<K,V> Map<K,V> emptyMap()`	вернуть пустое отображение `Map`
`<T> Set<T> emptySet()`	вернуть пустое множество `Set`
`<T> Iterator<T> emptyIterator()`	вернуть пустой `Iterator`
`<T> ListIterator<T> emptyListIterator()`	вернуть пустой `ListIterator`

`<T> NavigableSet<T> emptyNavigableSet()`	вернуть пустое `NavigableSet`
`<K,V> NavigableMap<K,V> emptyNavigableMap()`	вернуть пустое `NavigableMap`

Класс `Collections` предлагает удобные способы создания новых видов коллекций, содержащих нуль или более ссылок на один и тот же объект. Самые простые подобные коллекции – пустые.

В этом разделе, а также в следующем, посвященном фабрикам оберток, мы опустили методы, относящиеся к `SortedSet` и `SortedMap`, поскольку после появления `NavigableSet` и `NavigableMap` они считаются устаревшими.

Пустые коллекции могут быть полезны для реализации методов, возвращающих коллекции значений, в этом случае они означают, что возвращать нечего. Каждый метод возвращает ссылку на экземпляр синглтонного внутреннего класса `Collections`. Поскольку эти экземпляры неизменяемы, их можно безопасно разделять, поэтому вызов любого из этих фабричных методов не приводит к созданию объекта. До появления в Java дженериков для той же цели использовались поля `EMPTY_SET`, `EMPTY_LIST` и `EMPTY_MAP` класса `Collections`, но теперь они менее полезны, потому что их простые типы генерируют предупреждения о невозможности проверки при любом использовании. Сами методы `emptyList`, `emptyMap` и `emptySet` стали менее полезны, потому что их функции дублируются перегруженными вариантами фабричных методов `Set::of`, `List::of` и `Map::of` без параметров, порождающими немодифицируемые коллекции. Методы `emptyNavigableSet` и `emptyNavigableMap` в будущем тоже могут стать избыточными, если будут введены фабричные методы для создания немодифицируемых `NavigableSet` и `NavigableMap`.

Класс `Collections` также предлагает способы создания коллекций, содержащих только один элемент.

`<T> Set<T> singleton(T o)`	вернуть неизменяемое множество, содержащее только указанный объект
`<T> List<T> singletonList(T o)`	вернуть неизменяемый список, содержащий только указанный объект
`<K,V> Map<K,V> singletonMap(K key, V value)`	вернуть неизменяемое отображение ключа `K` на значение `V`

И эти методы могут быть полезны, когда нужно передать единственное входное значение методу, ожидающему коллекцию значений. И снова они дублируются фабричными методами, порождающими немодифицируемые коллекции.

Наконец, можно создать список, содержащий несколько копий данного объекта:

`<T> List<T> nCopies(int n, T o)`	вернуть неизменяемый список, содержащий `n` ссылок на объект `o`

Поскольку список, созданный методом `nCopies`, неизменяем, достаточно всего одного физического элемента для создания представления списка

нужной длины. Такие списки часто используются в качестве основы для построения дальнейших коллекций – например, в качестве аргумента конструктора или метода `addAll`.

Обертки

Класс `Collections` предоставляет объекты-обертки, которые модифицируют поведение стандартных классов коллекций одним из трех способов: синхронизируя их, делая их немодифицируемыми или проверяя типы добавляемых в них элементов. Эти обертки реализуют те же интерфейсы, что и обернутые объекты, и делегируют им всю содержательную работу. Их цель – ограничить условия, при которых будет выполнена работа. Это примеры использования *защитных прокси* (Gamma et al. 1995), варианта паттерна Заместитель, в котором заместитель (прокси) управляет доступом к реальному субъекту.

Прокси можно создавать разными способами. Здесь они создаются фабричными методами, которые обертывают переданный объект коллекции внутренним классом `Collections`, реализующим интерфейс той же коллекции. Впоследствии вызовы методов прокси (по большей части) делегируются объекту коллекции, но прокси контролирует условия вызова. В случае синхронизированных оберток делегируются все методы, но прокси организует синхронизацию, гарантирующую, что в каждый момент времени к объекту обращается только один поток. В случае немодифицируемых и проверяемых коллекций вызовы методов, нарушающие контракт прокси, возбуждают подходящее исключение.

Синхронизированные коллекции

Как было объяснено в разделе «Коллекции и потокобезопасность» главы 9, большинство классов коллекций в каркасе не являются потокобезопасными. Так и задумано, чтобы избежать накладных расходов на ненужную синхронизацию (чем страдали унаследованные классы `Vector` и `Hashtable`). Но для тех случаев, когда все же необходимо предоставить небольшому числу потоков конкурентный доступ к одной и той же коллекции, класс `Collections` предоставляет следующие синхронизированные обертки:

```
<T> Collection<T> synchronizedCollection(Collection<T> c)
<T> Set<T> synchronizedSet(Set<T> s)
<T> List<T> synchronizedList(List<T> list)
<K,V> Map<K,V> synchronizedMap(Map<K,V> m)
<T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> s)
<K,V> NavigableMap<K,V> synchronizedNavigableMap(NavigableMap<K,V> m);
```

Потокобезопасность, предлагаемая этими обертками, опирается на отказ от несинхронизированного доступа к базовой коллекции. Типичный способ применения – обернуть коллекцию в момент конструирования, не сохраняя никаких ссылок на обернутую коллекцию:

```
var synchList = Collections.synchronizedList(new ArrayList<>( ... ));
```

Классы, которые предоставляют эти синхронизированные представления, являются условно потокобезопасными (см. раздел «Коллекции и потокобезопасность» главы 9); хотя гарантируется, что все их операции атомарны, вам, возмож-

но, придется синхронизировать несколько вызовов методов самой обертки, чтобы получить согласованное поведение. В частности, итераторы необходимо создавать и использовать целиком внутри синхронизированного блока кода, иначе в лучшем случае будет возбуждено исключение `ConcurrentModificationException`. Это очень грубая синхронизация; если в вашем приложении активно используются синхронизированные коллекции, то его степень конкурентности заметно снижается. В этой ситуации лучше обратиться к подходящей конкурентной коллекции (см. раздел «Избегайте синхронизированных оберток коллекций» главы 17).

Немодифицируемые коллекции

Немодифицируемая коллекция возбуждает исключение `UnsupportedOperationException` в ответ на любую попытку изменить ее структуру или состав элементов. При должной осторожности это может быть полезно, когда вы хотите разрешить клиентам чтение внутренней структуры данных; передача структуры в немодифицируемой обертке не позволит клиенту изменить ее, но не помешает изменить содержащиеся в ней объекты, если они модифицируемые. Часто приходится защищать внутреннюю структуру данных, отдавая клиентам специально созданную копию или заключая объекты в немодифицируемые обертки. Мы рассмотрим эти варианты в разделе «Не забывайте о “владельцах” коллекций» главы 17.

Имеются фабричные методы для создания немодифицируемых коллекций, соответствующих всем основным интерфейсам в каркасе коллекций:

```
<T> Collection<T> unmodifiableCollection(Collection<? extends T> c)
<T> Set<T> unmodifiableSet(Set<? extends T> s)
<T> List<T> unmodifiableList(List<? extends T> list)
<K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m)
<T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<? extends T> s)
<K,V> NavigableMap<K,V> unmodifiableNavigableMap(NavigableMap<K,? extends V>
m)
<T> SequencedCollection<T>
unmodifiableSequencedCollection(SequencedCollection<? extends T> c)
<K,V> SequencedMap<K,V>
unmodifiableSequencedMap(SequencedMap<? extends K,? extends V> m)
<T> SequencedSet<T> unmodifiableSequencedSet(SequencedSet<? extends T> s)
```

Проверяемые коллекции

Предупреждение компилятора о невозможности проверки – сигнал о том, что нужно предпринять особые меры во избежание нарушений типов во время выполнения. Например, после передачи ссылки на типизированную коллекцию непараметризованному методу библиотеки мы не можем быть уверены, что он добавил в коллекцию только элементы правильного типа. Вместо того чтобы смириться с неуверенностью в типобезопасности коллекции, мы можем передать проверяющую обертку, которая следит за тем, чтобы все добавляемые в коллекцию элементы имели тот же тип, который был указан при ее создании. В разделе «Обеспечивайте типобезопасность при вызове недоверенного кода» главы 7 приведен пример такой техники.

Каждый из следующих методов создает объект, содержащий экземпляр некоторого интерфейса вместе с маркером типа элемента. Все операции делеги-

руются экземпляру, но перед добавлением нового элемента в коллекцию он сначала сравнивается с маркером типа, и в случае несовпадения возбуждается исключение `ClassCastException`:

```
<E> Collection<E> checkedCollection(Collection<E> c, Class<E> elementType)
<E> List<E> checkedList(List<E> c, Class<E> elementType)
<E> Set<E> checkedSet(Set<E> c, Class<E> elementType)
<E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> c, Class<E> element-
Type)
<K,V> Map<K,V> checkedMap(Map<K,V> c, Class<K> keyType, Class<V> valueType)
<K,V> NavigableMap<K,V>
    checkedNavigableMap(NavigableMap<K,V> c, Class<K> keyType, Class<V> valu-
eType)
<E> Queue<E> checkedQueue(Queue<E> c, Class<E> elementType)
```

Прочие методы

Класс `Collections` предоставляет ряд служебных методов, с некоторыми из них мы уже встречались. Далее они перечисляются в алфавитном порядке.

addAll

`boolean addAll(Collection<? super T> c, T... elements)`	добавляет все указанные элементы в указанную коллекцию

Мы несколько раз использовали этот метод как удобный и эффективный способ инициализировать коллекцию отдельными элементами или содержимым массива.

asLifoQueue

`<T> Queue<T> asLifoQueue(Deque<T> deque)`	возвращает представление `Deque` в виде LIFO-очереди (последним пришел, первым ушел)

Напомним (см. главу 13), что, хотя очереди могут упорядочивать свои элементы разными способами, не существует стандартной реализации `Queue`, предлагающей порядок LIFO. С другой стороны, все реализации `Deque` поддерживают порядок LIFO, если элементы удаляется с той же стороны очереди, с которой добавлялись. Метод `asLifoQueue` позволяет воспользоваться этой функциональностью через удобный интерфейс `Queue`.

disjoint

`boolean disjoint(Collection<?> c1, Collection<?> c2)`	возвращает `true`, если `c1` и `c2` не имеют общих элементов

Реализация метода в OpenJDK обходит одну из этих коллекций, проверяя каждый элемент на принадлежность другой. Метод возвращает `true`, если ни одного совпадения не найдено. Это не вызывает трудностей для коллекций, отличных от множества, в которых обычно для проверки принадлежности используется метод `equals`. Но в некоторых множествах для определения членства используются другие отношения эквивалентности (см. раздел «Опреде-

ление множества: отношения эквивалентности» главы 12); если на отсутствие пересечения проверяются два множества с разными отношениями эквивалентности, то результат может зависеть от того, какое из двух множеств выбрано для проверки принадлежности. Как и в случае равенства множеств, следует избегать использования `disjoint` для множеств с разными отношениями эквивалентности (см. подробное обсуждение этого вопроса в разделе «Несогласованность с equals» главы 18).

enumeration

`<T> Enumeration<T> enumeration(Collection<T> c)`	возвращает перечисление указанной коллекции

Этот метод предназначен для взаимодействия с API, методы которых принимают аргументы типа `Enumeration`, унаследованной версии `Iterator`. Перечисление `Enumeration`, которое он возвращает, отдает те же элементы и в том же порядке, что `Iterator`, предоставленный коллекцией `c`. Это метод, парный методу `list`, который конструирует `ArrayList` из элементов, отдаваемых в порядке, определяемом `Enumeration`.

frequency

`int frequency(Collection<?> c, Object o)`	возвращает число элементов в `c`, равных `o`

Если переданное значение `o` равно `null`, то `frequency` возвращает число элементов `null` в коллекции `c`.

list

`<T> ArrayList<T> list(Enumeration<T> e)`	возвращает `ArrayList`, содержащий элементы, возвращаемые указанным перечислением `e`

Этот метод предназначен для взаимодействия с API, методы которых возвращают результаты типа `Enumeration`, унаследованной версии `Iterator`. Список `ArrayList`, который он возвращает, содержит те же элементы и в том же порядке, что перечисление `e`. Это метод, парный методу `enumeration` (см. выше).

newSequencedSetFromMap и newSetFromMap

`<E> SequencedSet<E> newSequencedSetFromMap (SequencedMap<E,Boolean> map)`	возвращает `SequencedSet`, основанное на указанном `SequencedMap`
`<E> Set<E> newSetFromMap(Map<E,Boolean> map)`	возвращает `Set`, основанное на указанном `Map`

В главе 15 мы видели, что многие множества (например, `TreeSet` и `ConcurrentSkipListSet`) реализованы с помощью отображений и разделяют их порядок и свойства конкурентности и производительности. Однако у некоторых отображений (например, `WeakHashMap` и `IdentityHashMap`) нет стандартных эквивалентных множеств. Цель метода `newSetFromMap` – предоставить реализацию эквивалентного множества для таких отображений. Метод `newSetFromMap` обертывает свой аргумент, который должен быть пустым в момент передачи

и к которому в дальнейшем не должно быть прямого доступа. Этот код демонстрирует стандартную идиому для создания слабого `HashSet`, элементы которого удерживаются слабыми ссылками:

```
Set<Object> weakHashSet = Collections.newSetFromMap(
    new WeakHashMap<Object, Boolean>());
```

На первый взгляд кажется, что полезность `newSequencedSetFromMap` невелика, потому что, в отличие от реализаций `Map`, все реализации `SequencedMap` в JDK имеют стандартные эквиваленты для множеств. Однако реализации `LinkedHashSet`, которая соответствует `LinkedHashMap`, недостает одной важной вещи, имеющейся у отображения: способности определять политику вытеснения (или обратный вызов), которая вызывалась бы при каждом добавлении нового элемента. Эту возможность предоставляет `SequencedSet`, созданное из `LinkedHashMap`. Например, после того как количество элементов в множестве ниже станет равным пяти, добавление каждого следующего приводит к вытеснению самого старого элемента:

```
SequencedSet<String> set = Collections.newSequencedSetFromMap(
    new LinkedHashMap<String,Boolean>() {
      protected boolean removeEldestEntry(Map.Entry<String,Boolean> e) {
          return this.size() > 5;
      }
});
```

reverseOrder

`<T> Comparator<T> reverseOrder()`	возвращает компаратор, обращающий естественный порядок

Метод `reverseOrder` дает простой способ отсортировать или поддерживать коллекцию объектов типа `Comparable` в обратном естественном порядке. Приведем пример:

```
SortedSet<Integer> s = new TreeSet<>(Collections.reverseOrder());
Collections.addAll(s, 1, 2, 3);
assert s.equals(new TreeSet<>(Set.of(1, 2, 3)).reversed());
```

Этот метод появился раньше версии Java 8, в которой были введены статические методы интерфейсов. В Java 8 в интерфейс `Comparator` был включен метод `reverseOrder`, обладающий в точности такой же функциональностью, как `Collections::reverseOrder`. Разработчикам, которым нужно работать с компараторами, проще найти его именно в этом месте. Кроме того, он исправляет проблему в методе из класса `Collections`, параметр-тип которого определен неточно, так что возможно создание компараторов, порождающих ошибки во время выполнения. Например, компаратор, созданный как

```
Comparator<Object> objReverseCompr = Collections.reverseOrder();
```

будет работать правильно применительно к коллекции допускающих сравнение элементов:

```
var strings = new ArrayList<>(List.of("a", "b", "c"));
strings.sort(objReverseCompr);
assert strings.equals(List.of("c", "b", "a"));
```

но выдаст ошибку, если применить его к коллекции объектов, не допускающих сравнения:

```
var classes = new ArrayList<>(List.of(String.class,
    Integer.class));        // возбуждает ClassCastException
classes.sort(objReverseCompr);
```

Такая проблема не может возникнуть для метода интерфейса `Comparator`, который требует, чтобы параметром был тип `Comparable`:

```
<T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

В классе `Collections` имеется второй перегруженный вариант этого метода:

```
<T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

Этот метод похож на предыдущий, но обращает не естественный порядок коллекции объектов, а порядок компаратора, переданного в качестве аргумента. Его поведение в случае, когда переданный аргумент равен `null`, необычно для метода класса `Collections`. Контракт `Collections` оговаривает, что его методы возбуждают исключение `NullPointerException`, если переданные коллекции или объекты классов равны `null`, но если передать `null` этому методу, то он даст такой же эффект, как вызов `reverseOrder`, т. е. вернет `Comparator`, обращающий естественный порядок коллекции объектов.

У этого метода также имеется эквивалент в интерфейсе `Comparator`: метод `Comparator::reversed` по умолчанию (т. е. метод экземпляра), в котором место параметра статического метода `Collections` занимает объект, от имени которого вызван метод.

Заключение

Если вы дочитали до этого места, то имеете довольно полную картину возможностей, предлагаемых каркасом коллекций Java. Но знать, что доступно, и уметь это применять – разные вещи. Долгая жизнь каркаса коллекций – уже более четверти века на момент написания книги – позволила собрать опыт миллионов Java-программистов, пользовавшихся им. Следующая глава содержит наставление, извлеченное из этого коллективного опыта.

Глава 17

Наставление по использованию каркаса коллекций Java

Каркас коллекций Java активно используется миллионами работающих Java-программистов с момента своего появления в 1998 году, т. е. уже четверть века на момент написания книги. В этой главе изложено несколько уроков, которые мы можем извлечь из этого коллективного опыта. Даже вещи, кажущиеся элементарными, стоит сформулировать явно; каждое наставление основано на реальном коде, где пренебрежение им привело к системам, уязвимым к ошибкам или трудным для сопровождения.

В этой главе термины *библиотека*, *API* и *клиент* используются в следующем смысле: *библиотека* – любой класс, раскрывающий открытый метод, ее *API* – множество раскрываемых ей открытых методов, а *клиент* – любой класс, вызывающий методы API.

Примеры кода к этой главе можно скачать по адресу *https://github.com/MauriceNaftalin/JGC_2e_Book_Code/tree/main/src/main/java/org/ jgcbook/chapter17*.

Избегайте анемичных моделей предметной области

Термин *анемичная модель предметной области* предложил Мартин Фаулер (2003) для описания дизайна, в котором объекты модели предметной области по существу являются лишь носителями данных и не содержат почти никакой или совсем никакой бизнес-логики. В стандартной модели объектно ориентированного проектирования это ведет к концентрации бизнес-логики на уровне служб. Фаулер отмечает, что «проблема анемичных моделей предметной области заключается в том, что они несут все затраты такой модели [в частности, необходимость сохраняемых отображений], не получая от нее никакой выгоды». Описанный им сценарий допускает именование этих объектов существительными предметной области, тогда как в каркасе коллекций этот антипаттерн проявляется в виде безымянных коллекций, содержащих другие

коллекции, иногда глубоко вложенные, и полном отсутствии определенного пользователем поведения. В обоих случаях ошибка проектирования, по сути дела, одна и та же.

Простой пример поможет прояснить сказанное. Допустим, вы решили включить новую сущность в систему управления задачами, описанную в предыдущих главах: проекты, т. е. группы задач, собранные под единым именем. Уровень служб, предназначенный для управления проектами в интересах пользователя, мог бы смоделировать их просто как отображение имени проекта на коллекцию задач. С одним методом – для добавления задачи – уровень служб мог бы выглядеть примерно так:

```
public class ServiceLayer {

  Map<String,Set<Task>> projects = new HashMap<>();

  public void addTask(String projectName, Task task) {
    projects.merge(projectName, new HashSet<>(Set.of(task)),
        (oldValue,newValue) -> {oldValue.addAll(newValue); return oldValue;});
  }
  ...
}
```

Здесь главная проблема анемичной модели предметной области очевидна: логика, относящаяся к объекту предметной области, никак не выражена и на самом деле при таком представлении и не может быть выражена. Поэтому ее приходится переносить на уровень служб. Эта проблема растет как снежный ком по мере усложнения логики предметной области. Например, предположим, что мы включили в задачу ее продолжительность:

org/jgcbook/chapter17/A_avoid_anemic_domain_models/Task

```
public interface Task extends Comparable<Task> {
  @Override
  default int compareTo(Task t) {
    return toString().compareTo(t.toString());
  }
  Duration duration();
}
```

org/jgcbook/chapter17/A_avoid_anemic_domain_models/CodingTask

```
public record CodingTask(String spec, Duration duration) implements Task {}
```

org/jgcbook/chapter17/A_avoid_anemic_domain_models/PhoneTask

```
public record PhoneTask(String name, String number, Duration duration)
    implements Task {}
```

и что у проектов часто запрашивают суммарную продолжительность входящих в них задач, так что эту величину имеет смысл кешировать. Теперь уровень служб должен содержать отдельную структуру данных для хранения кешированных суммарных продолжительностей каждого проекта:

```
public class ServiceLayer {

  Map<String,Set<Task>> tasksPerProject = new HashMap<>();
  Map<String,Duration> durationsPerProject = new HashMap<>();
```

```
  public void addTask(String projectName, Task task) {
    tasksPerProject.merge(projectName, new HashSet<>(Set.of(task)),
       (oldValue,newValue) -> {oldValue.addAll(newValue); return oldValue;});
    durationsPerProject.merge(projectName, task.duration(), Duration::plus);
  }
  ...
}
```

Неявный инвариант приложения включает требование, что каждое значение в отображении `durations` должно быть суммой продолжительностей соответствующих задач в отображение `projects`. Этот инвариант очень непросто выразить, а поддерживать его без инкапсуляции будет и того труднее. Инвариант относится к компонентам одного проекта, поэтому инкапсуляция логики проекта в одном объекте позволила бы выразить это требование в более понятном и удобном для сопровождения виде:

```
public class Project {

  private final String name;
  private final Set<Task> tasks;
  private Duration totalDuration;

  public void addTask(Task task) {
    tasks.add(task);
    totalDuration = totalDuration.plus(task.duration());
  }
  ...
}
```

Хотя размещение обязанностей объектов типа `Project` в классе `Project` согласуется со стандартной практикой объектно ориентированного проектирования, полезно выработать навык распознавания вложенных типов коллекций как «запашок в коде», понуждающий подвергнуть дизайн рефакторингу.

Не забывайте о «владельцах» коллекций

Самые существенные решения при проектировании каркаса коллекций, после того как все согласились, что коллекции должны быть изменяемыми, касались управления изменениями. Из этих решений важнейшим было то, что операции чтения и записи должны быть объявлены в одном интерфейсе, а не разнесены по разным. В этом разделе мы рассмотрим последствия этого решения для разработчиков, желающих инкапсулировать данные коллекций в своих объектах.

Среди этих последствий наличие факультативных операций, которые в некоторых случаях не поддерживаются немодифицируемыми или частично модифицируемыми коллекциями и представлениями коллекций. Наличие факультативных операций вызвало критику решения объединить операции чтения и записи в одном интерфейсе, однако при изучении этого вопроса в разделе «Фундаментальные проблемы дизайна каркаса коллекций» главы 18 мы увидим, что ни один подход к проблеме гарантированной инкапсуляции не обходится без компромиссов.

Итак, при условии, что коллекции изменяемы, а интерфейсы объявляют методы изменения, как объект может безопасно инкапсулировать свои данные?

Ответы на этот вопрос мы исследуем на примере класса `Project`, описанного в предыдущем разделе. При проектировании API этого класса мы хотели бы рассмотреть операции, в которых может возникнуть надобность у клиента. Очевидно, что должна быть возможность добавить задачу в проект или удалить ее из проекта:

```
public class Project {
  ...
  public void addTask(Task task) {
    tasks.add(task);
    totalDuration = totalDuration.plus(task.duration());
  }
  public void removeTask(Task task) {
    tasks.remove(task);
    totalDuration = totalDuration.minus(task.duration());
  }
}
```

Но предположим, мы предчувствуем, что клиенту может потребоваться обойти содержимое коллекции `tasks` или вывести его в поток. Предвидеть все возможные способы использования данных трудно, поэтому может возникнуть искушение заменить все методы одним, который дает доступ к самому полю:

```
public class Project {
  ...
  public Set<Task> getTasks() {
    return tasks;
  }
  ...
}
```

Но у такого (весьма распространенного) подхода есть большой недостаток: никаким способом нельзя гарантировать, что клиент будет поддерживать инвариант класса. Возможно, он будет выполнять только операции чтения, и в этом случае никакой проблемы не будет. Но запись в коллекцию может приводить к тонким ошибкам, которые трудно диагностировать, поскольку их симптомы могут не иметь видимой связи с истинной причиной. В нашем примере, если клиент добавит или удалит задачу, не изменив `totalDuration`, то это поле перестанет правильно отражать полную длительность задач проекта. Вы, проектировщик API, доверяете клиенту? Быть может, и да, если это часть одной тесно связанной системы, которую будет поддерживать тот же человек или команда, что поддерживает ваш библиотечный код. Но если клиент ничего не знает о внутренней логике вашего кода или вы обеспокоены возможностью непреднамеренного изменения, то придется приложить дополнительные усилия для обеспечения инкапсуляции. Особенно инкапсуляция важна в большой системе: она позволяет рассуждать о коде, зная только о частях системы, а не о системе в целом. Лучший способ защитить состояние объекта `Project` – раскрывать его только в немодифицируемой обертке. Тогда возникает необходимость восстановить индивидуальные методы изменения, как в примере 17.1. (Мы добавили в этом примере реализации методов `equals` и `hashCode`, которые понадобятся позже.)

Пример 17.1. *Акцессор, возвращающий немодифицируемую коллекцию*

org/jgcbook/chapter17/B_ownership/Project

```
public class Project {

  private final String name;
  private final Set<Task> tasks;
  private Duration totalDuration;

  public Project(String name, Set<Task> tasks) {
    this.name = name;
    this.tasks = tasks;
    totalDuration = tasks.stream()
        .map(Task::duration)
        .reduce(Duration.ZERO, Duration::plus);
  }

  public String getName() { return name; }

  public Duration getTotalDuration() { return totalDuration; }

  public Set<Task> getTasks() {
    return Collections.unmodifiableSet(tasks);
  }
  public void addTask(Task task) {
    tasks.add(task);
    totalDuration = totalDuration.plus(task.duration());
  }
  public void removeTask(Task task) {
    tasks.remove(task);
    totalDuration = totalDuration.minus(task.duration());
  }
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Project project)) return false;
    if (!name.equals(project.name)) return false;
    return tasks.equals(project.tasks);
  }
  @Override
  public int hashCode() {
    int result = name.hashCode();
    result = 31 * result + tasks.hashCode();
    return result;
  }
}
```

Такое обертывание раскрываемого состояния гарантирует, что объект `Project` не будет поврежден клиентом, получившим ссылку на него с помощью методов-акцессоров.

Принцип, гласящий, что клиенты не должны иметь возможности изменить внутреннее состояние объекта, весьма общий, и ослабить его можно только в случае тесно связанных кооперативных компонентов системы. Когда, как в данном случае, раскрываемое состояние – одна из коллекций, входящих в каркас, немодифицируемые обертки класса `Collections` – именно то, что нуж-

но для реализации этого принципа. Для других видов изменяемых данных эта техника может быть непригодна; в таких случаях можно вместо нее использовать *защитное копирование* для защиты раскрываемого состояния объекта. При этом акцессоры создают копию состояния объекта и возвращают ссылку на нее. Эта копия отличается от самого состояния, поэтому клиент может модифицировать ее, не боясь повредить объект.

Необходимость в этом возникает, например, когда компонентом состояния объекта является массив. Массивы изменяемы, но неудобства, связанные с необходимостью копировать их при каждом вызове акцессора, в какой-то мере компенсируются тем фактом, что у массивов имеется метод `clone`. Если коллекцию задач проекта требуется представить массивом – по причине совместимости или производительности, – то код должен выглядеть примерно так:

```
public class ProjectWithArray {

  private final String name;
  private final Task[] tasks;
  private Duration totalDuration;

  public ProjectWithArray(String name, Task[] tasks) {
    this.name = name;
    this.tasks = tasks;
    totalDuration = Arrays.stream(tasks)
        .map(Task::duration)
        .reduce(Duration.ZERO, Duration::plus);
  }

  public Task[] getTasks() {
    return tasks.clone();
  }
  ...
}
```

В общем случае предоставление открытого метода `clone` означает, что класс реализует интерфейс `Cloneable`, и не всегда это тривиально (см. Bloch 2017, совет 13). Можно вместо этого предоставить копирующий конструктор или – это более современная идиома – статический фабричный метод копирования.

Защитное копирование также используется – и на самом деле даже чаще, – чтобы защитить состояние объекта от изменения входных данных. Например, инкапсуляция состояния `Project` еще не полна, как показывает следующий код:

```
var tasks = new HashSet<Task>();
tasks.add(new CodingTask("code ui", Duration.ofHours(4)));
tasks.add(new CodingTask("code db", Duration.ofHours(6)));
var project1 = new Project("Project1", tasks);
var project2 = new Project("Project2", tasks);
```

Наличие двух разных проектов с одной и той же коллекцией задач создало между ними тесную связь, поэтому добавление или удаление задачи из `project1` неожиданно влияет на `project2`, и наоборот. Это еще один способ, когда объекты могут утратить контроль над собственным состоянием, в данном случае потому что его нельзя однозначно назвать их собственностью: состояние разделяется с другим объектом. В компьютерной литературе эта пробле-

ма называется *совмещением* (aliasing). Ее можно избежать, сделав защитную копию и тем самым разорвав связь между состоянием проекта и коллекцией, переданной его конструктору. Это также позволяет конструктору выбрать собственное представление данных. Например, возможно, мы хотим хранить задачи внутри проекта в отсортированном виде, не обременяя этим клиента.

```
private final NavigableSet<Task> tasks;
...
public Project(String name, Set<Task> tasks) {
  this.name = name;
  this.tasks = new TreeSet<>(tasks);
  totalDuration = tasks.stream()
      .map(Task::Duration)
      .reduce(Duration.ZERO, Duration::plus);
}
```

Короче говоря, мы рассмотрели две основные техники инкапсуляции данных объекта.

- Чтобы защитить состояние объекта от изменения входных данных, нет никакой альтернативы защитному копированию, гарантирующему монопольный доступ к ссылке на объект.
- Чтобы защитить состояние, которое иначе было бы раскрыто внешнему миру, есть две возможности:
  - если возможно (как в случае коллекций), обернуть его немодифицируемой оберткой. Это лучший вариант, поскольку защитное копирование – потенциально дорогая операция, которую к тому же надо повторять при каждом вызове акцессора;
  - если получить немодицируемую обертку невозможно, то придется применить защитное копирование.

Отдавайте предпочтение неизменяемым объектам в качестве элементов множества или ключей отображения

Многие ошибки в коде с использованием каркаса коллекций имеют одну и ту же причину: неумение предотвратить изменение элементов хешированной или упорядоченной коллекции. Структуры данных, которые хранят свои элементы по значению, ожидают, что для их извлечения будет использовано то же самое значение. Хотя это правило может показаться очевидным, его легко нарушить, а результаты оказываются удивительными. Например, предположим, что мы хотели сохранить коллекцию объектов `Project`, определенных в примере 17.1, в множестве `HashSet`. Следующий код, выглядящий вполне разумно, может преподносить странные сюрпризы:

org/jgcbook/chapter17/C_use_immutable_objects_as_set_elements_map_keys/Program_1

```
var tasks = new HashSet<Task>(Set.of(new CodingTask("code ui",
                                                    Duration.ofHours(4))));
var myProject = new Project("My Project", tasks);
var projectSet = new HashSet<>(Set.of(myProject));
```

```
assert projectSet.contains(myProject);
myProject.addTask(new CodingTask("code db", Duration.ofHours(6)));
assert ! projectSet.contains(myProject);
```

Последнее утверждение может показаться еще более удивительным в свете того, что первая строчка метода `Project::equals` содержит проверку тождественности (добавлена ради эффективности). Множество `myProjects` не находит тот самый объект, который был в нем всего двумя строчками выше! Природу этой ошибки легко понять, глядя на рис. 12.1. Выбор ячейки, определяющей отправную точку поиска элемента, делается на основе хеш-кода объекта: хранимый объект ни за что не удастся найти, если искать его не в той цепочке переполнения. В данном случае `Project::hashCode` включает в вычисление хеш-кода компонент `tasks`, поэтому, когда этот компонент изменяется, изменяется и хеш-код, и значит, в качестве отправной точки поиска будет выбрана другая ячейка.

Естественный ход мысли подсказывает, что нужно исключить изменяемый компонент `tasks` из вычисления хеш-кода. В некоторых случаях это может решить проблему, но только ценой снижения качества алгоритма хеширования; здесь у нас есть какая-никакая уверенность, что разные проекты будут иметь разные имена, но в реальной ситуации исключение важных частей объекта может привести к неприемлемой концентрации записей в небольшом числе ячеек.

Нечто подобное может случиться и с `TreeSet`.

org/jgcbook/chapter17/C_use_immutable_objects_as_set_elements_map_keys/Program_2

```
Task codeUi = new CodingTask("code ui", Duration.ofHours(6));
var project1 = new Project("project1", new HashSet<>(Set.of(codeUi)));

Task codeDb = new CodingTask("code db", Duration.ofHours(4));
var project2 = new Project("project2", new HashSet<>(Set.of(codeDb)));

var projectSet = new TreeSet<>(Comparator.comparing(Project::getTotalDurati
on)
    .thenComparing(Project::getName));

projectSet.addAll(List.of(project1,project2)); ❶
assert projectSet.contains(project2);
project2.addTask(new CodingTask("code ai", Duration.ofHours(5))); ❷
assert ! projectSet.contains(project2); ❸
```

На рис. 17.1 показано состояние `TreeSet` до и после вызова `addTask` в точках ❶ и ❷.

После вызова `addAll` в точке ❶ проект `project2` помещается в левое поддерево `TreeSet`, потому что суммарная длительность задач в нем меньше, чем в корневом элементе, `project1`. Но после добавления новой задачи в `project2` в строке ❷ его состояние изменилось, и теперь он должен быть в правом дереве. Там его и ищет метод `contains`, вызванный в строке ❸, и, конечно, не находит. Алгоритм поиска в `TreeSet` зависит от инварианта, который утверждает, в частности, что элементы, считающиеся больше некоторой вершины, всегда должны находиться в ее правом поддереве. Этот инвариант был нарушен вызовом `addTask` в строке ❷, поэтому правильная работа алгоритма поиска больше не гарантируется.

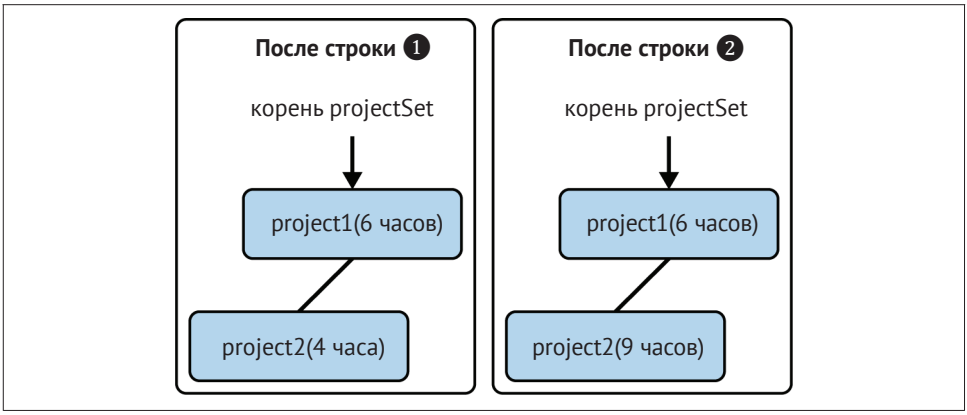


Рис. 17.1. `TreeSet` до и после нарушения инварианта

Любая коллекция, которая организует свои элементы согласно их значениям, будет подвержена тем же эффектам. Это относится также к коллекциям `HashMap` и `TreeMap`, которые организованы с учетом значений ключей. Из популярных классов коллекций только `List` не подвержен этой проблеме, потому что организация его элементов зависит от действий клиента. Другие классы коллекций попросту непригодны для хранения элементов, которые будут изменяться, и, чтобы не столкнуться с такими проблемами, желательно по возможности хранить в них неизменяемые элементы.

Соблюдайте баланс интересов клиента и библиотеки при проектировании API

В некотором смысле этот раздел посвящен проектированию API вообще, а не коллекциям. Общий лозунг объектно ориентированного проектирования – «программируйте, ориентируясь на интерфейс»; вопрос в том, на какой интерфейс? Этот вопрос возникает всякий раз, как мы проектируем метод, который принимает параметр или возвращает значение, типы которых находятся где-то в иерархии типов. С API, которые передают коллекции, это случается чаще, поскольку иерархия типов коллекций необычно глубока. Например, метод `Project::getTasks` из предыдущего раздела мог бы возвращать значение типа `Iterable`, `Collection`, `SequencedCollection`, `Set`, `SequencedSet`, `NavigableSet` и даже конкретного типа `TreeSet`.

Решение о том, в какое место иерархии типов поместить тип параметра или возвращаемого значения, – это упражнение на соблюдение баланса интересов приемника, – они склоняют нас к выбору наиболее специфичного типа – и интересов источника – а они побуждают сохранить максимум гибкости за счет передачи наиболее абстрактного типа. Например, разработчик класса `Project` мог бы отдать предпочтение возврату `Collection<Task>`, поскольку этот тип обеспечивает максимальную гибкость реализации. У персонала сопровождения осталась бы свобода изменить реализацию на коллекцию любого вида – скажем, список, – если это может дать какие-то преимущества. Про-

блема же клиента в том, что такой выбор лишает его потенциально полезной информации о возвращенной коллекции, а именно, что она упорядочена и не содержит дубликатов.

Бросаясь в противоположную крайность, мы могли бы вернуть из `getTasks` значение конкретного типа, `TreeSet`. Теперь проблема в том, что класс `Project` серьезно ограничен. Например, на более поздней стадии разработки могло бы оказаться, что коллекция задач должна быть потокобезопасной, а это потребовало бы перехода на реализацию `ConcurrentSkipListSet`. Однако внести такое изменение в `Project` было бы невозможно, если ранее было объявлено, что коллекция имеет тип `TreeSet`, и уже написаны клиенты, опирающиеся на этот факт.

Итак, ключевой вопрос – как найти баланс между требованиями к гибкости реализаций библиотеки и семантикой, которая фактически нужна ее потребителю. В данном случае станет ли клиенту легче, если он будет знать, что возвращенная коллекция отсортирована? Если да, то `getTasks` должна вернуть `NavigableSet`. Или же ему необходимо только иметь доступ к первому и последнему элементу? Тогда мы могли бы вернуть `SequencedSet` и оставить возможность в будущем изменить реализацию на `LinkedHashSet`. (Если вы забыли, как связаны эти типы, обратитесь к рис. 12.2.) Если клиенту не понадобится информация о том, что задачи уникальны, то мы могли бы вернуть `SequencedCollection`.

Аналогичные сообщения применимы к выбору типов параметров. Чтобы API был полезен максимально широкому кругу клиентов, он должен принимать тип, расположенный высоко в иерархии. Это имеет смысл и с точки зрения балансирования трудоемкости разработки: обрабатывать супертип нужно только один раз в коде библиотеки, тогда как адаптировать код клиента к требованиям библиотеки нужно при вызове каждого метода. В случае конструктора `Project` этот ход рассуждений мог бы побудить нас принимать произвольную коллекцию задач (`Collection`).

Вообще говоря, наилучшего баланса можно достичь, выбирая тип в середине иерархии, но в каждом конкретном случае принимать решение следует с учетом семантики API.

Пользуйтесь возможностями записей

Было непросто определять кортежи в Java, прежде чем записи получили свой окончательный и незыблемый статус в версии Java 16. Вообще говоря, кортежи в основном используются как переносчики данных. Если вам нужен был такой переносчик до Java 16, то приходилось определять для этой цели класс, что требовало явных объявлений: конструктора для инициализации полей, а также методов проверки на равенство, вычисления хеш-кода, доступа к полям и представления в виде строки. Кроме того, желательно, чтобы переносчики данных были неизменяемы (см. раздел «Неизменяемость и немодифицируемость» главы 9). Представление простого кортежа, удовлетворяющее этим требованиям, оказывалось очень тяжеловесным, поэтому Java-разработчики прибегали к различным уловкам, чтобы вообще избежать использования кортежей. С появлением записей, которым удовлетворяют этим требованиям автоматически, а заодно могут определять поведение, если нужно, необходимость в этих уловках отпала.

Отдавайте предпочтение записям перед параллельными списками

Одно из полезных применений записей – замена параллельных списков. Здесь термин *параллельный* означает не наличие нескольких потоков, в структуры данных, которые неявно объединяют несколько списков в параллельную структуру, где элементы с одинаковыми индексами представляют компоненты одного объекта[1]. Например, рассмотрим задачу моделирования турнирной лестницы, по которой игрок продвигается, победив игрока, стоящего прямо над ним. Одна из возможных реализаций – два параллельных списка: список игроков и параллельный ему список их рейтингов в лестнице. В примере 17.2 метод `exchange` вызывается, когда один игрок побеждает другого, который выше него по рейтингу на одну позицию. Список `players` остается неизменным, но рейтинги игроков меняются местами.

Пример 17.2. *Представление турнирной лестницы параллельными списками*

org/jgcbook/chapter17/E_records/Ladder

```
record Person(String name, int age) {}
public class Ladder {
  final private List<Person> players;
  final private List<Integer> rankings;
  public Ladder(List<Person> players, List<Integer> rankings) {
    this.players = players;
    this.rankings = rankings;
  }
  public List<Integer> getRankings() {
    return rankings;
  }
  public void exchange(Person winner, Person loser) {
    int winnerLocation = players.indexOf(winner);
    int loserLocation = players.indexOf(loser);
    int winnerRanking = rankings.get(loserLocation);
    rankings.set(winnerLocation, winnerRanking);
    rankings.set(loserLocation, winnerRanking + 1);
  }
  public static void main(String[] args) {
    final Person george = new Person("george", 33);
    final Person john = new Person("john", 31);
    final Person paul = new Person("paul", 30);
    Ladder ladder = new Ladder(new ArrayList<>(List.of(george, john, paul)),
                    new ArrayList<>(List.of(2, 3, 1)));
    ladder.exchange(john, george);
    assert ladder.getRankings().equals(List.of(3, 2, 1));
  }
}
```

Недостатки этого представления становятся очевидны, когда мы начинаем рассматривать другие операции с лестницей. Например, чтобы отсортировать

[1] В других языках параллельные массивы могут давать выигрыш в производительности, но этот выигрыш невелик по сравнению с другими проблемами, да и в любом случае его нет вовсе для списков Java, элементами которых всегда являются объекты, а не примитивные значения.

данные по-другому – скажем, по возрасту игроков, – нужно сначала отсортировать список `players`, записав проделанные изменения в вектор перестановки (т. е. новый список, в котором значение в позиции с некоторым индексом соответствует новой позиции элемента, ранее имевшего этот индекс). Затем перестановку можно использовать, чтобы внести соответствующие изменения в список `rankings`. Более привлекательная альтернатива – перенести данные в отсортированное отображение в предположении, что оно может оказаться лучшим представлением; однако у нее есть недостаток – какое бы отсортированное отображение вы ни выбрали, в нем будет всего один ключ сортировки.

Запись даст нам больше гибкости.

org/jgcbook/chapter17/E_records/RecordLadder

```
record PlayerRanking(Person player, int ranking) {}
public class RecordLadder {
  final private List<PlayerRanking> ladder = new ArrayList<>();
  public RecordLadder(List<Person> players, List<Integer> rankings) {
    for (int i = 0; i < players.size(); i++) {
      ladder.add(new PlayerRanking(players.get(i), rankings.get(i)));
    }
  }

  public List<PlayerRanking> getLadder() {
    return ladder;
  }
  ...
}
```

Теперь метод `exchange` стал чуть более громоздким, потому что пришлось определить новый вспомогательный метод, заменяющий `indexOf` как способ поиска элемента в лестнице:

org/jgcbook/chapter17/E_records/RecordLadder

```
private int locate(Person p) {
  for (int i = 0; i < ladder.size(); i++) {
    if (ladder.get(i).player().equals(p)) return i;
  }
  return -1;
}
public void exchange(Person winner, Person loser) {
  int winnerLocation = locate(winner);
  int loserLocation = locate(loser);
  int winnerRanking = ladder.get(loserLocation).ranking();
  ladder.set(winnerLocation, new PlayerRanking(winner, winnerRanking));
  ladder.set(loserLocation, new PlayerRanking(loser, winnerRanking + 1));
}
```

Зато сортировка теперь тривиальна.

org/jgcbook/chapter17/E_records/RecordLadder

```
public static void main(String[] args) {
  final Person peter = new Person("peter", 33);
  final Person paul = new Person("paul", 31);
  final Person mary = new Person("mary", 30);
  RecordLadder ladder = new RecordLadder(new ArrayList<>(List.of(peter, paul,
mary)),
```

```
        new ArrayList<>(List.of(2, 3, 1)));
    ladder.exchange(paul, peter);
    assert ladder.getLadder().get(ladder.locate(paul)).ranking() == 2 &&
        ladder.getLadder().get(ladder.locate(peter)).ranking() == 3;
    ladder.getLadder().sort(Comparator.comparing(pp -> pp.player().age()));
      assert  ladder.getLadder().stream().map(PlayerRanking::player).toList().
equals(
        List.of(mary, paul, peter));
}
```

Неудивительно, что сортировка, равно как и другие операции, проще в версии с записями, потому что записи явно выражают связи между элементами данных, не требуя никакой дополнительной работы по ее поддержанию. Этим они принципиально отличаются от списков, где связь неявна и должна поддерживаться вручную.

Используйте записи в качестве составных ключей

Записи дают изящное решение общей проблемы представления коллекций объектов, которые уникально идентифицируются подмножеством своих свойств. Например, рассмотрим, как представить библиотеку, состоящую из экземпляров класса `Book`:

```
class Book {
  private String title;
  private String author;
  private String publisher;
  private int pageCount;
  ...
}
```

Экземпляры `Book` однозначно идентифицируется комбинацией `title` и `author`. Предположим, что основной сценарий использования – быстро найти издателя книги `Book`, заданной автором и названием. На каких полях следует определить равенство и какую коллекцию `Book` выбрать, чтобы оптимально решить эту задачу?

Для большой коллекции хешированная структура данных была бы предпочтительнее структуры с линейным временем поиска, например списка. Ради скорости мы хотели бы уменьшить число полей, с помощью которых определяется равенство и хеш-код, а поскольку сочетание `author` и `title` уникально определяет книгу, кажется разумным только эти поля и использовать. Особенно это важно в типичном случае, когда существует много полей, помимо тех, что уникально определяют экземпляр.

Итак, одно из возможных решений этой задачи – снабдить `Book` такими методами:

```
class Book {
  ...
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Book book)) return false;
    if (!title.equals(book.title)) return false;
    return author.equals(book.author);
  }
```

```
  public int hashCode() {
    return Objects.hash(title, author);
  }
}
```

Но теперь рассмотрим, как, зная автора и название, мы сформулировали бы запрос к множеству объектов `Book` на предмет наличия в нем такой книги. Единственный способ – создать фиктивный объект `Book`, содержащий только автора и название (все остальные поля безразличны). Теперь в системе есть два вида книг: настоящие и фиктивные. Легко видеть, что это может привести к ошибкам. Хуже того, такой дизайн плохо приспособлен для модификации. Предположим, к примеру, что в класс `Book` добавлено поле, содержащее номер издания:

```
class Book {
  private String title;
  private String author;
  private int edition;
  private String publisher;
  private int pageCount;
  ...
}
```

Теперь уникальным идентификатором объекта `Book` является комбинация названия, автора и издания. А в дополнение к необходимости поиска объекта по этим трем полям может быть добавлено требование быстро найти все издания данной книги.

Эту проблему можно решить, определив запись, содержащую составной ключ:

```
record TitleAuthor(String title, String author) {}
```

В первой части примера, когда объект `Book` однозначно определяется названием и автором, коллекцию таких записей можно смоделировать отображением:

```
Map<TitleAuthor,Book>
```

что обеспечивает быстрый поиск по известным ключевым полям. После добавления поля `edition` представление можно изменить (количество изданий, скорее всего, будет невелико):

```
Map<TitleAuthor,List<Book>>
```

Конечно, того же эффекта можно достичь и без записей, определив класс `TitleAuthor`, но большое преимущество записей в том, что они краткие и обладают именно теми характеристиками, которые нам нужны: немодифицируемостью и подходящим определением `equals` и `hashCode`.

Управляйте изменяемостью записей

Обсуждая рекомендуемую практику управления изменяемостью в разделе «Не забывайте о “владельцах” коллекций» выше, мы имели дело с общим случаем. У записей же есть некоторые специальные свойства, которые упрощают применение описанных здесь принципов. Записи не являются неизменяемыми, как иногда думают, а коллекции, хранящиеся в их полях, даже

не являются автоматически немодифицируемыми, но их сравнительно легко сделать таковыми. Первый шаг, упомянутый в том разделе, – создать защитную копию данных, передаваемую объекту. Например, предположим, что мы хотим хранить моментальные снимки проектов, в которых содержатся множества задач, ассоциированных с ними на указанную дату. Будучи переносчиком данных, запись идеально подходит для этой цели. Мы сразу можем забыть об изменяемости полей, потому что отсутствуют методы установки. А раз так, то нет необходимости в защитном копировании имени или даты, поскольку они представлены неизменяемыми типами. Единственное оставшееся требование – создать защитную копию входных данных для множества задач. Компактный конструктор позволяет сделать это очень кратко:

```
record ProjectWithSet(String projectName, LocalDate date, Set<Task> tasks) {
  ProjectWithNavigableSet {
    tasks = Set.copyOf(tasks);
  }
}
```

В этом случае у метода `Set::copyOf` двоякая роль. Созданная им защитная копия немодифицируема, поэтому никакого дополнительного обертывания не требуется. А в отсутствие методов `set` и другого кода, позволяющего изменить содержимое `tasks`, одну и ту же не измененную копию можно многократно возвращать при вызове метода `get`.

Метод `copyOf` – вспомогательное средство, доступное не для каждой коллекции. Если бы мы хотели, чтобы моментальный снимок проекта хранил задачи в отсортированном виде, то могли бы определить запись, которая принимает коллекцию задач `NavigableSet`. Но в этом случае копирующий конструктор, которым мы воспользовались бы для создания защитной копии, не обеспечивает немодифицируемости, поэтому нужен дополнительный шаг – обернуть модифицируемую копию, сделав ее немодифицируемой.

И ссылка, возвращаемая с помощью метода `get`, должна быть немодифицируемой копией.

```
record ProjectWithNavigableSet(String name, NavigableSet<Task> tasks) {
  ProjectWithNavigableSet {
    var defensiveCopy = new TreeSet<>(tasks);
    tasks = Collections.unmodifiableNavigableSet(defensiveCopy);
  }
}
```

Если подлежащие хранению изменяемые объекты не являются коллекциями, то защитное копирование входных данных в конструкторе может потребовать больше работы. А если немодифицируемая обертка недоступна, то, возможно, придется переопределить акцессор, так чтобы он возвращал защитную копию при каждом вызове.

Избегайте унаследованных реализаций

Когда такой большой и сложный API, как каркас коллекций, развивается на протяжении многих лет, неизбежно рождаются и эволюционируют новые идеи, иногда замещая старые. Так, появление в Java 2 каркаса коллекций привело к устареванию коллекций из прежних версий: `Vector`, `Stack`

и `Hashtable`. Мы не станем обсуждать дефекты их дизайна, потому что маловероятно, что вы когда-нибудь столкнетесь с ними, разве что несчастливая судьба заставит вас взаимодействовать с Java-кодом, написанным на заре истории, до 1998 года.

Вместо этого мы обсудим некоторые реализации из самого каркаса коллекций, которых также лучше избегать.

Избегайте синхронизированных оберток коллекций

В разделе «Синхронизированные коллекции и итераторы с быстрым отказом» главы 9 мы видели, как отказ от потокобезопасности в новых коллекциях, появившихся в Java 2, – `ArrayList`, `HashSet` и т. д., — потребовал создания уровня потокобезопасности, который можно было бы добавить к этим коллекциям в случае необходимости.

Этот уровень предоставляется фабричными методами класса `Collections`, которые обертывают переданную коллекцию объектом, действующим как глобальная блокировка, т. е. разрешающим доступ к коллекции только одному потоку в каждый момент времени. До версии Java 5 эти объекты были единственной аппроксимацией потокобезопасности в каркасе коллекций. Например, чтобы создать синхронизированный список `List`, следовало обернуть экземпляр `ArrayList`. Объект-обертка, возвращенный методом `Collections::synchronizedList`, реализует интерфейс, делегируя вызовы методов переданной вами коллекции, но вызовы самой обертки синхронизированы.

Для иллюстрации того, как это работает, в классе `SynchronizedArrayStack` показана очень простая синхронизированная обертка интерфейса `Stack` из примера 9.1 (здесь она для удобства повторена).

org/jgcbook/chapter17/F_legacy/SynchronizedArrayStack

```
interface Stack {
  public void push(int elt);
  public int pop();
  public boolean isEmpty();
}

class SynchronizedArrayStack implements Stack {
  private final Stack stack;
  public SynchronizedArrayStack(Stack stack) {
    this.stack = stack;
  }
  public synchronized void push(int elt) { stack.push(elt); }
  public synchronized int pop() { return stack.pop(); }
  public synchronized boolean isEmpty() { return stack.isEmpty(); }
}
```

Чтобы получить потокобезопасный `Stack`, можно написать:

```
Stack threadSafeStack = new SynchronizedArrayStack(new ArrayStack());
```

Это предпочтительная идиома использования синхронизированных оберток; единственная ссылка на обернутый объект хранится внутри самой обертки, поэтому все обращения к обернутому объекту будут синхронизированы одной и той же блокировкой – той, что принадлежит обертке.

Безопасное использование синхронизированных коллекций

Даже класс, подобный `SynchronizedArrayStack`, методы которого полностью синхронизированы, а сам он потокобезопасен, все равно следует использовать с осторожностью в конкурентной среде. Например, следующий клиентский код не является потокобезопасным:

```
Stack stack = new SynchronizedArrayStack(new ArrayStack());
...
// не делайте так в многопоточной среде
if (!stack.isEmpty()) {
  stack.pop(); // возможно исключение IllegalStateException
}
```

Исключение было бы возбуждено, если бы последний элемент был удален другим потоком в момент между вычислением `isEmpty` и выполнением `pop`. Это пример типичной ошибки конкурентного программирования, которая называется по-разному: *проверь-потом-действуй* (test-then-act) или *TOCTOU* (time-of-check to time-of-use – между моментом проверки и моментом использования). Поведение программы в этом случае зависит от информации, которая могла уже устареть. Чтобы избежать этого, проверка и действие должны выполняться атомарно. Для синхронизированных оберток коллекций это можно гарантировать с помощью *блокировки на стороне клиента*. Например, если бы мы решили, что `Stack` должен расширять `Iterable`, так чтобы был возможен обход `SynchronizedArrayStack`, то документация рекомендует (там приведена другая, но эквивалентная форма итерирования) писать клиентский код, как показано ниже, и предупреждает, что иначе может иметь место недетерминированное поведение:

```
synchronized(stack) {
  for (int element : stack) {
    // обработать элемент
  }
}
```

Отсюда должно быть ясно, что потокобезопасность, обеспечиваемая блокировкой на стороне клиента, обходится дорого. Поскольку другие потоки не могут использовать никакие методы коллекций, пока действие не завершено, охрана длительного действия (скажем, обхода всего массива) плохо отразится на пропускной способности и может даже свести на нет все преимущества конкурентности.

Еще один недостаток блокировки на стороне клиента заключается в том, что все клиенты должны использовать одну и ту же блокировку. По соглашению используется та же блокировка, что в методах синхронизированной коллекции, т. е. сам объект-обертка. Проблема в том, чтобы все клиенты соблюдали эту дисциплину, а такого рода соглашения печально известны своей хрупкостью в процессе сопровождения. И в общем случае, когда клиенту передается ссылка на тип интерфейса (в данном случае `Stack`), он не может сказать, нуждается ли объект, на который эта ссылка ведет, в блокировке при выполнении обхода и составных операций.

При некоторых обстоятельствах синхронизированные коллекции все еще полезны; например, если нужен список, который сравнительно часто модифицируется (так что `CopyOnWriteArrayList` не годится), но за который нет сильной конкуренции, или если приложению нужна какая-то возможность синхронизированных коллекций, скажем монопольная блокировка. В таких случаях обыч-

но следует инкапсулировать блокировку на стороне клиента в верхнеуровневом объекте приложения, который координирует блокировку синхронизированной обертки (хотя в этом случае синхронизированная обертка может предоставлять преимущества несинхронизированной коллекции). Однако, если не считать таких сценариев, конкурентные коллекции почти всегда предпочтительнее.

Избегайте LinkedList

В табл. 14.2 приведен прекрасный пример причин, по которым временную сложность отдельных операций структуры данных неразумно рассматривать в изоляции. Глядя на таблицу, создается впечатление, что класс `LinkedList` должен выбираться в качестве предпочтительной реализации списка, когда основной сценарий – удаление или добавление элементов в начале списка или близко к нему, например при реализации стека. В классе `LinkedList` такие операции, рассматриваемые изолированно, имеют постоянную временную сложность, тогда как в `ArrayList` они требуют линейного времени из-за необходимости поддерживать непрерывность массива и индексирование с нуля – для этого все элементы с большими индексами перемещаются в начало коллекции. Однако этот факт следует рассматривать вкупе с другими характеристиками `LinkedList`, а взятые все вместе они склоняют чашу весов в пользу других реализаций.

Во-первых, `LinkedList` потребляет очень много памяти. Поскольку он дважды связанный, в каждом узле нужно хранить две ссылки – на предыдущий и на следующий узел. Таким образом, каждый узел требует по меньшей мере 24 байта (точное количество зависит от того, как JVM представляет указатели на объекты, и от выравнивания памяти) по сравнению с типичными накладными расходами 4 байта для `ArrayList`. Даже принимая во внимание тот факт, что базовый массив `ArrayList` потребляет больше памяти, чем требуется самой коллекции, это все равно дает `ArrayList` примерно трехкратный выигрыш в потреблении памяти. Это может быть существенно для очень больших коллекций, а помимо потребления больших объемов кучи и, возможно, физической памяти, еще и ведет к повышенным накладным расходам на сборку мусору и увеличенному времени приостановки.

Впрочем, даже для сравнительно небольших коллекций повышенное потребление памяти существенно, особенно если все новые и новые выделенные узлы оказываются сильно разбросаны в памяти. Как было описано в разделе «Память» главы 9, ключевым элементом современных аппаратных архитектур является иерархия памяти, в которой процессорные кеши играют важнейшую роль. Обновление кеша после непопадания в него может обходиться в сотни тактов процессора, поэтому минимизация непопаданий в кеш – существенная часть оптимизации производительности. Единицы хранения в кеше, называемые строками кеша, имеют ограниченную емкость – как правило, 64 байта, – а их содержимое всегда отражает блок соседних адресов памяти. Так что в случае удачи одна строка кеша могла бы содержать 16 ссылок на элементы `ArrayList` – по сравнению с двумя (и это в лучшем случае) узлами `LinkedList`. Так как во многих практических ситуациях непопадание в кеш является преобладающим фактором производительности программы, это можно считать огромным недостатком `LinkedList`.

Излюбленный аргумент в пользу `LinkedList` – способность добавлять и удалять один элемент за постоянное время. Однако в любом реалистичном сцена-

рии, где встречаются эти операции, всегда приходится добираться до позиции списка либо по индексу, либо линейным поиском. Даже в случае линейного поиска `ArrayList` показывает лучшую производительность (по причинам, описанным выше), а если его можно отсортировать, то двоичный поиск в `ArrayList` выполняется за время $O(\log N)$. На практике стоимость обхода большой коллекции намного перевешивает стоимость одиночной модификации, даже для `ArrayList`, когда для поддержания непрерывности элементов используется обращение к `System.arraycopy`.

Один из сценариев, который часто называют в качестве убедительной причины сохранить `LinkedList` в арсенале разработчика, – его использование в качестве стека или очереди. Но в каркасе коллекций есть более подходящие альтернативы для этой цели: `ArrayDeque`, а начиная с Java 21, еще и обращенный `ArrayList`.

Настраивайте коллекции с помощью абстрактных классов

В книге «Effective Java» (2017, совет 18) Джошуа Блох выдвигает аргументы против безоглядного использования наследования реализации как инструмента повторного использования. Самый убедительный из них – *проблема хрупкого базового класса*, смысл которой в том, что зависимость подкласса от деталей реализации суперкласса ведет к неожиданному поведению в случае, если реализация суперкласса изменяется. Следующий упрощенный код иллюстрирует эту проблему:

```
class Super {
  public void foo() { System.out.println("вызван метод суперкласса"); }
  public void bar() { System.out.println("вызван метод суперкласса"); }
}

class Sub extends Super {
  @Override
  public void bar() {
    super.foo();
    System.out.println("вызван метод подкласса");
  }
}
```

Вызов `Sub::bar` печатает:

```
вызван метод суперкласса
вызван метод подкласса
```

но программист, сопровождающий `Super::foo`, мог бы изменить реализацию, следуя принципу DRY (не повторяйся):

```
public void foo() { bar(); }
```

И хотя поведение `Super` осталось прежним после такого изменения реализации, вызов `Sub::bar` теперь приводит к бесконечной рекурсии, потому что в его объявлении переопределяется объявление `bar` в `Super`. Проблема хрупкого базового класса имеет одну и ту же причину во всех своих вариациях: некоторая деталь реализации класса раскрывается наследующим ему классам, поэтому

их поведение может неожиданно измениться после изменения реализации суперкласса. Поэтому Блох (Bloch 2017, совет 17) настоятельно рекомендует одно из двух: либо класс должен быть специально спроектирован для наследования, либо наследование ему следует предотвращать. Будучи первым главным проектировщиком каркаса коллекций, Блох прошел длинный путь к реализации этого принципа: хотя конкретные классы каркаса и не спроектированы для наследования, и не предотвращают его, каркас все же предоставляет абстрактные классы, которые специально предназначены для этой цели и дают отличную основу для настройки под себя имеющихся классов коллекций.

Пример поможет лучше разобраться в этой идее. Допустим, что приложению нужна коллекция, которая ведет себя как `ArrayList`, но с одним новым инвариантом: один элемент, переданный на этапе конструирования, должен всегда оставаться в нулевой позиции (с индексом 0). Первая мысль – что нужно переопределить оба перегруженных метода `remove`, а также `clear`, `removeIf` и `removeAll`. Но поскольку значение фиксированного элемента изменять запрещено, метод `set` также необходимо переопределить. Затем вы могли бы вспомнить, что имеется перегруженный вариант `add`, позволяющий вставить новый элемент в нулевую позицию. А потом еще нужно вспомнить, что итераторы, возвращенные методами `iterator` и `listIterator` предоставляют метод `remove`, который должен учитывать инвариант коллекции. И итераторы списка предоставляют также метод `add`; в нем тоже нужно производить проверку на вставку в нулевую позицию.

Очевидно, эти изменения оставляют большой простор для ошибок и упущений. Но по-настоящему убедительный аргумент против такой стратегии – проблема хрупкого базового класса. Допустим, вы произвели эти изменения до Java 21. Даже зная, что в этой версии появились новые упорядоченные коллекции, предоставляющие класс `ArrayList` с новыми методами `addFirst`, `removeFirst` и `removeLast`, вы могли бы не без оснований надеяться, что они реализованы поверх уже переопределенных вами методов, а значит, не нарушат инвариант. Если вы и правду понадеялись на это, то будете жестоко разочарованы: авторы оптимизировали `removeFirst`, так что он не вызывает `remove`. Вместо этого он опускает проверку допустимости индекса и сразу вызывает `System::arraycopy`. В результате этой детали реализации новой версии `ArrayList` инвариант вашей коллекции окажется нарушен – если только вы не переопределите `removeFirst`.

Отсюда следует, что вы должны тщательно изучать все будущие версии `ArrayList`, чтобы подобные проблемы не повторялись. А поскольку наследование связывает подкласс с точным поведением методов его суперкласса, вам придется изучать не только такие новые методы суперкласса, как `removeFirst`, но и вообще все, чтобы убедиться, что ни рефакторинг, ни оптимизация не сделали ваш подкласс жертвой изменившегося поведения.

По счастью, имеется способ лучше. При проектировании каркаса коллекций Блох последовал собственному совету, сопроводив каждый из основных интерфейсов – `Collection`, `Set`, `List`, `Map` и `Queue` – соответствующим абстрактным классом: `AbstractSet`, `AbstractCollection` и т. д. Эти частичные реализации (в документации они называются заготовками (skeletal implementation)) специально предназначены для наследования, но, что самое главное, предлагают реализации почти всех методов соответствующего интерфейса. Что до наше-

го примера, в документации `AbstractList` сказано, что, для того чтобы расширить его с целью создания конкретного модифицируемого списка, мы должны только переопределить `size` и индексированные варианты `get`, `set`, `add` и `remove`. В примере 17.3 показана минимальная реализация.

Пример 17.3. *Реализация пользовательского списка*

org/jgcbook/chapter17/G_customize_collections_using_the_abstract_classes/ListWithFixedFirstElement

```
public class ListWithFixedFirstElement<E> extends AbstractList<E>
    implements RandomAccess {

  private final List<E> backingList;

  public ListWithFixedFirstElement(E fixedElement) {
    this.backingList = new ArrayList<>();
    backingList.add(fixedElement);
  }

  @Override
  public E set(int index, E element) {
    if (index != 0) {
      return backingList.set(index, element);
    } else {
      throw new IllegalArgumentException("Cannot change fixed first element");
    }
  }

  @Override
  public void add(int index, E element) {
    if (index != 0) {
      backingList.add(index, element);
    } else {
      throw new IllegalArgumentException("Cannot change fixed first element");
    }
  }

  @Override
  public E remove(int index) {
    if (index != 0) {
      return backingList.remove(index);
    } else {
      throw new IllegalArgumentException("Cannot change fixed first element");
    }
  }

  @Override
  public E get(int index) {
    return backingList.get(index);
  }

  @Override
  public int size() {
    return backingList.size();
  }
}
```

Класс `ListWithFixedFirstElement` создает базовый список, в данном случае `ArrayList`, хотя любая реализация `List` была бы функционально эквивалентна. Он обрабатывает обращения к любому из переопределенных методов, проверяя сначала (в случае необходимости), не нарушает ли он инвариант класса, а затем делегируя работу базовому списку. Для остальных методов интерфейса `List` класс `AbstractList` предоставляет реализации, и все они в какой-то момент вызывают переопределенные методы. Поскольку в основе класса лежит массив, он реализует интерфейс `RandomAccess`, поэтому будут выбираться алгоритмы, основанные на индексировании, а не на обходе по итератору.

Эта реализация не подвержена проблеме хрупкого базового класса. Что касается вышеупомянутого примера, метод `removeFirst` не только имеет оптимизированную реализацию в `ArrayList`, но еще и реализован как метод по умолчанию, объявленный в интерфейсе `List`, чтобы поддержать подклассы, которые – в отличие от `ArrayList` – не были модифицированы с целью воспользоваться преимуществами его появления в новой версии. Этот метод по умолчанию вызывает `remove(0)` и потому сталкивается с проверкой инварианта в классе `ListWithFixedFirstElement`.

Вы, наверно, обратили внимание, что зависимость от реализаций, предоставляемых абстрактной коллекцией, означает, что эта техника может помешать вам воспользоваться оптимизациями, включенными в базовую коллекцию. Но зато она дает быстрый, простой и надежный способ настройки коллекции под свои нужды. В случае проблем с производительностью всегда остается возможность переопределить критичные методы, делегировав вызовы оптимизированным методам базовой структуры данных. Например, `AbstractList::addAll` повторно вызывает `add(size(), e)` для каждого элемента переданной коллекции. При таком вызове `size` производится отдельная проверка индекса для каждого элемента. Было бы легко переопределить `addAll` и делегировать вызов напрямую базовому `ArrayList`.

Заключение

Наставления, приведенные в этой главе, – это попытка извлечь некоторые уроки из долгого опыта, накопленного сообществом Java за время использования каркаса коллекций. Мы надеемся, что вы узнали о полезных приемах, и это поможет вам избежать некоторых подвохов при построении систем, использующих каркас.

В следующей и последней главе мы дадим краткий обзор дизайна и эволюции каркаса коллекций Java, а особенно некоторых его спорных моментов, в свете этого коллективного опыта. Очевидно, что базовый дизайн уже не претерпит изменений! Но все равно очень интересно проанализировать когда-то принятые решения и оценить, что из этого получилось на практике. Мы рассмотрим пять вопросов, все еще вызывающих споры.

Глава 18

Ретроспективный взгляд на дизайн

В начале главы 17 мы отметили, что долгий опыт работы сообщества Java с каркасом коллекций дает возможность извлечь некоторые практические уроки его использования. В этой главе тот же опыт послужит другой цели: дать обзор некоторых проектных решений и компромиссов, которые сформировали облик каркаса коллекций с самого начала и продолжают оказывать влияние на его эволюцию.

Фундаментальные проблемы дизайна каркаса коллекций

FAQ по дизайну API коллекций в Java (*https://docs.oracle.com/javase/8/docs/technotes/guides/collections/designfaq.html*), написанный в 1998 году, когда каркас был впервые явлен публике, начинается таким вопросом:

> Почему вы не поддерживаете неизменяемость прямо в интерфейсах основных коллекций, чтобы можно было упразднить факультативные операции (и исключение `UnsupportedOperationException`)?

На этот вопрос дается такой ответ:

> Это самое спорное проектное решение во всем API. Очевидно, что статическая (на этапе компиляции) проверка типов в высшей степени желательна и является нормой в Java. Мы бы поддержали ее, если бы полагали, что это практически осуществимо…

По прошествии более 25 лет первое предложение по-прежнему неоспоримо. В обзоре предложения по второму изданию этой книги можно найти множество комментариев по поводу этого решения:

> Даже скромные дополнения, такие как неизменяемые коллекции, ограничены требованием реализовать методы изменения API коллекций, а простое возбуждение `UnsupportedOperationException` является, мягко говоря, кошмарным хаком. Неизменяемость новых коллекций Java вообще не видна на уровне типов, и это отражает базовую несовместимость между в высшей степени императивным дизайном коллекций, которому вот уже больше 20 лет, и текущей тенденцией к стилю, более близкому к функциональному программированию.

Это замечание, как и FAQ, содержит предположение о том, что неизменяемость относится к *структурным* вещам – принципиально это вопрос о том,

в каком месте иерархии классов должны располагаться методы изменения. Их следует отличать от *семантических* вещей, а именно: насколько точно можно управлять изменениями и как передать информацию об изменяемости объекта его пользователям. В этом разделе мы попытаемся проанализировать эти вопросы в более широком контексте. Наша цель – не просто дать обзор отдельных проектных решений, а рассмотреть общие возможности дизайна любого каркаса коллекций, написанного на Java. Попутно мы надеемся поместить критические замечания к каркасу коллекций в исторический контекст.

Далее FAQ продолжает отвечать на первый вопрос, представив последствия замены единого универсального интерфейса наподобие `List` модифицируемыми, немодифицируемыми и неизменяемыми вариантами, например `ModifiableList`, *UnmodifiableList* и `ImmutableList`. Из этого обсуждения следует, что немодифицируемые интерфейсы – это просто интерфейсы, которым недостает методов изменения. Но интерфейс не может контролировать свои реализации; он может только выразить раскрываемые им возможности. Поэтому интерфейсы, раскрывающие только методы чтения, следует называть так, чтобы было понятно, что они предназначены для чтения. Далее мы будем использовать имена `ReadableList`, `ReadableSet` и т. д. до тех пор, пока не появится возможность исследовать способы добавления ограничений, которые могли бы обеспечить соблюдение или хотя бы уведомление о семантических свойствах немодифицируемости и неизменяемости.

Желательные характеристики каркаса коллекций

Для полноты картины ретроспективный обзор следует начать с перечисления целей проектирования. Каковы желательные характеристики каркаса коллекций, написанного на Java?

1. У него должна быть небольшая площадь поверхности. Это была приоритетная цель первоначального дизайна, поставленная в предположении, что каркасом будут ежедневно пользоваться миллионы разработчиков различной квалификации. Сделать его малым настолько, чтобы можно было освоить без усилий, значило повысить вероятность того, что разработчик поймет его достаточно хорошо для правильного применения в приложении. Отвечая на первый вопрос, FAQ по API каркаса коллекций аргументирует, что возможность до конца понять всю иерархию типов была бы утрачена, если бы были предоставлены немодифицируемые версии каждого интерфейса наряду со всеми теми, что необходимы частично модифицируемым и немодифицируемым представлениям.
2. Он должен быть легко расширяемым. Это требование вытекает из характеристики 1: по определению минимальный каркас не будет включать специализированные сценарии, поэтому путь к настройке под свои нужды должен быть прямым.
3. Как можно больше ошибок должно обнаруживаться на этапе компиляции, а не выполнения. В частности, мы хотели бы по возможности избежать ошибок времени выполнения, относящихся к разряду `UnsupportedOperationException`. Это проявление принципа разделения интерфейсов (Martin 2002): «Клиенту не следует раскрывать методы, в которых он не нуждается».

4. Каркас должен всячески поощрять возможность проверки соблюдения семантических ограничений. Примером ограничения, которое можно проверить, является уникальность в коллекции, которая оговаривается в контракте интерфейса `Set`. А примером ограничения, которое нельзя проконтролировать в каркасе коллекций, – требование, чтобы список `List` мог содержать только четное число элементов.
5. Свойства коллекции должны быть выводимы из типа ее интерфейса. Например, в каркасе коллекций пользователь `SortedSet` может быть уверен, что при обходе коллекции элементы будут возвращаться в естественном порядке или в порядке, определяемом компаратором множества.

Существенная часть проектирования каркаса – оценка относительной важности этих характеристик, когда они противоречат друг другу. Например, детальный API, спроектированный для удовлетворения характеристики 3, имел бы очень много специализированных интерфейсов, вследствие чего характеристика 1 – небольшой API – оказалась бы недостижимой. Аналогично расширяемый каркас, спроектированный с учетом характеристики 2, открыл бы возможность реализации неизменяемых или немодифицируемых интерфейсов с помощью изменяемых классов.

Обеспечение немодифицируемости путем создания подтипов

Предположим, что мы вознамерились спроектировать каркас коллекций, используя иерархию типов, чтобы поставить во главу угла характеристику 1, а остальные удовлетворив по мере возможности. Интуитивно кажется, что соблюдение принципа разделения интерфейсов диктует необходимость разделить интерфейсы чтения и записи коллекций (игнорируя пока сложности, связанные с частично допускающими запись представлениями, упоминаемые в разделе «Контракты» главы 9). Как можно было бы достичь такого разделения? Можно представить себе три возможности.

Допускающие чтение типы в качестве супертипов для типов, допускающих чтение и запись

Например, `ReadableCollection` мог бы быть супертипом `Collection`, который раскрывает только неизменяющие методы и позволяет `Collection` расширить `ReadableCollection` путем добавления методов изменения. Точно так же, `ReadableList` был бы супертипом `List`, `ReadableSet` – супертипом `Set` и т. д.; тем самым принцип разделения интерфейсов был бы соблюден. Но допускающий только чтение список не то же самое, что немодифицируемый; мы не можем заменить имя `ReadableList` именем *UnmodifiableList*, не определив – согласно принципу подстановки – `ArrayList` как разновидность *UnmodifiableList*, что, очевидно, неприемлемо.

Допускающие только чтение типы в качестве подтипов типов, допускающих чтение и запись

Такой дизайн выбран в библиотеке Guava (*https://github.com/google/guava/wiki/ ImmutableCollectionsExplained*). Например, `ImmutableCollection` (в терминологии «поверхностно неизменяемая», а в нашей терминологии «не-

модифицируемая») – это класс, наследующий `java.util.Collection`. Проектировщики Guava используют для представления типов классы, а не интерфейсы, потому что до появления запечатанных типов в Java 17 это был единственный способ предотвратить внешнее наследование. В результате, получив ссылку на неизменяемую коллекцию Guava, вы можете быть уверены, что она относится к классу, в котором все методы изменения возбуждают исключение `UnsupportedOperationException`, а не к классу, который наследует типу Guava и переопределяет эти методы; иными словами, налицо гарантия (поверхностной) неизменяемости.

Параллельные иерархии типов

Это модель коллекций в Eclipse. Каждый интерфейс существует в двух вариантах: `MutableCollection` в паре с `ImmutableCollection`, `MutableList` в паре с `ImmutableList` и т. д. Верхнеуровневые изменяемые интерфейсы расширяют соответствующие интерфейсы каркаса коллекций, т. е. `MutableCollection` расширяет `java.util.Collection` и т. д. Предоставляются методы преобразования изменяемых типов в неизменяемые, а неизменяемых типов в типы каркаса коллекций. В наших терминах неизменяемые интерфейсы – это интерфейсы, допускающие чтение: ничто не мешает реализовать их с помощью изменяемого класса. В этом каркасе имена неизменяемых типов служат указанием на намерение, а не гарантией неизменяемости.

Заметим, что все три возможности предполагают по меньшей мере удвоение числа интерфейсов в иерархии типов. В первом варианте верхнеуровневые интерфейсы следует называть `ReadableList`, `ReadableSet` и т. д., чтобы не нарушать принцип подстановки; имя указывает, что семантика этого дизайна совершенно отлична от неизменяемых интерфейсов, к которым мы стремились с самого начала. Второй вариант, реализованный в библиотеке Guava, гарантирует немодифицируемость некоторых интерфейсов, но, опять-таки подчиняясь принципу подстановки, должен ограничить расширяемость и предоставить реализации методов изменения, которые возбуждают исключение `UnsupportedOperationException`. Третий вариант добавляет сложности преобразования параллельных иерархий к выбору между расширяемостью и характеристикой 5 из предыдущего раздела; иными словами, клиент, получивший ссылку типа `ImmutableXxx`, не может знать, действительно ли объект ссылки немодифицируемый или принадлежит подклассу, в котором реализованы методы изменения.

Ограничения подтипов

Система типов – главное средство обнаружения ошибок на этапе выполнения, поэтому оно должно быть максимально точным при прочих равных условиях. Но из обсуждения в предыдущем разделе ясно, что на практике прочие условия никогда не бывают равными: каждый из трех исследованных вариантов имеет последствия, влияющие на семантику типов, расширяемость и размер каркаса. Кто чрезмерно полагается на иерархию типов для решения проблем проектирования API, тот сталкивается с другими проблемами – или игнорирует их. Назовем три важных проблемы такого рода.

- Многие желательные свойства системы невозможно представить системой типов Java. Например, чтобы объект был неизменяемым, весь его граф объектов, т. е. сам объект и все, на что он ссылается, прямо или косвенно, не должны изменяться (по крайней мере, наблюдаемым извне образом) после конструирования, как мы видели в разделе «Неизменяемость и немодифицируемость» главы 9. Чтобы средство проверки типов могло убедиться в этом на этапе компиляции, оно должно иметь возможность для каждого изменяемого компонента определить, что граф объектов имеет к нему монопольный доступ. Чтобы понять, насколько трудна эта проблема, нужно лишь заметить, что в основе почти всех коллекций лежит массив и что массивы в Java всегда изменяемы.

  Установление факта монопольного доступа тесно связано с проблемой анализа локальности (Gay and Steensgaard 2000), оптимизации компилятора, которая может быть осуществлена только на этапе выполнения, после того как было произведено динамическое связывание метода. Поэтому обеспечить неизменяемость в Java (и в любом другом объектно ориентированном языке) средствами статической типизации невозможно. Конечно, система типов может *представить* неизменяемость, не проверяя ее; например, в предложении по неизменяемым коллекциям для библиотеки расширения Kotlin `kotlinx` (*https://github.com/Kotlin/kotlinx.collections.immutable/blob/ master/proposal.md*) говорится, что «неизменяемость гарантируется только <...> контрактом; реализаторы обязаны обеспечить неизменность элементов коллекции после ее конструирования».

  Даже если ограничения не так амбициозны, для описания их средствами системы типов нужны экстраординарные меры. Например, рассмотрим указанную в комментарии рецензента возможность введения типа `Unmodifiable` для ссылки на немодифицируемые коллекции, появившиеся в Java 9. Должен ли этот тип возвращаться также методами создания немодифицируемых оберток из класса `Collections`? Если да, то клиент, получивший ссылку на коллекцию типа `Unmodifiable`, не смог бы определить, действительно ли коллекция немодифицируемая, или ее все же можно модифицировать из любого кода, имеющего ссылку на базовую коллекцию. Различение этих двух ситуаций средствами структуры типа требует дальнейших уточнений; выбрав этот путь, мы придем к очень детальной (а потому очень большой) иерархии типов. В следующем разделе мы рассмотрим альтернативы, которые можно лучше адаптировать к представлению семантических свойств.

- Если вы попытаетесь использовать систему типов, чтобы полностью устранить самое распространенное критическое замечание в адрес каркаса коллекций – об «ужасном хаке» `UnsupportedOperationException` – то столкнетесь с еще худшей – на иной взгляд – проблемой: взрывным ростом API, сопровождающим первый и третий из описанных в предыдущем разделе вариантов. На самом деле, как мы уже упоминали, FAQ по дизайну API коллекций заходит еще дальше в обосновании отсутствия немодифицируемых (в терминологии FAQ) интерфейсов, предлагая вообразить, сколько интерфейсов потребовалось бы, чтобы отразить всевозможные ограничения на

операции представлений коллекций. Например, список, возвращенный методом `Arrays::asList`, потребовал бы интерфейса `FixedSizeList`, который, в свою очередь, потребовал бы нового интерфейса итератора списка `FixedSizeListIterator` и т. д. Можно было бы предложить не столь непримиримую позицию: например, предоставить немодифицируемые интерфейсы для немодифицируемых классов Java 9, но все же разрешить возбуждение `UnsupportedOperationException` из методов представлений[1].

Поскольку устранение факультативных методов средствами системы типов обходится так дорого, имеет смысл рассмотреть альтернативы и не зацикливаться только на уровне языка и API. В следующем разделе мы рассмотрим другие способы определить поведение коллекций, в том числе практики разработки, с которыми мы уж встречались (в разделе «Не забывайте о “владельцах” коллекций» главы 17), которые могут снять практические проблемы, связанные с исключением `UnsupportedOperationException`.

- Самое серьезное критическое замечание в комментарии рецензента – то, что немодифицируемость коллекций Java 9 не отражена в системе типов. У этой критики есть определенные основания: как мы видели, на самом деле возможно точно определить типы, допускающие только чтение, исключив методы изменения. Но часто забывают упомянуть, какую цену нужно заплатить за то, чтобы сделать это эффективно: в объектно ориентированном языке наследование обычно допускает возможность создать реализацию с добавленными методами изменения. Предотвращение этой возможности с помощью запечатанных интерфейсов или путем определения типов коллекций как финальных (как сделано в Guava) заодно исключает возможность расширения каркаса.

Очевидно, что статическая иерархия типов занимает важное место в дизайне каркаса. Но это не единственный инструмент, доступный проектировщику API; другие могут быть лучше приспособлены к предоставлению семантических свойств, нужных разработчикам на практике. Мы рассмотрим эти возможности в следующем разделе.

Не только статическая типизация

Рассматривая в целом способы описания дизайна API, мы видим, что ограничения статического типа являются лишь одним из инструментов, доступных проектировщику; из других отметим статический анализ, аннотации, имена типов и контракты. Имена типов важны, как показывает пример неизменяемых типов

[1] Это не парирует формальное критическое замечание о том, что существование факультативных методов нарушает принцип подстановки (см. раздел «Подтипизация и принцип подстановки» главы 2). На эту формальную критику можно дать формальный ответ: «принцип» – это на самом деле не нерушимый *принцип*, а *описание* систем, в которых любое использование супертипа, такого как интерфейс, можно заменить экземпляром одного из его подтипов, например реализующим классом, не изменяя смысла программы. В действительности, поскольку контракты интерфейсов специфицируют операции изменения как факультативные, принцип подстановки правильно описывает поведение представлений и немодифицируемых коллекций. Но истинный вопрос лежит в практической плоскости: отражает ли существование факультативных методов реальную проблему практической разработки программного обеспечения?

в библиотеке `kotlinx`, а многие простые семантические правила можно закодировать для статических анализаторов[1]. Но на практике самым важным дополнением к статической типизации являются контракты программных компонентов. На самом деле ограничения статических типов можно рассматривать как очень грубые контракты. Например, в Java коллекция может быть типизирована только как модифицируемая или немодифицируемая, тогда как контракт может специфицировать ее детальную семантику – что ее можно модифицировать, но только определенными способами или в определенных пределах. Граница между теми и другими может даже двигаться: до Java 21 порядок вхождения был специфицирован в контрактах классов и интерфейсов, реализующих его, но после появления упорядоченных коллекций он переместился в систему типов.

Контракты могут показаться слабым видом проверки по сравнению с проверкой типов на этапе выполнения, но на самом деле в каркасе коллекций они использовались с самого начала и не сталкивались с критикой. Интерфейс `Set` расширяет `Collection` без какой-либо дифференциации типов; методы `Set` идентичны методам `Collection`. Свойство уникальности элементов гарантируется только контрактами: общим контрактом всего интерфейса `Set` и контрактами его методов. Уникальность элементов сродни вышеупомянутым точным ограничениям на изменяемость в том смысле, что о свойстве коллекции можно сообщить только через контракт или формальную спецификацию.

Контракты, как мы их обычно представляем себе (и как они описаны в разделе «Контракты» главы 9) оговаривают пред- и постусловия для программного компонента. Предусловие представляет то, на что может рассчитывать компонент в части состояния среды и входных значений в начале выполнения, а постусловие – то, какое состояние среды и возвращаемое значение он обещает после завершения выполнения. Такой стиль контракта не всегда подходит для описания того вида ограничений, который мы обсуждали в разделе «Не забывайте о “владельцах” коллекций» главы 17. Рассмотрим такой код:

```
void foo(Collection<Object> c, Object x) {
  assert c.contains(x);
  bar();
  System.out.println(c.contains(x)); // true или false?
}
```

Учитывая, что `bar` может вызывать методы других объектов и что коллекция `c` может разделяться многими потоками, контракт с пред- и постусловиями для метода `bar` не может специфицировать, возможно ли изменение содержимого `c` в результате его выполнения. Однако `c` может управляться контрактом другого вида, который оговаривает, что коллекция требует от своей среды и как она поведет себя при доступе из кода, отличного от «владеющего» ей объекта. Это называется контрактом типа *упование-гарантия* (rely/guarantee), и, как и контракт с пред- и постусловием, он состоит из двух частей: *упования*, которое в примере выше описывает, какие изменения могут быть внесены в `c` во время выполнения `bar`, и *гарантии*, которая описывает исход взаимодействия `c` с ее средой во время выполнения `bar`.

[1] Обзор статических анализаторов Java и их использования см. в работе Valeev (2024, § 1.4).

Терминология «упование-гарантия» берет начало в дисциплине формальной спецификации (Jones 1983), которая предлагает точную семантику на уровне приложения для программных компонентов, работающих во взаимодействующей с ними среде. Она не годится для общего описания коллекций, точное поведение которых, очевидно, меняется от приложения к приложению. Но при построении программ, в которых коллекции передаются между компонентами системы, она очень полезна, если сами коллекции соблюдают минимальные условия упования и гарантии. В частности, должна быть возможность сообщить классу или методу, которому передается ссылка на коллекцию, информацию о том, что эта коллекция:

- *уповает* на то, что ее не изменяют (и только тогда будет работать правильно);
- *гарантирует*, что либо не будет модифицирована, либо что никакая часть ее объектного графа не будет изменена.

Эта информация помогает проектировщику компонента системы узнать, *что* разрешено делать с коллекцией, которая пересекает границу компонента, и *как* это делать. Например, программа, которой требуется многократно читать из коллекции должна знать, может ли она уповать на то, что переданная ей коллекция, останется неизменной, – в противном случае она должна будет сделать моментальный снимок и читать из него. Аналогично она должна знать, приемлемо ли изменение переданной ей коллекции для ее владельца (см. раздел «Не забывайте о “владельцах” коллекций» главы 17).

Более детальные ограничения находятся в серой зоне между коллекцией и семантикой приложения. Например, можно представить себе ограничение, согласно которому клиент может только добавлять записи в коллекцию (например, записей в журнале). Об этом ограничении можно сообщить, либо назначив коллекции специальный тип, допускающий только дописывание в конец, либо специфицировав это требование к программе иными средствами, например в виде правил статического анализа, в документации контракта на естественном языке или (с большей точностью, но и с большими усилиями) на языке формальных спецификаций.

Проектирование API должно начинаться с выбора семантических свойств, которые нужно представить; вопрос о том, *как* их представлять, важный, но вторичный. Ответ на этот вопрос даже может изменяться со временем, как мы видели во введении к главе об упорядоченных коллекциях. Статические ограничения типов и контракты – лишь пути к цели, а сама цель – передача информации о семантике API и ее проверка.

Объединение и разделение

В книге «The Principles of Classification and a Classification of Mammals» натуралист Джордж Симпсон (1945) писал:

> Разделители создают очень маленькие таксономические единицы – их критики говорят, что если они могут различить два животных, то помещают их в разные роды… а если не могут, то в разные виды… Объединители создают большие таксономические единицы – их критики говорят, что если хищник не является ни собакой, ни медведем, то они называют его кошкой.

Противоположные устремления объединения и разделения, обычно проявляющиеся в системах классификации, могут также описывать движущие силы проектирования программного обеспечения. Объединение порождает модели с мощными общими правилами, относительно простые и легкие для понимания, но с большим количеством специальных случаев. Разделение порождает более точные модели, но ценой потери общности, что ведет к большей сложности и, возможно, дублированию. Например, рассмотрим класс `IdentityHashMap`, который имеет много общих свойств с другими отображениями, но при этом отличается от них тем, что отношением эквивалентности является тождественность, а не равенство значений. Было ли правильным решением сделать `IdentityHashMap` расширением `Map`? С другими отображениями у него есть как сходства (механизм поиска), так и различия (отношение эквивалентности; см. раздел «Определение множества: отношения эквивалентности» главы 12). В настоящее время он объединен с другими отображениями; решение разделить их обеспечило бы большую точность, но ценой дублирования многих методов `Map` в новом типе.

Одной из основных целей первоначального дизайна каркаса коллекций было получение небольшого API, который легко понять, а это естественно привело к модели объединения. При повседневном использовании API обнажились недостатки компромиссов, внутренне присущих его дизайну, поэтому исключения из правил, свойственные объединенным в одну кучу классам каркаса, может вызывать раздражение. Но при объективном ретроспективном взгляде следует оценивать отдельные проблемы в контексте общего дизайна: в общем и целом была бы разделенная модель лучше, чем объединенная, которую выбрали проектировщики? Как и в цитате, приведенной в начале этого раздела, ответ зависит от вашего личного взгляда на сравнительную ценность обобщения и точности.

Резюме

Раздел получился длинным, потому что мы обсуждали конфликтующие цели проектирования каркаса коллекций, различные средства их достижения и общие философии проектирования, в рамках которых о них можно судить. Для большинства рассмотренных вопросов нет правильного или неправильного ответа, потому что любое проектное решение подразумевает компромиссы, которые в конечном счете являются оценочными суждениями. Мы полагаем, что принципы, изложенные в этой книге, особенно в главе 17, позволят вам безопасно обходить исключение `UnsupportedOperationException` и другие проблемы. Но как бы ни оценивать отдельные решения, сформировавшие облик каркаса коллекций, следует признать, что он стал значительным достижением, обязанным прежде всего Джошуа Блоху, проложившему путь в океане конфликтующих целей, которые мы описали. Не стараясь преуменьшить недостатки дизайна, мы можем признать, что они не оказали серьезного влияния на успехи каркаса, который применялся для построения бесчисленных информационных систем на протяжении последних тридцати лет. Идеи проектирования будут развиваться и дальше, как и должно быть, но это достижение никто не отменит.

Значения null

Мало найдется в языке программирования средств, которые вызывали бы столько ожесточенных споров – или, точнее, ругани, – чем `null`. Поиск в вебе статей с перечислением проблем, вызванных `null`, возвращает множество постов в блогах на пять-десять тысяч слов с такими названиями, как «Худшая ошибка в информатике». Статей, представляющих противоположную точку зрения, гораздо меньше. Но, несмотря на это, `null` вездесущ – и в прикладных программах, и в базах данных. К сожалению, каркас коллекций Java Collections Framework не остался в стороне от этой амбивалентности сложного и противоречивого употребления `null`.

Обвинение против `null` состоит из двух частей. Более общее относится к системам типов, которые не исключают значений `null` в качестве допустимых ссылок[1]. Программы на любом языке, допускающем `null` в качестве значения ссылки, должны включать проверки на `null` в качестве специального случая. Пожалуй, самое часто встречающееся исключение в Java – `NullPointerException`, результат попытки разыменовать нулевую ссылку, но есть и другие ситуации, когда с `null` нужно обращаться специальным образом. Например, в процессе линейного зондирования, используемого в `IdentityHashMap` (см. рис. 15.3), нужно уметь различать ключи, равные `null`, и неиспользуемые позиции таблицы. Поэтому `null` в качестве значения ключа, следует внутри кода заменять специальным ненулевым объектом и переходить от `null` к этом объекту и обратно в процессе сохранения и извлечения. Эта уловка позволяет избежать главной проблемы, касающейся `null`, в дизайне каркаса коллекций Java: он часто играет две роли – допустимого элемента коллекции и индикатора отсутствия элемента.

Несогласованную обработку `null` можно считать первородным грехом коллекций Java, предшествующим появлению на свет каркаса коллекций. Класс `Hashtable` из Java 1.0 (предтеча `HashMap`) не допускал `null` ни в качестве ключа, ни в качестве значения, тогда как `Vector` (предтеча `ArrayList`) принимал значения `null` в качестве элементов. Каркас коллекций, появившийся в Java 2, пытался разрешить эту проблему единообразно и поначалу принимал `null` в качестве допустимых значений почти во всех коллекциях. Эта попытка, пожалуй, сделала только хуже: в документации метода `Map::get` возвращаемое им значение описывается так:

> Если это отображение допускает значения `null`, то возврат значения `null` *необязательно* означает, что в отображении нет записи с таким ключом; возможно также, что имеется запись, отображающая ключ на `null`. Чтобы различить эти случаи, можно воспользоваться операцией `containsKey`.

Хотя контракт интерфейса `Map` явно признает, что в будущем реализации могут отвергать `null` в качестве допустимого элемента, в первоначальных реализациях каркаса коллекций – `HashMap`, `IdentityHashMap` и `LinkedHashMap` – `null` допускался.

Так обстояло дело, пока в версию Java 5 не был включен интерфейс `ConcurrentMap` и его реализация `ConcurrentHashMap`. В этот момент контракт `Map::get` обернулся

[1] Тони Хоар (Tony Hoare 2009), который ввел `null` в систему типов для ссылок в Algol W, называл это своей «ошибкой на миллиард долларов».

боком. Из документации следовало, что *после* обращения к `Map::get`, вернувшего `null`, клиент должен вызвать `Map::containsKey`, чтобы устранить двусмысленность. Но в конкурентной среде такую последовательность «проверь, затем действуй» (см. раздел «Безопасное использование синхронизированных коллекций» главы 17) нельзя надежно выполнить, не заблокировав коллекцию. Поэтому было принято решение запретить ключи, равные `null`, в `ConcurrentHashMap`, хотя интерфейс `ConcurrentMap` не переопределял контракт методов `Map` с целью исключить реализации, допускающие `null`.

Но в `ConcurrentHashMap` были добавлены составные атомарные операции `remove`, `replace` и `putIfAbsent`, которые в Java 8 были перенесены выше, в `Map`, одновременно с включением методов `merge`, `getOrDefault` и `compute*`. Как и более старые методы – `get`, `put` и `containsKey`, – `remove`, `replace` и `getOrDefault` считают `null` допустимым значением. Прочие методы в Java 8 не отличают значение `null` от отсутствия записи, но их поведение различается в случае попытки поместить в отображение `null`: если вычисление лямбда-выражения, переданного `compute`, `computeIfPresent` или `merge`, дает `null`, то существующая запись удаляется. (Однако `computeIfAbsent` не удаляет существующее значение `null`; это в каком-то смысле согласуется с его представлением о том, что такой записи не существует.)

Два метода Java 8, которые принимают значение, а не лямбда-выражение, – `putIfAbsent` и `merge`. Получив на входе значение `null`, `putIfAbsent` поместит его в отображение (хотя впоследствии не распознает его как существующее значение). Поведение метода `merge` при получении значения `null` худо-бедно согласуется с его нетерпимостью к `null`, но не с API отображения в целом, потому что он возбуждает исключение `NullPointerException`.

Коренная причина многих противоречий в обращении с `null` заключается в том, что в версии Java 5 каркас коллекций был объединен с конкурентной библиотекой, в которой были реализованы идеи, описанные в книге Doug Lea (1999) «Concurrent Programming in Java». Классы из этой библиотеки не считали `null` допустимым значением (по вышеупомянутой причине), поэтому перенос методов из такого нетерпимого к `null` класса, как `ConcurrentHashMap`, в существующий интерфейс `Map` неизбежно породил несогласованности в рамках одного интерфейса. Поэтому перед проектировщиками потоков в Java 8 и немодифицируемых коллекций в Java 9 встал нелегкий выбор – какой бы путь они ни выбрали, он мог лишь привнести имеющиеся несогласованности в каркас[1]. (В итоге для потоков было принято решение смириться с `null` в большинстве случаев, а для немодифицируемых коллекций – запретить `null`.)

Даже задним числом не удается вынести из этой истории ясный урок. Конкурентные интерфейсы и классы, появившиеся в Java 5, стали ключом к успеху Java и легли в основу бесчисленных конкурентных систем, написанных за 20 лет использования; невозможно представить современный Java без них.

[1] Альтернативный дизайн мог бы обойти этот выбор стороной, воспользовавшись системой типов, чтобы различить терпимые и нетерпимые к `null` коллекции точно таким же способом, каким `Collection` и `Set` раскрывают одни и те же методы, но с разной семантикой. Такое предпочтение разделению перед объединением (см. раздел «Объединение и разделение» выше) обменяло бы имеющуюся несогласованность на значительное увеличение количества интерфейсов.

Их перепроектирование с целью привести обращение с `null` в соответствие с существующими типами каркаса коллекций потребовало бы экстраординарных (и не оправданных результатом) усилий.

Некоторые современные проектировщики Java полагают, что терпимость к `null` всегда была ошибкой (*https://mail.openjdk.org/pipermail/lambda-libs-spec-experts/2012-September/000038.html*). Но когда ошибка проектирования вскрывается, должны ли новые средства предложить улучшенное поведение ценой несогласованности, или ради согласованности нужно смириться с ошибкой навечно? Этот вопрос снова и снова возникает в ходе эволюции крупных программных платформ типа Java. Иногда предпочтение отдается продлению несогласованности. Разумеется, есть и третья возможность: мы можем и включить улучшенное поведение, и обеспечить согласованность, но ценой внесения несовместимых изменений. Однако такие изменения очень дорого обходятся пользователям, поэтому Java всегда отвергал эту стратегию.

Несогласованность с equals

В разделе «Согласованность с equals» главы 3 мы отметили, что в документации интерфейса `Comparable` рекомендуется делать метод `compareTo` *согласованным с `equals`*, т. е. для двух объектов метод `equals` должен возвращать `true` тогда и только тогда, когда `compareTo` считает их равными. Но в документации ничего не сказано ни о причинах, по которым можно отклониться от этой рекомендации, ни о последствиях такого решения. В разделе «Согласованность с equals» в качестве контрпримера приведен класс `BigDecimal`. Экземпляр `BigDecimal` состоит из двух частей: числового значения и точности, с которой оно хранится. Поведение округления зависит от последней, поэтому равенство двух экземпляров `BigDecimal` зависит от обеих частей. Например, результат деления 2 на 3 равен 2/3; чтобы записать это число в виде десятичной дроби, его необходимо округлить. Результат деления `BigDecimal` округляется до точности делимого, как показывают следующие два утверждения `assert`:

```
assert new BigDecimal("2.0").divide(BigDecimal.valueOf(3), HALF_UP)
    .equals(new BigDecimal("0.7"));
assert new BigDecimal("2.00").divide(BigDecimal.valueOf(3), HALF_UP)
    .equals(new BigDecimal("0.67"));
```

Два значения, удовлетворяющие отношению равенства, должны вести себя одинаково, поэтому экземпляры, созданные с помощью `new BigDecimal("2.0")` и `new BigDecimal("2.00")`, не должны быть равны, и действительно значением `BigDecimal("2.0").equals(BigDecimal("2.00"))` является `false`. Но, разумеется, представляемые ими числовые значения равны, так что `BigDecimal("2.0").compareTo(BigDecimal("2.00"))` равно `0`, и структуры данных, в которых используется числовое сравнение, а не отношение равенства, не смогут их различить. Например, следующее утверждение `assert` выполняется без ошибок:

```
assert new TreeSet<>(Set.of(new BigDecimal("2.0")))
    .contains(new BigDecimal("2.00"));
```

но, конечно же,

```
assert ! new HashSet<>(Set.of(new BigDecimal("2.0")))
    .contains(new BigDecimal("2.00"));
```

Поскольку равенство множеств вычисляется проверкой принадлежности, симметрия вполне может быть утрачена при сравнении множеств с разными отношениями эквивалентности:

```
Set<BigDecimal> hs = new HashSet<>(Set.of(new BigDecimal("2.0")));
Set<BigDecimal> ts = new TreeSet<>(Set.of(new BigDecimal("2.00")));
assert ts.equals(hs);
assert ! hs.equals(ts);
```

Это тот вид «странного» поведения, который, как предупреждает документация `Comparable`, является следствием несогласованности естественного порядка с `equals`.

Класс `BigDecimal` необычен в том смысле, что естественный порядок не согласован с `equals`. С другой стороны, очень легко создать компаратор с несогласованным методом `compare`. Это имеет место в широко распространенном сценарии, когда коллекцию объектов нужно поддерживать упорядоченной по одному полю. Например, чтобы поддерживать коллекцию книг – объектов `Book` (см. раздел «Используйте записи в качестве составных ключей» главы 17) – упорядоченной по числу страниц, мы могли бы использовать упорядоченную коллекцию:

```
new TreeSet<Book>(Comparator.comparingInt(Book::getPageCount))
```

Печальные последствия на своей шкуре испытали бесчисленные разработчики: в результате добавления двух объектов `Book` с одинаковым числом страниц второй считается дубликатом и «пропадает».

Компаратор можно перепроектировать, но естественный порядок фиксирован. Поэтому, чтобы сохранить значения типа `BigDecimal` в упорядоченной коллекции, необходимо определить компаратор, индуцирующий *полный порядок* на множестве значений, т. е. отношение порядка, определенное для любой пары значений. Конечно, можно было бы определить компаратор, который сделал бы значение `BigDecimal`, соответствующее `"2.0"`, большим – или меньшим – значения, соответствующего `"2.00"`, но не очевидно, насколько полезным был бы такой порядок.

Является ли эта проблема дефектом проектирования упорядоченных коллекций, класса `BigDecimal` или того и другого сразу? Она, безусловно, объясняет совет определять порядок, согласованный с `equals`, всюду, где возможно, но мы видели, что, по крайней мере, для `BigDecimal` разумно определить такой порядок невозможно. Тогда должны ли упорядоченные коллекции использовать равенство в качестве отношения эквивалентности? Если да, то упорядоченные коллекции смогут содержать значения `BigDecimal`, соответствующие и `"2.0"`, и `"2.00"`, когда контракт любого метода этих коллекций предполагает существование полного порядка на множестве элементов.

Проблема вызвана не этими проектными решениями, а ни на чем не основанным ожиданием (подкрепленным контрактом), что для любого множества `Set` метод `equals` будет использоваться в качестве отношения эквивалентности. Мы обсуждали эту проблему в разделе «Определение множества: отношения эквивалентности» главы 12; хотя выбор `equals` для определения отношения эквивалентности правилен для `HashSet` и большинства других реализаций множества, тем не менее это лишь один из кандидатов. Отношение эквивалентности для упорядоченных коллекций может, если оно несовместимо

с `equals`, рассматривать неравные объекты как дубликаты, тогда как отношение эквивалентности для множества, созданного из `IdentityHashMap`, рассматривает равные объекты как не дубликаты. Эти отношения эквивалентности просто трактуют одни и те же данные по-разному для разных целей. Проблемы возникают, когда множества (и отображения) обрабатываются в неправильных предположениях об их отношении эквивалентности. Например, применение бинарных операций типа `equals`, `addAll`, `removeAll` или `retainAll` к множествам с разными отношениями эквивалентности ведет к неожиданным результатам. И точно так же, когда вашему коду передают пустое множество `Set<String>`, вы, наверное, будете удивлены, обнаружив, что добавление строк `"one"` и `"two"` в это множество увеличивает его размер только на единицу. Но именно так и произойдет, если реализацией множества является класс `TreeSet`, в котором компаратор упорядочивает строки только по длине, так что любые две строки равной длины являются дубликатами. Проблема здесь в том, что код полагается на общий контракт `Set`, который не принимает во внимание различные отношения эквивалентности и потому нарушается классом `TreeSet`.

Как и в случае терпимости и нетерпимости к значениям `null`, дизайн множеств (и отображений) с разными отношениями эквивалентности на основе разделения, а не объединения помог бы избежать этих проблем, но снова ценой значительного увеличения количества типов.

Сравнение Object и E

В нескольких местах в этой книге мы отмечали кажущуюся странность сигнатур методов `Collection contains`, `remove` и `retain`, каждый из которых принимает параметр типа `Object`, и соответствующих ориентированных на коллекции версий `containsAll`, `removeAll` и `retainAll`, которые принимают параметр типа `Collection<?>`. Почему эти методы не параметризованы типом `E` или соответственно `Collection<E>`?

Недолгое размышление показывает, что эти методы, которые удаляют элементы из коллекции или проверяют принадлежность ей, даже потенциально не могут поставить под угрозу ее типобезопасность – в отличие от `add` и `addAll`, в сигнатурах которых действительно указан тип `E`. Стало быть, проектировщики *могли* безопасно выбрать эти типы параметров. Но почему они это сделали? Предлагались различные причины (в том числе самими проектировщиками).

- Обратная совместимость. Чтобы минимизировать количество несовместимых изменений, связанных с использованием дженериков, методы делались обобщенными, только если это было *необходимо* для обеспечения типобезопасности, как в случае `add` и `addAll`.
- Разрешение использования ограниченных джокерных типов. Предположим, что параметр-коллекция метода имеет тип `Set<? extends Foo>`. Если бы `contains` требовал параметр такого типа, то его вообще нельзя было бы использовать в этой ситуации, потому что нельзя было бы передать никакой аргумент типа `? extends Foo` (кроме `null`). А если параметр `contains` имеет тип `Object`, то его можно использовать с коллекциями любого типа.
- Возможность использования несвязанных типов. Самая убедительная причина – та, которую мы видели в разделе «Удаление элементов» гла-

вы 10: во многих ситуациях, в частности при вычислении перечисления двух коллекций, типы элементов могут быть никак не связаны. Тогда мы приводили пример вычисления пересечения коллекции элементов типа `PhoneTask` и коллекции элементов супертипа `Task`, но даже такая связь необязательна. Например, подумайте о вычислении пересечения двух коллекций объектов типа `List`. Два списка `List` равны, если они содержат одни и те же элементы, поэтому `retainAll` можно было бы использовать, чтобы оставить в `Collection<ArrayList>` только те `ArrayList`, которые равны какому-то элементу другой коллекции `Collection<LinkedList>`. Единственный тип параметра `retainAll`, при котором это возможно, – `Collection<?>`.

Компромисс между двумя вариантами этих типов методов коллекции весьма тонкий. С одной стороны, можно привести аргумент, что использование `Object` в качестве типа параметра означает, что будут пропущены некоторые ошибки, которые можно было бы обнаружить при более точной типизации. В конце концов, одно из основных преимуществ дженериков заключается в том, что точная типизация позволяет отловить больше ошибок на этапе компиляции. Но в случаях, подобных вышеупомянутым, когда точная типизация была бы неуместна, проблему пришлось бы решать с помощью дополнительных непроверенных приведений – один из основных источников неточности в системе параметризованных типов. Чтобы судить о том, правильное или неправильное решение приняли проектировщики, полезно знать, какие проблемы это решение было призвано решить.

Конкурентная модификация

Центральная проблема проектирования библиотеки коллекций – *конкурентная модификация*. Ранее мы обсуждали различные политики конкурентной модификации в JDK. Основные коллекции реализуют политику быстрого отказа, описанную в разделе «Синхронизированные коллекции и итераторы с быстрым отказом» главы 9. Конкурентные коллекции привносят две дополнительные политики конкурентной модификации: моментального снимка и слабо согласованную. Они обсуждаются в разделе «Механизмы работы конкурентных коллекций» главы 9. В этом разделе мы попробуем реконструировать рассуждение, благодаря которому в каркасе коллекций сегодня приняты именно эти политики конкурентной модификации, объяснить, почему для большинства неконкурентных коллекций выбрана политика быстрого отказа и насколько хорошо эти политики показали себя с момента внедрения.

Первыми классами коллекций, которые предшествовали каркасу коллекций, были `Vector` и `Hashtable`. Для их обхода использовался класс `Enumeration`. Ни в какой документации вообще не упоминалась политика конкурентной модификации. В зависимости от реализации конкурентная модификация могла приводить к пропуску элементов во время обработки, к обработке одного элемента несколько раз и даже к зацикливанию. Разработка каркаса коллекцией дала шанс исправить эту ситуацию.

Политика моментального снимка кажется самой простой для понимания и реализации. Увы, при этом она и самая дорогая. По сути дела, требуется делать полную копию коллекции при каждом создании итератора. По-

этому для большинства коллекций политика моментального снимка не годится. В каркасе коллекций она используется только для `CopyOnWriteArrayList` и `CopyOnWriteArraySet`, которые предположительно модифицируются редко.

Предполагалось, что основная существовавшая в то время альтернатива, слабо согласованная политика, будет применима как к неконкурентным, так и к конкурентным политикам. Главная проблема заключается в том, что семантика настолько слаба, что может приводить к непредсказуемому поведению, – столкнуться с таким в неконкурентной программе стало бы шоком. Рассмотрим однопоточную программу, которая обходит `HashSet`. Реализация `Iterator` хранит состояние, указывающее на текущую позицию итератора во внутренней таблице `HashSet`. Если элемент добавляется в середине обхода, то он может оказаться во внутренней таблице раньше или позже, в зависимости от своего хеш-кода. Хуже того, если добавление привело к перехешированию таблицы, то в новой таблице элементы могут оказаться в совершенно других позициях. Если обход продолжится, то некоторые существующие элементы могут быть обработаны повторно, другие пропущены, а новый элемент может встретиться позднее, а может и не встретиться. Вот уж действительно слабая семантика! Хотя в конкурентной системе такой вид несогласованности приемлем, в однопоточной системе столь непредсказуемое поведение не кажется разумным.

В погоне за предсказуемым поведением конкурентной модификации естественно задаться вопросом, нельзя ли подправить активные итераторы, когда коллекция подвергается изменению. Для `HashSet` это, возможно, и не получится, но для `List` вполне осуществимо. При добавлении или удалении элемента список мог бы обновить свои итераторы, перепозиционировав их в зависимости от того, произошла модификация до или после текущей позиции итератора. Конечно, при этом коллекция должна была бы следить за всеми своими итераторами. Это типичный подход к реализации *устойчивых итераторов*, описанный в книге «Паттерны проектирования» (Gamma et al. 1995, 261).

Приложения создают новые итераторы всякий раз, как требуется обойти коллекцию. Поэтому объекты итераторов накапливались бы бесконечно, если бы в интерфейсе `Iterator` не было какой-то операции «закрытия». Но такой API был бы неудобен клиентам, поскольку от них потребовалось бы заключать каждый обход в предложение `try-finally` или `try`-с-ресурсами. *Не* требовать от клиентов закрытия итераторов – правильный подход. Он значительно упрощает использование API и позволяет клиентам прервать обход в любой момент, например вернуть элемент сразу после того, как он найден.

Но и без операции «закрытия» все же можно организовать отслеживание итераторов коллекцией – нужно лишь воспользоваться классом `WeakReference` для обнаружения того факта, что клиент больше не пользуется итератором. `WeakReference` позволяет коду удерживать ссылку на объект до тех пор, пока на него существуют сильные ссылки. А если сильных ссылок не осталось, то `WeakReference` автоматически очищается (сбрасывается в `null`). Эта техника используется в конкурентной коллекции `ArrayBlockingQueue`. Но она существенно усложняет код, и представляется неразумным обременять каждую реализацию коллекции обязанностью следить за своими итераторами с помощью слабых ссылок.

На других платформах эта проблема решается по-разному. Например, в Smalltalk в классах коллекций вообще нет общей политики конкурентной модификации; такие действия могут приводить к непредсказуемым результатам. В этом отношении они похожи на оригинальные коллекции `Vector` и `Hashtable` в JDK 1.0. Ситуация, сложившаяся в Smalltalk, считается проблемой, и типичная идиома – скопировать коллекцию и обойти копию, разрешив модифицировать оригинал. В некоторых реализациях Smalltalk имеются расширения или инструменты, помогающие обнаружить конкурентную модификацию.

В библиотеке контейнеров C++ (*https://en.cppreference.com/w/cpp/container.html*) конкурентная модификация называется *инвалидацией итератора*. Операции коллекции могут никогда не инвалидировать ее итераторы, могут делать это иногда, а могут и всегда. Если итератор инвалидирован, то последующие операции с ним могут приводить к *неопределенному поведению*. Произойти может почти все что угодно: возврат неверных результатов, повреждение данных или аварийное завершение программы. Избегать использования инвалидированного итератора – обязанность программиста.

Вместо того чтобы пытаться обеспечить предсказуемое поведение, политика *быстрого отказа* в каркасе коллекций считает конкурентную модификацию программной ошибкой. Она старается обнаружить факт конкурентной модификации и возбуждает исключение `ConcurrentModificationException`, если это удается. Отметим, что такое обнаружение не гарантировано: существуют ситуации, когда конкурентная модификация имела место, но исключение *не* возбуждается. Таким образом, этот механизм полезен для обнаружения ошибок, но слепо полагаться на него во всех случаях нельзя.

Альтернативой прямому изменению коллекции является механизм ее изменения с помощью самого итератора. При этом одновременно обновляется и итератор, и коллекция, которую он обходит, так что они остаются согласованными. Но современная идиома обхода – цикл *foreach*, который скрывает итератор. Разумеется, классический цикл `for` по-прежнему доступен, но многие программисты предпочитают вообще избегать изменения на месте, копируя элементы в новую коллекцию по мере их обработки. Потоковый API поощряет использование этой идиомы.

Политика быстрого отказа довольно необычна для библиотеки. Исключение возбуждается не в ответ на нарушение предусловия, а вследствие обнаружения вероятной ошибки программирования. Кроме того, исключение возбуждается не кодом, допустившим программную ошибку, а «жертвой» этой ошибки. Если *этот* код обходит коллекцию, а другой код конкурентно модифицирует ее, то обнаруживает модификацию и возбуждает исключение *этот* код. Начинающих программистов такое поведение часто приводит в замешательство.

Но исключение `ConcurrentModificationException` может ставить трудные проблемы и профессиональным программистам. Оно возбуждается, когда срабатывает механизм обнаружения, а это результат «лучших из возможных усилий» (что на практике означает «минимальные усилия»). А раз так, то бывает, что исключение возбуждается не каждый раз, что может стать неприятным сюрпризом во время эксплуатации успешно протестированной программы.

Иногда возбуждать исключение неудобно. Альтернативу быстрому отказу можно было бы назвать *быстрым завершением*: вместо того чтобы возбуждать

исключение, код обхода организует как можно более быстрый выход из цикла и заносит в протокол сообщение. Это ни в коей мере нельзя назвать более корректным, чем возбуждение исключения; обход все равно может остаться незавершенным, а данные частично обновленными или вовсе не обработанными. Однако протоколирование сообщения может оказаться более подходящим способом уведомить разработчиков о том, что произошло нечто неправильное, без прерывания потока выполнения в результате исключения.

Главный вклад каркаса коллекции в эту область – разработка концепции *политики конкурентной модификации*. У разных реализаций коллекций разные политики в зависимости от ожидаемого использования. Это породило у Java-разработчиков всеобщее ожидание того, что конкурентная модификация обрабатывается в достаточной степени корректно. Очевидно, это лучше, чем отсутствие всякой политики или получения неопределенных результатов, когда бремя обнаружения и диагностики ошибок лежит всецело на программисте.

Политика быстрого отказа иногда приводит в замешательство, но у нее есть преимущество – простота и низкие накладные расходы. Возбуждение исключения временами вызывает неудобство, но оно уравновешивается выявлением ошибок, которые иначе могли бы остаться незамеченными. Проектировщики полагали, что это одна из самых ценных инноваций каркаса, и в общем и целом мы можем сказать, что время подтвердило это мнение.

Послесловие

Жизнь этой книги началась, когда Фил Уодлер вернулся в Шотландию из Америки в 2003 году, чтобы занять кафедру в Эдинбургском университете. В то время предложение Generic Java (Bracha et al. 1998), принятие которого дебатировалось в сообществе Java вот уже пять лет, наконец запланировали к включению в Java 5. Фил был одним из авторов Generic Java, а у меня имелся опыт функционального программирования, поэтому проект совместной книги по этому крупному ожидаемому обновлению напрашивался сам собой. Как оказалось, я не мог сравниться с Филом в глубине знаний по теории типов, поэтому он предложил в качестве дополнения к теоретическому обсуждению параметризованных типов включить самое важное практическое их приложение – каркас коллекций Java. Не могу сказать, что я специально занимался этой темой, поэтому мои первые потуги оказались, мягко говоря, не слишком удачными. Я в неоплатном долгу перед Джошем Блохом, который написал необычайно щедрую и подробную рецензию на последний черновой вариант рукописи, где были исправлены все мелкие ошибки, а сама ткань материала была переработана.

Возврат к книге спустя 20 лет стал интересным опытом. До начала работы я понимал, что может обнаружиться, что ни дженерики, ни коллекции не изменились настолько сильно, чтобы оправдать новое издание; в конце концов, язык Java знаменит своей обратной совместимостью, поэтому материал не должен быть ограничен временными рамками, верно? Но, конечно, погрузившись в детали я поразился своей наивности: не странно ли думать, что идиомы и стиль программирования могли застыть в неподвижности на двадцать с лишим лет? Но самым удивительным было то, как эти две вещи могут уживаться друг с другом, – Java остался поразительно стабильным, а стиль и идиомы написанных на нем программ сильно изменились. Я нахожу это особенно интересным, потому что мой профессиональный рост происходил в мире, где мы не раз становились свидетелями того, как какой-то стиль программирования выходит из моды, связанные с ним языки умирают или, по крайней мере, уходят со сцены. Но больше такое не происходит; языки 1990-х, включая Java, все еще популярны, хотя прошло 30 лет. А это ставит трудную задачу: как сделать так, чтобы язык, уходящий корнями в другой период времени, продолжал восприниматься как не чуждый современному окружению.

Быть может, самым важным единичным изменением стал гигантский рост быстродействия и мощности процессоров по сравнению с памятью. Это значит, что многопроцессорные системы с несколько уровнями кеша стали обыденностью, а эффекты задержки памяти и в особенности поведение кешей часто определяют производительность программы. Поэтому пространственная локальность стала главным условием высокой производительности, что подорвало предположение, изначально лежавшее в основе философии автомати-

ческого управления памятью в Java: что физическое расположение объектов в памяти не слишком важно. Продолжившееся развитие JVM и реализаций сборки мусора помогли защитить прикладных программистов от полномасштабных последствий, но все же каждое использование связанных структур данных следует подвергать тщательному анализу. В первом издании этой книги был раздел – не сохранивший актуальность, – в котором использование массивов в качестве структур данных осуждалось! Признаком продолжающейся адаптации Java можно считать одну из целей проекта Valhalla (*https://openjdk.org/projects/valhalla/*) – разрешить эту проблему, разгладив объекты и приведя их к виду, удобному для размещения в кеше.

Во втором издании соединена работа трех человек. Влияние Фила на часть I остается очень сильным. Но многие внесенные в нее поправки и многочисленные новые идеи в части II – результат долгих обсуждений со Стюартом Марксом, который формально является техническим редактором этого издания, но на самом деле полноценным соавтором. Его идеи, ободрение, поддержка и здравый смысл были неоценимы для реализации проекта. Из многих замечательных людей в сообществе Java, с которыми я имел счастье сотрудничать и у которых мне повезло учиться, не могу назвать никого, кто бы лучше подходил для руководства разработкой будущих версий каркаса коллекций Java. Я уверен, что и в следующие 20 лет он не растеряет своей притягательности!

— Морис Нафталин,
Эдинбург.
Февраль 2025

Литература

Bloch, Joshua. 2017. *Effective Java*. 3-е издание. Boston: Addison-Wesley.

Bracha, Gilad, Martin Odersky, David Stoutamire, Philip Wadler. 1998. «Making the Future Safe for the Past: Adding Genericity to the Java Programming Language». В *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 183–200. New York: ACM. *https://doi.org/10.1145/286936.286957*.

Calaprice, Alice, ed. 2011. *The Ultimate Quotable Einstein*. Princeton, NJ: Princeton University Press.

Farach-Colton, Martin, Andrew Krapivin, William Kuszmaul. 2025. «Optimal Bounds for Open Addressing Without Reordering». *arXiv*, *https://arxiv.org/abs/2501.02305v2*.

Fowler, Martin. 2003. «Anemic Domain Model». *https://martinfowler.com/bliki/AnemicDomainModel.html*.

Fuller, Thomas. 1732. *Gnomologia: Adagies and Proverbs; Wise Sentences and Witty Sayings, Ancient and Modern, Foreign and British*. London: Barker, Bettesworth and Hitch.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Gay, David, and Bjarne Steensgaard. 2000. «Fast Escape Analysis and Stack Allocation for Object-Based Programs». В *Compiler Construction. CC 2000*, edited by D. A. Watt. Lecture Notes in Computer Science 1781. Springer. *https://doi.org/10.1007/3-540-46423-9_6*.

Goetz, Brian, with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Boston: Addison-Wesley.

Goetz, Brian. 2020. «Background: How We Got the Generics We Have (Or, How I Learned to Stop Worrying and Love Erasure)». *https://openjdk.org/projects/valhalla/design-notes/in-defense-of-erasure*.

Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman. 2023. *The Java Language Specification, Java SE 21 Edition*. Oracle, Inc. *https://docs.oracle.com/javase/specs/jls/se21/html/index.html*.

Hoare, Tony. 2009. “Null References: The Billion Dollar Mistake.” *InfoQ*. *https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare*.

Igarashi, Atsushi, and Mirko Viroli. 2006. «Variant Parametric Types: A Flexible Subtyping Scheme for Generics». *ACM Transactions on Programming Languages and Systems*, 28 (5): 795–847. *https://doi.org/10.1145/1152649.1152650*.

Jones, C. B. 1983. «Tentative Steps Toward a Development Method for Interfering Programs». *ACM Transactions on Programming Languages and Systems*, 5 (4): 596–619. *https://doi.org/10.1145/69575.69577*.

Kabutz, Heinz. 2001. «Circular Array List». *The JavaSpecialists' Newsletter*, 27. *https://www.javaspecialists.eu/archive/Issue027-Circular-Array-List.html*.

Kabutz, Heinz. 2004. «References». *The JavaSpecialists' Newsletter*, 98. *https://www.javaspecialists.eu/archive/Issue098-References.html*.

Knuth, Donald E. 1974. «Structured Programming with *go to* Statements». *ACM Computing Surveys*, 6 (4): 261–301. *https://doi.org/10.1145/356635.356640*.

Knuth, Donald E. 1998. *The Art of Computer Programming*: *Sorting and Searching*, vol. 3, 2nd ed. Reading, MA: Addison-Wesley.

Lea, Doug. 1999. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Boston: Addison-Wesley.

Liskov, Barbara. 1987. Keynote Address—Data Abstraction and Hierarchy. В *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, 17–34. *https://dl.acm.org/doi/10.1145/62138.62141*.

Martin, Robert C. 2002. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall.

Naftalin, Maurice. n.d. «Maurice Naftalin's Lambda FAQ». *https://www.lambdafaq.org*.

Naftalin, Maurice. 2015. *Mastering Lambdas: Java Programming in a Multicore World*. New York: McGraw-Hill.

Newton, Isaac. 1675. «Letter to Robert Hooke». In *The Correspondence of Isaac Newton*, edited by H.W. Turnbull, vol. 1, 116–118. Cambridge: Cambridge University Press, 1959.

Odersky, Martin, Enno Runne, Philip Wadler. 2000. «Two Ways to Bake Your Pizza—Translating Parameterized Types into Java». Technical Report CIS-97-016, University of South Australia. *https://pizzacompiler.sourceforge.net/doc/pizza-translation.pdf*.

Sedgewick, Robert, Kevin Wayne. 2011. *Algorithms*. 4th ed. Boston: Addison-Wesley.

Simpson, George Gaylord. 1945. *The Principles of Classification and a Classification of Mammals*. New York: American Museum of Natural History.

Smith, Brian, Ross Cartwright. 2008. «Java Type Inference Is Broken: Can We Fix It?» В *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'08)*, 505–524. *https://doi.org/10.1145/1449764.1449804*.

Steele, Julie, and Noah Iliinsky, eds. 2010. *Beautiful Visualization: Looking at Data through the Eyes of Experts*. Sebastopol, CA: O'Reilly Media.

Torgersen, Mads, Christian Plesner Hansen, Erik Ernst, Peter von der Ahe, Gilad Bracha, and Neal Gafter. 2004. "Adding Wildcards to the Java Programming Language." В *Proceedings of the 2004 ACM Symposium on Applied Computing*, 1289–1296. *https://doi.org/10.1145/967900.968162*.

Valeev, Tagir. 2024. *100 Java Mistakes and How to Avoid Them*. Shelter Island, NY: Manning Publications.

Предметный указатель

Морис Нафтален, Филип Уодлер

Элегантный код Java: дженерики и коллекции

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Слинкин А. А.*
Корректор *Абросимова Л. А.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 27,85. Тираж 100 экз.

«Эта книга действительно обогатила мои знания о внутренних мотивах и реализации параметризованных типов и каркаса коллекций. Очень полезное чтение для тех, кто не просто потребляет, но еще и хочет эффективно применять эти мощные части языка Java».

Роберт Шольте,
основатель компании Sourcegrounds, Java Champion

Элегантный код Java

Это обновленное издание самого известного за последнее десятилетие руководства охватывает Java 21 и содержит актуальную и точную информацию о работе с параметризованными типами (дженериками) и коллекциями. Вы узнаете все необходимое для эффективного использования и написания параметризованных API. В книге подробно рассматривается библиотека коллекций, чтобы вы всегда знали, какую коллекцию применять для решения стоящей перед вами задачи. Проводится сравнение потоков и коллекций, рассмотрен выбор оптимальной модели. Это поможет получить максимум пользы от платформенной библиотеки.

Краткое содержание книги:

- основы параметризованных типов: типы-параметры и обобщенные методы;
- подтипизация и джокеры;
- параметризованные типы и рефлексия;
- рекомендации по использованию параметризованных типов;
- упорядоченные коллекции, появившиеся в Java 21;
- конкурентное программирование и потокобезопасность;
- лучшие практики использования и расширения каркаса коллекций Java.

Морис Нафтален — технический директор Morningside Light Ltd., консалтинговой компании по разработке ПО в Великобритании. Много лет использует и преподает Java, автор книги «Mastering Lambdas» (2015).

Филип Уодлер — профессор теоретической информатики в Эдинбургском университете, занимается исследованиями в области функционального и логического программирования.

Стюарт Маркс — руководитель проекта JDK Core Libraries в группе Java в компании Oracle. В настоящее время занимается сопровождением каркаса коллекций Java. Имеет степень магистра информатики, полученную в Стэнфордском университете.

ISBN 978-6-01140-648-2