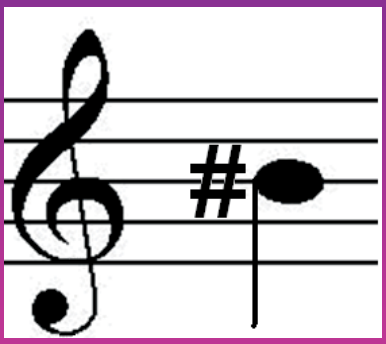
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина
Уральский энергетический институт

О. М. КОТОВ

ЯЗЫК C#: КРАТКОЕ ОПИСАНИЕ И ВВЕДЕНИЕ В ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Учебное пособие



Министерство образования и науки Российской Федерации
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

О. М. Котов

Язык C#: краткое описание и введение в технологии программирования

Рекомендовано методическим советом УрФУ
*в качестве **учебного пособия** для студентов, обучающихся*
по программе бакалавриата по направлению подготовки
140400 – Электроэнергетика и электротехника

Екатеринбург
Издательство Уральского университета
2014

УДК 004.43(075.8)
ББК 32.973-018.1я73
К73

Рецензенты:
кафедра автоматизированных систем электроснабжения РГППУ (протокол № 3 от 27.10.2011 г.) (завкафедрой доц., канд. техн. наук С. В. Федорова);
замдиректора филиала ОАО «НИИПТ» «Системы управления энергией» канд. техн. наук В. Г. Неуймин

Научный редактор – доц., канд. техн. наук П. А. Крючков

Котов, О. М.

К73 *Язык C#*: краткое описание и введение в технологии программирования : учебное пособие / О. М. Котов. – Екатеринбург : Изд-во Урал. ун-та, 2014. – 208 с.

ISBN 978-5-7996-1094-4

Настоящее учебное пособие содержит теоретический материал, многочисленные примеры, демонстрирующие практическое использование конструкций языка, а также варианты заданий для самоподготовки. Материал пособия может быть использован при изучении дисциплин «Информатика» (вторая часть) и «Современные языки программирования» студентами всех форм обучения направления 140400 «Электроэнергетика и электротехника».

Библиогр.: 4 назв. Табл. 3. Рис. 50. Прил. 2.

УДК 004.43(075.8)
ББК 32.973-018.1я73

Учебное издание

Котов Олег Михайлович
Язык C#: краткое описание и введение в технологии программирования

Подписано в печать 22.01.2014. Формат 60x90/16.
Бумага писчая. Плоская печать. Гарнитура Times New Roman.
Усл. печ. л. 13,0. Уч.-изд. л. 9,9. Тираж 200 экз. Заказ № 72.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620075, Екатеринбург, ул. Тургенева, 4
Тел. + (343) 350-56-64, 350-90-13
Факс + (343) 358-93-06
E-mail: press-urfu@mail.ru

ISBN 978-5-7996-1094-4

ВВЕДЕНИЕ

Язык программирования – это набор правил, с помощью которых программист записывает исходную программу. Далее из полученного текста специализированные программы (трансляторы, компоновщики и др.) практически без участия человека формируют код, предназначенный для процессора. По степени соответствия конструкций языка машинному (процессорному) коду языки программирования делятся на низкоуровневые (машинно ориентированные) и высокоуровневые. В свою очередь, языки высокого уровня делятся на структурные (процедурно ориентированные) и объектно ориентированные. В первом случае концепция программирования может быть определена как набор функций (центральный элемент системы), обрабатывающих данные (второстепенный элемент). В объектно ориентированных языках центральное место отведено данным, а выполнение функций так или иначе организовано вокруг этих данных. Типичным представителем процедурных языков считается язык Си, объектно ориентированных – Си++. Последний до недавних пор считался универсальным для решения широкого круга задач. При этом использование Си++ в конкретных инструментальных системах (например, Builder) и для конкретного вида приложений (например, сетевых приложений) требовало специализированных добавлений (надстроек) языка и в ряде случаев приводило к созданию своего рода диалектов, в которых базовые конструкции зачастую оказывались не всегда различимым фоном.

Когда суммарный вес специфических добавок превысил некоторый порог, появился новый язык программирования C# (произносится как *си шарп*, хотя разработчики предполагали название *си-диез*), вобравший в себя наиболее значимые черты своих предшественников и в большей степени отвечающий современным потребностям.

Руководителем группы разработчиков языка C# был Андерс Хейлсберг, сотрудник компании Microsoft (ранее входил в группу разработчиков инструментальной системы Builder). Цели, которые преследовались созданием этого языка, были сформулированы им следующим образом:

• создать объектно ориентированный язык, в котором любая сущность представляется объектом, объединяющим как информационную (данные), так и функциональную (действия над этими данными) части;

• создать первый компонентно ориентированный язык программирования семейства C/C++[*)];

• упростить C++, сохранив по возможности его мощь и основные конструкции;

• повысить надёжность программного кода.

Язык C# разработан прежде всего для платформы .NET (произносится как *дот-нет*), которая является средой, объединяющей программные технологии, для разработки Web- и Windows-приложений (отсюда и название).

Основу среды .NET составляет CLR (Common Language Runtime) – общеязыковая среда исполнения, которая состоит из двух основных частей:

- ядра (набор служб, управляющих загрузкой приложения в память, собраны в библиотеке *mscoree.dll*);
- библиотеки базовых классов (главная сборка в библиотеке *mscorlib.dll*).

В составе библиотеки базовых классов выделяется CTS (Common Type System) – общая система типов и подмножество этой системы CLS (Common Language Specification) – общеязыковая спецификация (содержит типы данных, которые гарантированно поддерживаются во всех языках .NET).

Процесс создания приложения на C# в среде .NET представляет собой последовательность этапов, изображённую на рис. 1.

Наряду с языком C#, в среде .NET в рамках указанной технологии могут быть использованы такие языки, как C++, VB.NET, Visual-Basic, Jscript, J# и другие (всего более 30 языков). С одной стороны, язык C# можно считать непосредственным преемником языков C и C++. С другой стороны, нельзя не заметить в его составе многочисленные признаки языка Java.

[*)] Компонентами называются объекты специальных типов, позволяющие прежде всего эффективно разрабатывать модульные оконные приложения. Конструкции и состав компонентов обычно определяется не только языком, но и платформой, на которой этот язык реализован.

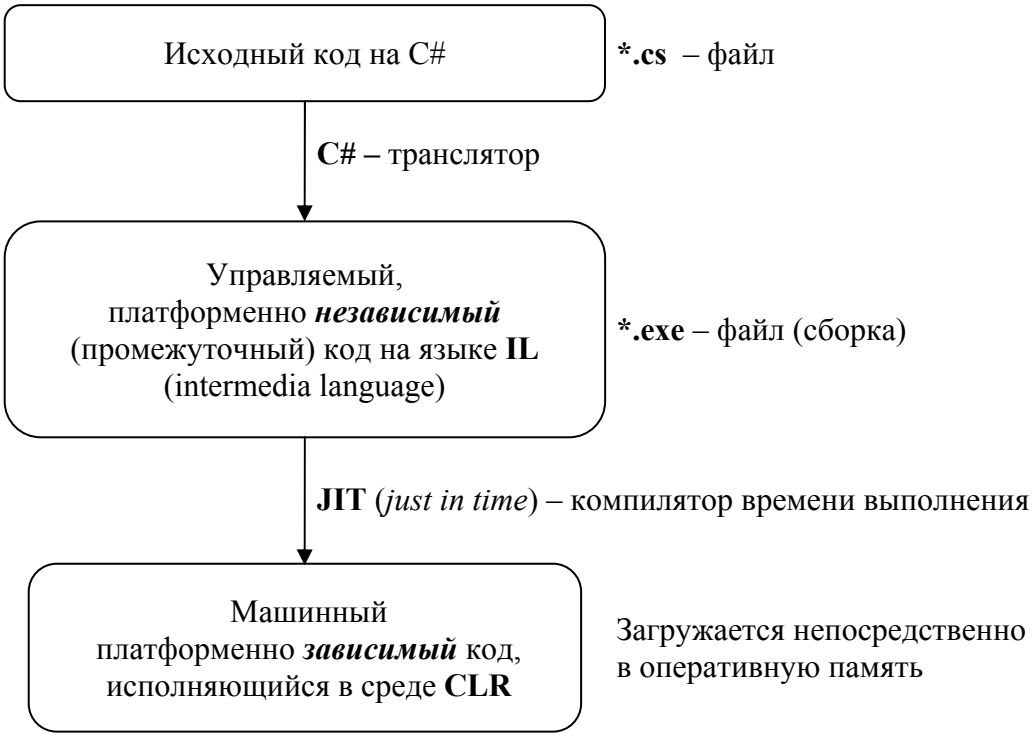


Рис. 1. Очередность этапов создания приложений

ПЕРВАЯ ПРОГРАММА НА C#

Текст первой (традиционной) программы предельно прост:

```
class FirstProgram
   {
      static void Main()
      {
      System.Console.WriteLine("Здравствуй, Мир!");
      }
   }
```

В консольном окне в режиме «Запуск без отладчика» (рис. 2) содержимое.

Рис. 2. Результат запуска первой программы

Слово `class`, расположенное в первой строке текста первой программы, относится к объектно ориентированной части языка, и разговор об этом будет отдельный. Для начала можно отметить, что `class` – это способ описания конструкции объекта. Элементами класса могут быть:

- поля – именованные области памяти для размещения данных;
- методы – фрагменты кода, выполняющие некоторое действие;
- свойства и операции – упрощенные (специализированные) вариации методов;
- события – специальные ссылки для так называемого *обратного* вызова методов. Механизм событий используется прежде всего для управления работой программы.

Слово `class` должно присутствовать в любой программе на C# хотя бы один раз. Здесь проявляется заимствование из языка Java: даже *начальная* программа `Main` должна принадлежать классу и поэтому является методом. В данном случае имя класса `FirstProgram`, а программа `Main` – часть этой конструкции. Имя после `class` произвольное.

Во второй строке слово `static` объявляет метод `Main` статическим, что определяет возможность запуска на выполнение без

предварительного создания объекта класса. Фраза `void Main( )` является заголовком метода `Main`: объявляет пустой возвращаемый тип (`void`) и пустой `( )` список входных параметров. Следующий далее блок фигурных скобок ограничивает *тело* метода `Main`, состоящее из одного оператора – вызова функции (правильнее сказать – *метода*) `WriteLine` с входным аргументом – строкой «Здравствуй, Мир!». Можно отметить, что аргументы методов всегда помещаются внутри круглых скобок. Оператор заканчивается символом «`;`», который является обязательным ограничителем большинства конструкций C#. Приписка перед именем функции `System.Console.` указывает, что функция `WriteLine` является методом класса `Console` (тоже статическим), который, в свою очередь, относится к категории системных (`System` – корневое пространство имён).

Имя главной программы `Main` не может быть изменено, так как система именно с этой программы начинает выполнение любой программной системы (так называемая *точка входа*).

Если использовать оператор `using`, определяющий использование пространства имён `System`, то оператор вызова функции `WriteLine` несколько упрощается:

```
using System;
class FirstProgram
    {
        static void Main()
        {
            Console.WriteLine("Здравствуй, Мир!");
        }
    }
```

Пространства имён

Как уже отмечалось выше, программирование на C# основано на системе (библиотеке) типов CTS. Эта библиотека насчитывает более 4000 различных типов, которые для удобства работы объединены в функциональные группы – пространства имён. Пространство имён может включать классы, структуры, интерфейсы, делегаты, перечисления. Пространства имён структурированы в иерархическую древовидную систему. Часть этой системы представлена на рис. 3.

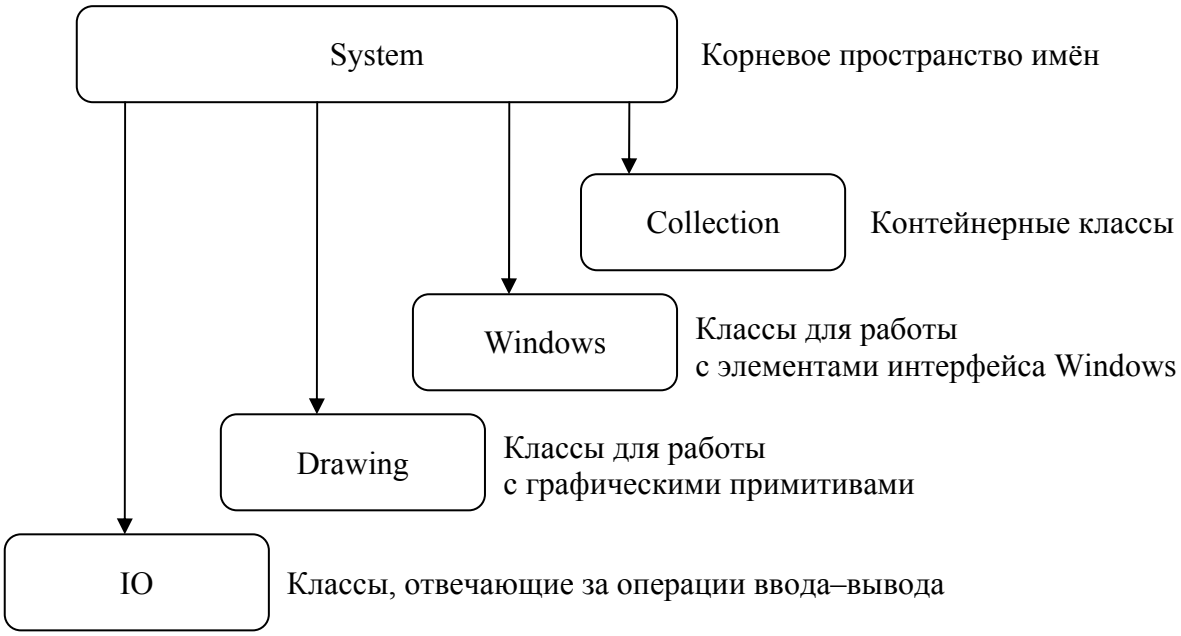


Рис. 3. Фрагмент пространства имён

ОБЩАЯ ХАРАКТЕРИСТИКА И ВСПОМОГАТЕЛЬНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА

Алфавит (множество литер) языка программирования C# составляют:

- строчные и прописные буквы латинского алфавита;
- цифры от 0 до 9;
- символ подчеркивания «_»;
- набор специальных символов: " ( ) { } | [ ] + - % / \ ; ' : ? < > = ! & # ~ *
- прочие символы.

Алфавит C# служит для построения слов, которые называются лексемами. Различают пять типов лексем:

- ключевые слова;
- разделители;
- идентификаторы;
- константы;
- знаки (символы) операций.

В составе языка C# 77 *ключевых слов*, из которых в первых главах пособия будет использована примерно половина (подчёркнуто):

abstract	false	params	this
as	finally	private	throw
base	fixed	protected	true
bool	float	public	try
break	for	readonly	typeof
byte	foreach	ref	uint
case	goto	return	ulong
catch	if	sbyte	unchecked
char	implicit	sealed	unsafe
checked	in	short	ushort
class	int	sizeof	using
const	interface	stackalloc	virtual
continue	internal	static	void
decimal	is	string	volatile
default	lock	struct	while
delegate	long	switch	
do	namespace		
double	new		
else	null		
enum	object		
event	operator		
explicit	out		
extern	override		

Лексемы языка имеют собственные правила словообразования (синтаксис) и обособляются *разделителями*:

- скобками ( ) { } [ ];
- пробелами;
- табуляцией;
- символом новой строки;
- комментариями.

О символах табуляции и новой строки речь пойдёт в разделе о литералах.

Комментарий – это текст, который предназначен только для читающего программу человека и компилятором игнорируется. В C# комментарий оформляется одним из трёх способов:

- при помощи парного комментария, произвольное количество строк /* … */;
- при помощи строчного комментария, который заканчивается в конце данной строки //…;
- xml – комментарий для создания самодокументирующихся программ (///…).

Идентификаторами называются имена, которые назначает программист. Именуются объекты, классы, структуры, методы, метки и тому подобное. Правила написания идентификаторов достаточно простые:

- идентификатор состоит из одного или более символов. Обычно даётся осмысленное, отражающее назначение элемента имя;
- размер идентификатора не ограничен;
- идентификатор не может совпадать с ключевым словом. Точнее сказать, может, но с точностью до `@` (например, `@else`, `@for`, `@case` – допустимые идентификаторы);
- первый символ – обязательно буква (либо `@`). Хорошую читабельность придают знаки нижнего подчёркивания внутри имени (`my_firstr_variable`) или использование прописных букв (`MyFirstrVariable`);
- в идентификаторах можно использовать русские буквы, например, `моя_первая_переменная` или `МояПерваяПеременная`.

ОБЪЕКТЫ ДАННЫХ И БАЗОВЫЕ СРЕДСТВА ИХ ОБРАБОТКИ

Иерархия встроенных типов в C#

Все объекты данных в C# являются производными от системного *супербазового* типа `Object` (или `object`) (рис. 4).

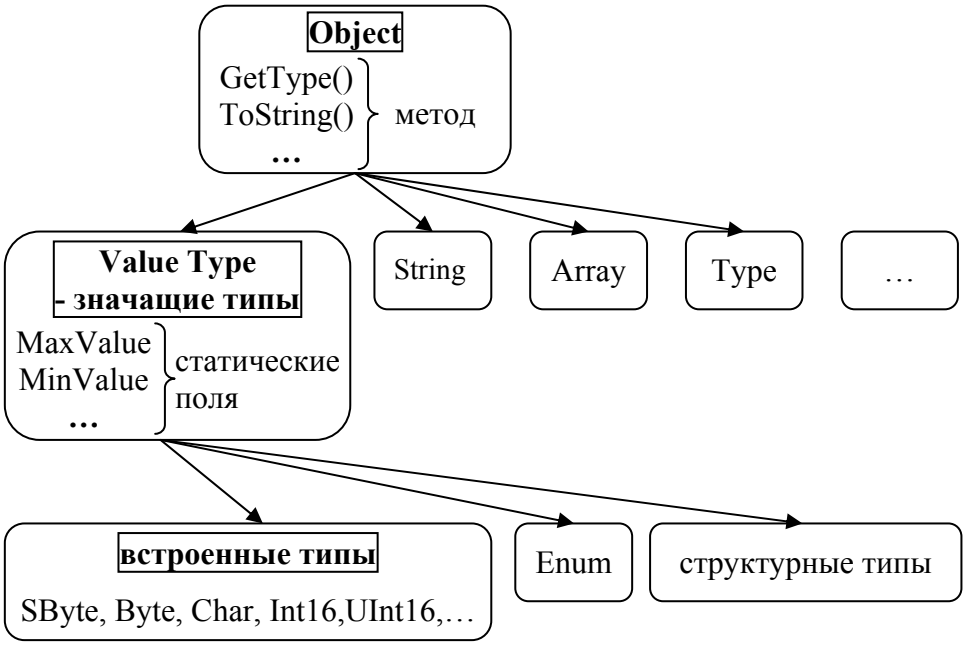


Рис. 4. Фрагмент иерархии типов

Методы `GetType()` и `ToString()` класса `Object` изначально возвращают строку, содержащую имя типа. Обычной практикой является *переопределение* метода `ToString()` под содержимое конкретного типа. Для встроенных типов это переопределение уже выполнено.

Любая программа .NET организуется в виде самодокументируемого файла – сборки (`assembly`). Сборка состоит из одного или нескольких модулей в виде IL-кода, а также необходимых описаний для использования этого кода. Такие описания называются *метаданными*. Для работы с подобными описаниями и разработан специальный тип `Type`. Он позволяет получить информацию о конструкторах, методах, полях, свойствах и событиях класса, а также о том, в каком модуле и сборке развернут соответствующий тип.

Константы

Константами называются объекты данных, которые не изменяют своего значения на всём времени выполнения программы. Константы в C# бывают трёх типов:

- литералы (или простые константы);
- символические константы;
- перечислимые константы.

Литералов насчитывается четыре типа: целочисленный, вещественный, символьный, строковый.

Целочисленный литерал (или целочисленная константа) служит для записи целых значений и является соответствующей последовательностью цифр. Этой последовательности может предшествовать знак '-', в данном случае являющийся операцией смены знака. Целочисленный литерал, начинающийся с `0x` или `0X`, воспринимается как шестнадцатеричное целое. В этом случае целочисленный литерал может включать символы от `A` (или `a`), до `F` (или `f`). В зависимости от значения целочисленный литерал размещается последовательно в одном из следующих форматов (по мере увеличения значения):

- целочисленный знаковый размером 4 байта;
- целочисленный беззнаковый размером 4 байта;
- целочисленный знаковый размером 8 байтов;
- целочисленный беззнаковый размером 8 байтов.

Непосредственно за константой могут располагаться в произвольном сочетании один или два специальных суффикса: `U` (или `u`) и `L` (или `l`). При этом суффикс `U` (или `u`) регламентирует использование беззнакового формата, а `L` (или `l`) – использование формата удвоенного размера (8 байтов).

Вещественный литерал служит для задания вещественных значений. Он представляет собой запись соответствующего значения в десятичной системе, в том числе в экспоненциальной форме, когда мантисса отделена от порядка символом `E` (или `e`). Размещается вещественный литерал в 8-байтовом плавающем формате (соответствует плавающему типу с удвоенной точностью `double`). Непосредственно за вещественным литералом может располагаться один из двух специальных суффиксов – `F` (или `f`) – для создания 4-байтовой константы плавающего типа (`float`).

Значением *символьного литерала* является соответствующий код. Синтаксически символьный литерал представляет собой последовательность одной или нескольких литер, заключенных в апострофы, и размещается в поле размером 2 байта. Вариантов написания несколько:

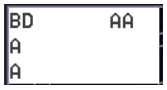
- символ, заключённый в апострофы, например `` `D` ``. Так рационально задавать символьные константы, для которых есть и клавиша на клавиатуре и символ (говорят: *печатный символ*);
- заключенная в апострофы целочисленная константа после символа «обратный слеш». Вариант оправдан для кодов, которые не представлены на клавиатуре. При этом сама константа должна быть представлена в шестнадцатеричной системе счисления. Например, константа `` `\x43` `` содержит код буквы D – 67;
- заключенная в апострофы литера после обратного слеша. Таким образом задаются `esc` – последовательности, представляющие собой команды управления некоторыми устройствами, прежде всего экраном:
  `` `\u` `` – задание четырёхзначного шестнадцатеричного кода в системе Unicode. Применимо к любому символу, но обязательно содержит четыре цифры;
  `` `\a` `` – звуковой сигнал (`\u0007`);
  `` `\b` `` – возврат на одну позицию назад (`\u0008`);
  `` `\n` `` – переход на новую строку (`\u000A`);
  `` `\r` `` – возврат каретки (курсора) в первую позицию строки (`\u000D`);
  `` `\t` `` – переход к следующей метке горизонтальной табуляции (`\u0009`);
  `` `\v` `` – переход к следующей метке вертикальной табуляции (`\u000B`);
  `` `\`` `` – апостроф;
  `` `\″` `` – кавычки.

Строковые литералы являются последовательностью (возможно, пустой) литер в одном из возможных форматов представления, заключенных в двойные кавычки. При этом сами литеры могут быть представлены либо символом, либо кодом:

```
using System;
class Primer0
{
 static void Main()
 {

Console.WriteLine("\t\u0041\x41\r\x42\x43\b\u0044");
  Console.WriteLine("A"+"\xA"+"A");
 }
}
```

Содержимое консольного окна после запуска приведённой программы имеет следующий вид:



Строка, переданная методу `Console.WriteLine()` во втором вызове, по сути дела является конкатенированной (объединённой). Строковый литерал, перед которым находится символ `@` (например, `@""`) называется *копирующим*, или *буквальным*, и обладает следующими свойствами:

- в его содержимом `esc` – последовательности – игнорируются;
- содержимое можно разбивать на строки, и это разбиение сохранится при выводе в консольное окно.

Следующий пример демонстрирует особенности разбивки на строки копирующих и строковых констант:

```
static void Main()
        {
            Console.WriteLine(@"
            строка1
            строка2
            строка3");
            Console.WriteLine(@"\tt\bb\nn\rr");
            Console.WriteLine("\tt\bb\nn\rr");
            Console.WriteLine("----------");}
```

Результат выполнения программы (консольное окно) приведён на рис. 5.

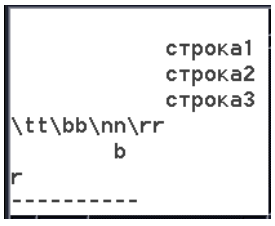


Рис. 5. Консольное окно

Примечание: обычную (некопирующую) константу разбивать на строки в тексте программы нельзя!

Рассматриваемые конструкции языка по ходу изложения материала будут и дальше иллюстрироваться на примерах с консольным выводом. Основными инструментами при этом будут статические методы класса `Console`:

`WriteLine()` – вывод строки на консоль с переводом курсора на новую строку (`Write()` – то же самое, но без перевода курсора);

`ReadLine()` – ввод строки с консоли;

`Read()` – чтение с консоли одного символа.

Метод `WriteLine()` можно использовать с одним аргументом, преобразование при этом выполняется автоматически:

```
using System;
class Primer
  {
   static void Main()
    {
Console.WriteLine(100);
Console.WriteLine(0x64);
Console.WriteLine('d');
    }
       }
```

Консольное окно будет таковым:

Кроме рассмотренных упрощенных вариантов метод `WriteLine()` может быть вызван вместе со средствами форматирования.

Пример:

```
using System;
class Primer1
    {
        static void Main()
        {
Console.WriteLine("Простые константы:");
Console.WriteLine("десятичная {0}",100);
Console.WriteLine("шестн. {0} символьная {1}",0x64,`d`);
   }
}
```

В таком случае содержимое полностью консольного окна следующее:

```
Простые константы:
десятичная 100
шестн. 100 символьная d
Для продолжения нажмите любую клавишу . . .
```

Отметим, что в данном примере методу `WriteLine` во втором вызове передаются уже два входных аргумента: строка с дополнительной конструкцией вида {`число`}, константа `100`.

Конструкции вида {`число`} называются метками форматирования, или подстановочными выражениями (*placeholder* – место заполнения). Число показывает номер элемента списка вывода (перечисление должно начинаться с нуля). Список вывода – это аргументы, за исключением первой строки. В предшествующем примере метки форматирования служат только одной цели – позиционировать выводимые константы в строке консоли. При использовании дополнительных параметров функциональность меток форматирования может быть значительно расширена. Дополнительные параметры располагаются после номера метки через двоеточие:

`D` или `d` – вывод целого десятичного значения;

`X` или `x` – вывод целого шестнадцатеричного значения;

`E` или `e` – вывод плавающего десятичного значения в экспоненциальной форме;

`F` или `f` – вывод плавающего десятичного значения в форме с фиксированной точностью;

`G` или `g` – вывод плавающего десятичного значения в универсальной форме;

`N` или `n` – вывод плавающего десятичного значения с разделением по три разряда.

Все параметры, за исключением последнего (`N` или `n`), могут дополняться целым числом – количеством позиций, которое для целых значений определяет размер всего поля, а для плавающих значений – количеством позиций для вывода дробной части (точность представления).

Ниже приведён пример вывода в консольное окно с форматированием:

```
using System;
class Primer2
{
```

```
 static void Main()
  {
Console.WriteLine("Форматирование с параметрами:");
Console.WriteLine("десятичное{0:d8}шестнадцатеричное {1:x8}", 0x64,100);
Console.WriteLine("стандартная запись={0:f3}",3.141592);
Console.Write("  экспоненциальная  запись  ={1:e4}",
3.141592);
  }
}
```

Консольное окно будет выглядеть так:

```
Форматирование с параметрами:
десятичное 00000100 шестнадцатеричное 00000064
стандартная запись =3,142 экспоненциальная запись=3,1416e+000
Для продолжения нажмите любую клавишу . . . _
```

Пример для самостоятельного выполнения. Установите, что появится на экране:

```
using System;
class Primer4
{
 static void Main()
 {
Console.WriteLine("  A={2:d}  B={1:x}  C={0:d}",  'M',
77, 0x77 );
 }
}
```

Ответ:

```
A=119 B=4d C=M
```

Для управления выводом вещественных значений на консольное окно с помощью `Console.WriteLine` можно воспользоваться шаблонами, которые определяют количество позиций и размещение разделителей:

```
class Primer3
{
 static void Main()
 {
Console.WriteLine("Шаблон ###.######## = {0:###.########}",
1234.56789);
Console.WriteLine("Шаблон ###.### = {0:###.###}",
1234.56789);
Console.WriteLine("Шаблон #`###.### ={0:#`###.###}",
1234.56789);
```

```
Console.WriteLine("Шаблон #`##.### = {0:#`##.###}",
1234.56789);
 }
}
```

Результат работы примера:

```
Шаблон ###.########  = 1234,56789
Шаблон ###.###       = 1234,568
Шаблон #`###.###     = 1`234,568
Шаблон #`##.###      = 12`34,568
```

Как следует из результата, если в шаблоне недостаёт позиций для размещения целой части числа, то он автоматически расширяется, если для дробной – дробная часть значения округляется до требуемой точности. На самом деле символические (литеральные) константы также являются объектами, производными от базового типа `Object`. Следующий пример демонстрирует это:

```
using System;
using C = System.Console;
class Константы
{
  static void Main()
    {
    C.WriteLine(0xabc.ToString());
    C.WriteLine(0xabc.GetType());
    }
}
```

В результате получим

```
2748
System.Int32
```

Для некоторого сокращения записи здесь с помощью конструкции `using` задан псевдоним класса `Console`.

Кроме задания констант в достаточно привычном виде, в языке C# имеется возможность задания и использования *символических*, или *именованных констант*. По сути дела это переменные так называемых встроенных типов, которые обязательно должны быть инициализированы при объявлении, впоследствии их значение не может быть

изменено. При объявлении символических констант на первом месте помещают модификатор `const`. Подробнее об этом речь пойдёт ниже.

Перечисления являются наборами символических констант. Особенности их объявления и использования будут рассмотрены после материала о символических константах.

Скалярные объекты встроенных типов данных

Язык C#, как и его ближайшие родственники, относится к языкам со строгой типизацией. Это означает, что любой объект до его использования должен быть объявлен. Исключение из этого правила составляют только простые константы.

Общий синтаксис оператора объявления объекта (обычно называемого *переменной*) в несколько облегченном виде следующий:

```
тип идентификатор = значение;
```

В качестве значения может выступать простая константа или уже объявленная переменная. При этом само значение в операторе объявления не является обязательным, но в любом случае переменная должна получить конкретное значение её *первого использования*. В следующем фрагменте во время трансляции будет получено сообщение об ошибке, связанной с попыткой использования неинициализированной переменной `b`:

```
int a=20, b;
Console.WriteLine(" a= {0} b= {0}", a, b);
```

Все объекты в языке C# объявляются в каком-либо блоке, в этом отношении они являются локальными. Их область видимости (`scope` – часть программы, в границах которой объект доступен для использования) начинается с оператора объявления и заканчивается фигурной скобкой, завершающей блок. Допускается неограниченное количество вложений одних блоков в другие. При этом каждый раз область видимости более внешнего объекта распространяется на более внутренний блок. Соответственно обратное не допускается. Следствием этого правила является запрет на совпадение имён переменных во внешних и внутренних блоках.

Типы в языке C#, как и в любом другом объектно-ориентированном языке программирования, делятся на встроенные (системные, интегрированные) и пользовательские. Первые определяются стандартом языка программирования. Вторые разрабатывает

сам программист. И те и другие типы в общем случае могут быть использованы для объявления двух видов объектов:

- размерных (или значащих). К объектам этого типа относятся скалярные переменные, структуры и перечисления. Отличительная особенность значащих объектов в том, что в них самих содержатся данные, а размещаются они в стековой памяти;
- ссылочных. Объекты такого типа всегда содержат множество элементов. При объявлении объекта ссылочного типа сначала создаётся *ссылка* – переменная, значением (содержимым) которой является адрес. Затем в динамической области выделяется память под размещение объекта, и *адрес* данного участка памяти сохраняется в *объекте-ссылке*. Это сильно напоминает косвенную адресацию, известную из ассемблера. Примером объекта ссылочного типа могут служить строки, массивы. Любой объект пользовательского типа, объявленного с помощью оператора `class`, также является ссылочным.

Из табл. 1 несложно заметить, что встроенные типы данных языка C# в основном соответствуют машинным форматам данных защищённого режима.

Таблица 1

Характеристика встроенных типов данных языка C#

Тип	Размер, байтов (состав и размеры полей, битов)	Соответствующий системный тип	Назначение
`bool`	1 (8)	`Boolean`	Логический `{true, false}`
`sbyte`	1 (1_7)	`Sbyte`	Знаковое целое в диапазоне [–128 … 127]
`byte`	1 (8)	`Byte`	Беззнаковое целое в диапазоне [0 … 255]
`short`	2 (1_15)	`Int16`	Знаковое целое в диапазоне [–32 768 … 32767]
`ushort`	2 (16)	`UInt16`	Беззнаковое целое в диапазоне [0 … 65 535]
`char`	2 (16)	`Char`	Код символа из таблицы Unicode (беззнаковый)
`int`	4 (1_31)	`Int32`	Знаковое целое в диапазоне [–2 · 147 483 648 … 2 · 147 483 647]

Окончание табл. 1

Тип	Размер, байтов (состав и размеры полей, битов)	Соответствующий системный тип	Назначение
uint	4 (32)	UInt32	Беззнаковое целое в диапазоне [0 … 4 294 967 295]
long	8(1_63)	Int64	Знаковое целое в диапазоне [–9 223 372 036 854 775 808 … … 9 223 372 036 854 775 807]
ulong	8 (64)	UInt64	Беззнаковое целое в диапазоне [0 … 18446744073709551615]
float	4 (1_8_23)	Single	Вещественное в диапазоне $[\pm 1{,}5 \cdot 10^{-45} \ldots \pm 3{,}4 \cdot 10^{38}]$, точность 6–7 знаков
double	8 (1_11_52)	Double	Вещественное в диапазоне $[\pm 5{,}0 \cdot 10^{-324} \ldots \pm 1{,}7 \cdot 10^{308}]$, точность 15–16 знаков
decimal	$16(1_{\_7}\_8_{\_16}\_96)$ (подстрочным шрифтом обозначены поля, заполненные нулями)	Decimal	Вещественное для финансовых расчётов в диапазоне $[\pm 1{,}0 \cdot 10^{-28} \ldots \pm 7{,}9 \cdot 10^{28}]$, точность 28–29 знаков

Типы, перечисленные в таблице, упорядочены по старшинству. Старшинство типа определяется размером поля, в котором размещается значение объекта: чем больше поле, тем старше тип. Старшинство типа имеет значение при операциях и преобразованиях между числовыми типами. *Типом по умолчанию* считается тип `int`.

Изучение объектов целесообразно начать с более простых *размерных*, или скалярных, объектов.

Размер типа в байтах может быть получен с помощью операции `sizeof`. В следующем примере показаны способы объявления и инициализации объектов встроенных типов, а также использование операции `sizeof`:

```
using System;
class Primer5
{
 static void Main()
 {
byte  a=255;
```

```
//неявное преобразование - выполняется только для
    //целочисленных типов
sbyte b=127;
char  c= 'd';
bool  d;
short e;
ushort f;
int g= 0x100;
float h= 5.2F;
double i= 123e-2;
decimal j=0.00000000001M;
d=false;
e=32767;
f=65535;
Console.WriteLine("{0}={1}\t\tразмер={2}",
a.GetType(), ++a, sizeof(byte));
Console.WriteLine("{0}={1}\tразмер={2}", b.GetType(),
++b, sizeof(sbyte));
Console.WriteLine("{0}={1}\t\tразмер={2}",
c.GetType(), ++c, sizeof(char));
Console.WriteLine("{0}={1}\tразмер={2}", d.GetType(),
d, sizeof(bool));
Console.WriteLine("{0}={1}\tразмер={2}", e.GetType(),
++e, sizeof(short));
Console.WriteLine("{0}={1}\t\tразмер={2}",
f.GetType(), ++f, sizeof(ushort));
Console.WriteLine("{0}={1}\tразмер={2}", g.GetType(),
++g, sizeof(int));
Console.WriteLine("{0}={1}\tразмер={2}", h.GetType(),
++h, sizeof(float));
Console.WriteLine("{0}={1}\tразмер={2}", i.GetType(),
++i, sizeof(double));
Console.WriteLine("{0}={1}\tразмер={2}", j.GetType(),
++j, sizeof(decimal));
 }
}
```

В итоге работы будет получен следующий вид на экране (рис. 6).

```
System.Byte=0           размер=1
System.SByte=-128       размер=1
System.Char=e           размер=2
System.Boolean=False    размер=1
System.Int16=-32768     размер=2
System.UInt16=0         размер=2
System.Int32=257        размер=4
System.Single=6,2       размер=4
System.Double=2,23      размер=8
System.Decimal=1,00000000001    размер=16
Для продолжения нажмите любую клавишу . . .
```

Рис. 6. Результат работы примера

В данном примере был использован нестатический метод с именем `GetType()`, который имеется у любого встроенного типа (а значит, и у любого объекта встроенного типа). Данный метод возвращает системное название типа. После соответствующего названия типа на экран выводится значение объекта, увеличенное на 1, (за исключением булевого типа `d`) и размер типа в байтах.

Следующий пример еще раз доказывает, что встроенные типы являются значащими:

```
using System;
using C = System.Console;
class Пример 5_1
{
 static void Main()
    {
        int a1=10;
        int a2 = a1;
        a2 = 20;

        int b1 = new int();
        int b2 = b1;
        b2 = 40;
        C.WriteLine("a1={0}, a2={1}", a1, a2);
        C.WriteLine("b1={0}, b2={1}", b1, b2);
    }
}
```

Результат примера будет

```
a1=10, a2=20
b1=0, b2=40
```

Использованная для инициализации переменной `b1` операция `new` выделяет память, как правило, для ссылочных объектов, но может быть использована и для значащих, как в данном случае. Аргументом операции при размещении скалярных объектов является тип и, если необходимо, инициализирующее значение в скобках. При этом память для значащих объектов выделяется в стеке, а для ссылочных – в куче (`heap`).

Tun float

Тип `float` соответствует системному плавающему типу одинарной точности. Структура типа `float` такова:

31	*30* … *23*	*22* … *0*
Знак	Смещённый порядок(8бит)	Мантисса(23бит)

Поставим цель проверить значение наибольшего числа из диапазона для типа `float`. Поскольку наибольшее значение смещённого порядка на единицу меньше максимально возможного и составляет 0xFE, а наибольшее значение нормализованной мантиссы (0xFFFFFE) размещается без единицы в целой части, постольку искомое число во внутреннем представлении имеет вид 0x7F7FFFFF. Следовательно, знак равен 0, $p_{смещённое} = 254$, $M_{нормализованная} = 0x1.FFFFFE$, $p_{реальное} = 127$, результат составляет $0x1.FFFFFE \cdot 2^{127} = 0x1.FFFFFE \cdot 2^{128} \cdot 2^{-1} = 0x1.FFFFFE \cdot (2^4)^{32} \cdot 2^{-1} = 0x1.FFFFFE \cdot 16^{32} \cdot 2^{-1} = 0x1FFFFFE \cdot 16^{26} \cdot 2^{-1}$.

Следующий фрагмент на C# позволяет собрать полученное число:

```
class Program
    {
        static void Main()
        {
          float d, e= 0x1fffffe;
          d = (float)Math.Pow(16, 26);
         Console.WriteLine("Результат = {0:g}",d*e/2);
        }
    }
```

В результате на экране появится надпись:

Tun decimal

Для представления вещественного числа повышенной точности в язык C# добавлен тип `decimal`. Его размер 16 байтов, или 128 битов. Аппаратные возможности для этого типа обеспечивает блок дополнительных XMM-регистров, которые интегрируют в процессоры начиная с шестого поколения (Pentium III).

Структура формата `decimal` следующая:

127	*126 120*	*119 112*	*111 96*	*95 0*
S	*нули (7бит)*	Множитель (8 битов)	*Нули (16 битов)*	Целое число (96 битов)

В ней можно выделить:

- S – знаковый бит;
- множитель, или масштабирующий множитель (ММ) – это целое число, которое может принимать значения от 0 до 28. Математически ММ представляет собой отрицательную степень в экспоненциальном представлении вещественного числа при его

нормализации к нулевой дробной части (по сути – количество цифр в дробной части).

Пример:

1234.456 = 1 234 456 · 10^{-3} = 12d618***h*** · 10^{-3} (MM = 3).

Внутреннее представление числа:

00 03 00 00 00 00 00 00 00 00 00 00 12 d6 18 ***h***

В следующем примере для получения составляющих объекта `D` типа `decimal` используется статический метод `GetBits` класса `System.Decimal`, интерфейс которого – `public static int[] GetBits (decimal d)`. Возвращаемое значение метода – массив 32-разрядных целых чисел со знаком, состоящий из четырех элементов типа `Long`:

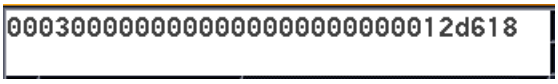
- первый, второй и третий элементы возвращаемого массива содержат соответственно младшие, средние и старшие разряды 96-разрядного целого числа (по 32 разряда в каждом параметре);
- четвертый элемент возвращаемого массива содержит масштабирующий множитель и знак. Он состоит из следующих частей:
  - разряды с 0 по 15 (младшее слово) не используются и должны равняться нулю;
  - разряды с 16 по 23 должны содержать показатель степени от 0 до 28, в которую нужно возводить число 10 для преобразования искомого числа в целое;
  - разряды с 24 по 30 не используются и должны равняться нулю;
  - разряд 31 содержит знак: 0 соответствует знаку плюс, 1 – знаку минус.

Пример:

```
using System;
class Primer
    {
        static void Main()
        {
            decimal D = 1234.456m;
            int[] form = Decimal.GetBits(D);
Console.Write("{0:x8}",form[3]);
Console.Write("{0:x8}", form[2]);
Console.Write("{0:x8}", form[1]);
```

```
Console.Write("{0:x8}", form[0]);
Console.ReadLine();
        }
}
```

Результат его выполнения



Выражения и операции

Основной объём обработки данных в программах выполняется с помощью имеющегося в составе языка набора операций. Лексическая конструкция для выполнения подобных действий называется *выражением*. Операции делятся по назначению. На с. 27 приведен полный список операций языка C#.

В общем случае выражение – это объединение операндов (объекты данных) с помощью операций. Синтаксис выражений максимально приближен к алгебраической форме:

```
операнд1 @ операнд2 @ операнд3,
```

где `@` – знак операции.

Операции разделяются по *арности* (количество операндов):

- унарные (или одноместные – один операнд);
- бинарные (двуместные – два операнда);
- тернарные (трехместные – три операнда). Тернарная операция единственная, по-другому называется *условной*, или операцией *проверки*.

Операции в C#, как и алгебраические, упорядочены по приоритету. В табл. 2 операции таких групп, как арифметические, битовые, логические, отношения, сдвига и замещения, расположены в порядке уменьшения приоритета. Операции одинакового приоритета выполняются в очередности слева направо, кроме операций присваивания, которые выполняются справа налево. Изменить очередность операции в выражении можно, как и в алгебре, с помощью круглых скобок.

Операции языка C#:

Арифметические ..... `+ - * / % ++ --`

Логические (булевые) ..... `! ~ && || true false`

Битовые ..... `& | ^`

Слияния строки ..... `+`

Сдвиговые ..... << >>
Отношения ..... == != < > <= >=
Замещения..... = += -= *= /= %= &= |= ^= <<= >>= ??
Доступа к элементу ..... .
Индексации ..... []
Приведения типа..... ()
Выбор по условию..... ?:
Конкатенация и удаление делегата ..... + -
Создания объекта..... new
Типа информации ..... as is sizeof typeof
Управление исключениями по переполнению.....checked unchecked
Адресации и разадресации .....* -> [] &

Таблица 2

Характеристика операция языка C#

Знак операции	Приоритет	Арность	Действие	Примечание
-, +	0	1	Смена знака (унарные минус, плюс)	–
!	0	1	Логическая НЕ	Для булевого операнда
~	0	1	Инверсия	Для целочисленных операндов
++	0	1	Инкремент	Префиксная операция выполняется до использования переменной, постфиксная – после
--	0	1	Декремент	
*	1	2	Умножение	–
/	1	2	Деление	
%	1	2	Остаток от деления	Первый операнд может быть вещественным
+, -	2	2	Сложение, вычитание	–
<<	3	2	Сдвиг влево	Для целочисленных операндов
>>	3	2	Сдвиг вправо	–
<	4	2	Отношение меньше	Результат – значение булевого типа (`true` или `false`)
>	4	2	Отношение больше	
<=	4	2	Отношение меньше или равно	
>=	4	2	Отношение больше или равно	
==	5	2	Отношение равенства	

Окончание табл. 2

Знак операции	Приоритет	Арность	Действие	Примечание
`!=`	5	2	Отношение неравенства	Результат – значение булевого типа (`true` или `false`)
`&`	6	2	Битовая И	Для целочисленных операндов
`^`	7	2	Битовая исключающая ИЛИ	
`|`	8	2	Битовая ИЛИ	
`&&`	9	2	Логическая И	Для булевого операнда
`||`	10	2	Логическая ИЛИ	Для булевого операнда
`??`	11	2	Логического замещения	Первый операнд проверяется на `null`, и если не равен, его значение возвращается, в противном случае возвращается значение второго операнда
`?:`	12	3	Проверка или выбор по условию	
`=`	13	2	Присвоение	
`*=`		2	Умножение с замещением	–
`/=`		2	Деление с замещением	
`+=`		2	Сложение с замещением	
`-=`		2	Вычитание с замещением	
`<<=`		2	Сдвиг влево с замещением	
`>>=`		2	Сдвиг вправо с замещением	
`&=`		2	Битовая И с замещением	
`^=`		2	Битовая исключающая ИЛИ с замещением	
`|=`		2	Битовая ИЛИ с замещением	

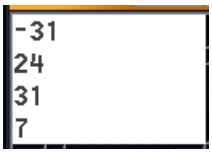
Битовые операции

Битовые операции могут быть применены только для объектов целочисленных типов. По своему смыслу они соответствуют одноимённым процессорным командам (AND, OR, XOR, NOT).

Пример выполнения битовых операций:

```
using System;
class Primer
{
    static void Main()
    {
        byte a = 25;
        sbyte b = 30;
        Console.WriteLine(~b);
        Console.WriteLine(a & b);
        Console.WriteLine(a | b);
        Console.WriteLine(a ^ b);
    }
}
```

Результат выглядит так:



В качестве самостоятельного упражнения выполните в ручном режиме указанные битовые преобразования и проверьте представленные в результате значения.

Операции сдвига

Как и рассмотренные в предыдущем разделе битовые операции, операции сдвига могут быть выполнены только с операндами целочисленных типов. Сдвиги нециклические: при сдвиге влево единица из старшего разряда уходит во флаг переноса, но при сдвиге вправо значение флага переноса не используется и младший разряд всегда устанавливается в нуль. При этом если выполняется сдвиг вправо переменной со знаком, то знаковый разряд каждый раз восстанавливается, а при сдвиге переменной со знаком влево знаковый разряд может быть деформирован. Следующий пример демонстрирует использование сдвиговых операций.

```
using System;
class Primer
{
    static void Main()
    {
        byte b1 = 128;
```

```
            sbyte s1 = 64, s2 = -128 ;
            b1 <<= 1;
            s1 <<= 1;
            s2 >>= 1;
            Console.WriteLine(" b1 << 1 = {0}", b1 );
            Console.WriteLine(" s1 << 1 = {0}", s1 );
            Console.WriteLine(" s2 >> 1 = {0}", s2 );
            s2 <<= 2;
            Console.WriteLine(" s2 << 2 = {0}", s2);
        }
    }
```

Результат выполнения операция следующий:

```
b1 << 1 = 0
s1 << 1 = -128
s2 >> 1 = -64
s2 << 2 = 0
```

Битовые и сдвиговые операции можно использовать, например, для перевода десятичного значения в двоичное:

```
using System;
class Primer
{
    static void Main()
    {
        ushort m = 32768, val = 6464;
        while (m > 0)
        {
            if ( (m & val) == 0) Console.Write(0);
            else Console.Write(1);
            m >>= 1;
        }
    }
}
```

В итоге получим

```
0001100101000000
```

Арифметические операции

По синтаксису арифметические операции наиболее близки привычным алгебраическим выражениям. Так же как и в арифметике, умножение и деление старше, чем сложение и вычитание. Некоторые особенности использования арифметических операций демонстрирует следующий пример.

```
  using System;
  class Primer7
{static void Main()
 {short e = 25;
  int g = 10;
  float h= 10F;
  double i= 25e-1;
  Console.WriteLine(e/g);
  Console.WriteLine(e/h);
  Console.WriteLine(i*h);
  }
}
```

Отличие результата во второй строке по сравнению с результатом в первой строке объясняется работой механизма *автоприведения типа* в арифметических операциях.

Преобразование типов

Зачастую данные в процессе обработки необходимо преобразовывать из одного формата представления в другой, а иногда, как это продемонстрировано в предшествующем примере, это происходит в автоматическом режиме. Существует два режима приведения типов:

- принудительное, или явное, приведение (`explicit`);
- автоматическое, или неявное, приведение (`implicit`).

Механизм приведения типов применяется в следующих ситуациях:

- автоприведение типа в операции присваивания (как правило, *расширяющее* преобразование);
- принудительное преобразование типа с помощью операции (`тип`);
- автоприведение типа (как правило, *расширяющее* преобразование) в двуместной арифметической операции.

По своей идее *автоприведение типа в операции присваивания* сводится к копированию данных по новому месту расположения, которое определяется типом операнда-приёмника. По этой причине результат выражения справа от операции «=» (или просто тип операнда-источника) преобразуется в тип операнда-приёмника. Для исключения при этом потери данных автоприведение типа в операции

присвоения выполняется только в сторону повышения старшинства типа (расширяющее преобразование).

В случае когда необходимо выполнить приведение в сторону уменьшения старшинства типа (*сужающее* преобразование), до операции присваивания следует выполнить унарную операцию приведения типа (`тип`), в которой `тип` определяет формат данных, к которому будет преобразовано значение операнда:

```
ОперандПриёмник = (тип) ОперандИсточник;
```

Здесь ответственность за возможную потерю данных лежит на программисте.

Можно отметить, что имеются все возможные варианты преобразования между встроенными *числовыми* типами. При этом не существует ни явных, ни неявных преобразований из типа `bool` в любой иной тип и обратно.

Автоприведение типа в двуместной арифметической операции выполняется неявно и в сторону повышения типа. Дело в том, что двуместные арифметические операции процессор выполняет всегда с операндами одинакового формата. Но в арифметических выражениях на любом из высокоуровневых языков программирования допускается совмещать в одной операции операнды различных размеров в расчёте на то, что перед непосредственным выполнением данные из операнда *младшего* типа будут преобразованы к типу *старшего* операнда. Так в предыдущем примере деление `e/h` выполняется в типе `float`, к которому предварительно приведено значение переменной `e` типа `short`.

Приведение типа в операции присваивания и принудительное приведение типа демонстрирует следующий пример:

```
using System;
class Primer8
{
    static void Main()
    {
        byte e;
        short g = 260;
        float h;
        h = g;
        e = (byte)g;
        Console.WriteLine(g);
        Console.WriteLine(h);
```

```
            Console.WriteLine(e);
            g = -10;
            e = (byte)g;
            Console.WriteLine(e);
        }
    }
```

```
260
260
4
246
```

Следует обратить внимание на результат в последней строке: потеря данных при усечении типа может приводить и к таким неожиданным эффектам!

Несколько другим образом выполняется приведение в целочисленных выражениях.

1. Если в них имеются типы младше `int`, они будут автоматически приведены к этому типу (именно `int` является базовым типом для целочисленных данных). Такой приём называется *целочисленным продвижением типа* (`integer promotion`).

Так во фрагменте

```
byte a, b = 10, c = 20;
a = b + c ;
```

из-за невозможности неявного преобразования из типа `int` в тип `byte` во второй строке генерируется ошибка. Действительно, преобразование в процессе присвоения выполняется только в сторону повышения типа, но возникает вопрос: откуда справа от операции присваивания появился тип `int`. Ответом является работа механизма `integer promotion`. Правильный вариант может быть, например, следующим:

```
byte a, b = 10, c = 20;
a = (byte)(b + c);
```

Существенно, что и тип `char` также попадает под действия механизма *целочисленного продвижения* типа.

2. Если в выражении смешиваются знаковые типы не старше `int` (фактически продвинутые до `int`) и беззнаковый `uint`, то все операнды будут расширены до следующего по иерархии знакового типа, т. е. до `long`:

```
using System;
class Primer8_1
{
    static void Main()
    {
        int i = -30;
        uint ui = 20;
        Console.WriteLine(ui+i);

    }
}
```

Если бы данного правила не существовало, данный результат был бы невозможен:

Поэтому нельзя смешивать `ulong` с каким-либо целочисленным знаковым типом – повышать тип в данном случае уже некуда. Не допускается также смешивать в одном выражении тип `decimal` с другими плавающими типами (т. е. не существует неявного приведения). При этом с целочисленными тип `decimal` смешивать можно!

Кроме этого, не существует неявных приведений типов (даже расширяющих) из типа `char` и в него.

Особый порядок выполнения операций замещения

Операции замещения представляют собой сокращенную запись выражения из двух операций, в каждой из которых участвует `operand1`:

operand1 @= operand2 эквивалентно
operand1 = operand1 @ operand2

Представляет интерес реализация выражения, в которое операция замещения входит с другими операциями:

```
operand1 @1= operand2 @2 operand3;
```

В этом случае эквивалентная запись:

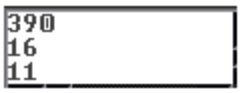
```
operand1 = operand1 @1 (operand2 @2 operand3)
```

Таким образом, из-за присутствия в эквивалентной записи круглых скобок операция замещения вне зависимости от старшинства остальных операций выражения выполнится последней.

Пример на операции замещения:

```
using System;
class Primer9
{
    static void Main()
        {
         short e = 10;
         int g = 1;
         float h = 27f;
         e *= 25 + 14 ;
         g <<= 4 + e / (e + 1);
         h %= g;
         Console.WriteLine(e);
         Console.WriteLine(g);
         Console.WriteLine(h);
        }
}
```

Результат следующий:



Логические операции и операции отношения

Объединяет данные группы операций их результат – значение булевого типа: `true` или `false`. Операндами логических операций могут быть только булевы объекты, а операций отношения – любые объекты. В следующем примере демонстрируется использование указанных операций:

```
using System;
class Primer10
{      static void Main()
         {
         bool  e,g ;
         Console.WriteLine( 10>>2 <= 5);
         Console.WriteLine( 10%3 > 10%2 || 10%4 > 10%5 );
         e = 10 >= 10/3*3;
         g = !e;
         Console.WriteLine(e);
         Console.WriteLine(g);
         Console.WriteLine(e && g);
         }
}
```

Основным местом использования операций отношения и логических операций являются операторы управления программой.

Логические операции (`оп1 && оп2` и `оп1 || оп2`) иногда учебниках называют сокращёнными. Дело в том, что второй операнд (а это в общем случае может быть и выражение) не вычисляется вовсе в случаях:

- если в операции `оп1 && оп2` значение первого операнда `false`;
- в операции `оп1 || оп2` значение первого операнда `true`.

В следующем фрагменте ни при каких обстоятельствах деление на нуль не произойдёт:

```
int d=0, n=1;
if(  d != 0 && (n / d) == 1)Console.WriteLine(d);
if(  d == 0 || (n / d) == 1) Console.WriteLine(d);
```

Операции инкремента и декремента

Момент выполнения данных операций зависит от формы их реализации:

`@operand` – префиксная, значение объекта изменяется *до* его использования;

`operand@` – постфиксная, значение объекта изменяется *после* его использования.

Операнд может быть использован или в выражении, или в качестве аргумента метода, или в каком-либо ином качестве, например:

```
using System;
using C=System.Console;
class Primer11
{
        static void Main()
        {
            int i = 1, j = 2;
            C.WriteLine(i++);
            C.WriteLine(++i);
            C.WriteLine(++i);
```

```
            j = ++j * (i++ - 1);
            C.WriteLine(i);
            C.WriteLine(j);
        }
    }
```

Результат выполнения:

Объявление и использование символических констант

Символические (именованные) константы предназначены для размещения значений, которые остаются неизменными на всём времени выполнения программы, но при этом имеют имена как объекты (переменные) встроенных типов. Использование именованных констант даёт программисту следующие преимущества:

- возможность ассоциации имени с содержимым константы;
- возможность оперативного изменения значения константы (достаточно изменить инициализирующую константу в операторе объявления вместо того, чтобы искать по тексту программы операторы, в которых использовалось изменяемое число).

Объявление символических констант имеет следующие особенности:

- оператор объявления начинается модификатором `const`;
- обязательной является начальная инициализация.

В результате синтаксис объявления именованной константы имеет вид:

```
const тип имя = значение;
```

При этом `тип` может быть опущен, и тогда для объявления переменной будет использован тип по умолчанию – `int`.

Объявление и использование перечислимых констант

Перечислимые константы, или перечисления, – это, как уже указывалось выше, объект, полученный в результате объединения некоторого количества символических констант. Он является наиболее простым по конструкции, но уже агрегированным объектом. В сле-

дующем примере для перечисления используется тип по умолчанию int:

```
using System;
class Primer11_1
{
enum Друзья { Александр, Игорь, Константин, Ярослав};
enum Клавиши { F1=59,F2, F5=63, F6, F9=67};
static void Main()
  {
Console.WriteLine("{0}={1:d}",Друзья.Константин,
Друзья.Константин);
int i =(int)Друзья.Игорь+(int)Друзья.Ярослав+(int)Клавиши.F6;
Console.WriteLine(i);
  }
}
```

```
Константин = 2
68
```

В примере следует обратить внимание на конструкцию, обеспечивающую доступ к символической константе:

ИмяПеречисления.ИмяЭлемента.

Также в примере показано, что перечисление является не только агрегированным, но и пользовательским типом. Действительно, только программист задаёт и тип констант в перечислении, и их количество, и начальные значения. Объявлять перечисление следует вне тела какого-либо метода. Кроме того, любое перечисление представляет собой объект, производный от типа System.Enum. Здесь же это проявляется в том, что перечисление не тождественно типу составляющих его констант, и поэтому использование значений констант требует явного приведения типов. Отметим также, что значением (и базовым типом) перечислений могут быть только целочисленные типы, начальное значение *по умолчанию* – ноль.

Операция выбора

Операция выбора (единственная тернарная операция) записывается в следующем виде:

operand1 ? operand2 : operand3;

Непосредственное использование операции выбора демонстрирует следующий пример.

```
using System;
class Primer12
{
        static void Main()
        {
            int  j ;
            bool b = true || false && true;
            j == b ? 10 : -10;
            Console.WriteLine(j);
        }
}
```

```
10
```

Чаще операцию выбора используют, когда `operand1` является собой логическим выражением.

На следующем примере покажем использование операции выбора для определения максимального из двух введенных чисел.

```
using System;
class Primer13
{
  static void Main()
  {
   int  i, j, k;
   string s;
   Console.WriteLine("Задайте первое число!");
   s = Console.ReadLine();
   i = Convert.ToInt32(s);
   Console.WriteLine("Задайте второе число!");
   s = Console.ReadLine();
   j = Convert.ToInt32(s);
   k = i > j ? i : j;
  Console.WriteLine("Максимум из заданных чисел = {0}",k);
  }
}
```

```
Задайте первое число!
1
Задайте второе число!
20
Максимум из заданных чисел = 20
```

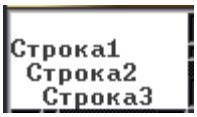
Для ввода значения с клавиатуры в данном примере был использован метод `ReadLine()` того же класса, что и `Console`. Такой метод возвращает значение типа `string`, и для размещения этого значения объявлен специальный объект `s` (типа `string`). Для проеобразования введенных с клавиатуры кодов в числовое значение

используется метод `ToInt32()` класса `Convert`. Аргументом метода служит строка `s`.

Тип `string` – это встроенный системный тип для работы с текстовой информацией. Объект этого типа является ссылочным, но объявляется как простой размерный объект, да и использован может быть так же, как простой объект. Содержимое объекта – последовательность юникодов.

Следующий пример демонстрирует возможности *замороженной* строки. Таким способом задаются строки, которые на экран выводятся так же, как они заданы в листинге. При этом игнорируются все форматирующие `esc`-последовательности, если они имеются в составе строки.

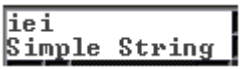
```
using System;
class Primer13_1
{
    static void Main()
    {
       string str = @"// копирующий строковый литерал
Строка1
 Строка2
  Строка3";
         Console.WriteLine(str);
    }
 }
```



Для объектов типа `string` применимы, например, следующие операции (говорят: перегружены): `+` – конкатенации; `[]` – индексации. Смысл этих операций иллюстрирует пример:

```
using System;
class Primer14
{
        static void Main()
        {
            string s = "Simple", t = "String", u="";
            u += s[1];
            u += s[5];
            u += t[3];
```

```
            Console.WriteLine(u);
            s += " ";
            s += t;
            Console.WriteLine(s);
        }
}
```



Операция присваивания

В подавляющем большинстве случаев завершающим этапом формирования того или иного результата является операция присваивания. По приоритету она самая младшая и может сопровождаться расширяющим автоприведением типов. Результат операции присваивания и по типу, и по значению соответствует операнду-приёмнику. Как и любая другая операция, операция присвоения в выражении может быть не одна. Но в этом случае в ней должны участвовать либо операнды одного типа, либо такие, чтобы последовательные присвоения выполнялись только с повышением типа:

```
using System;
class Primer14_1
{  static void Main()
    {
        float a;
        int b;
        short c;
        Console.WriteLine(a= b = c = (short)12.5 );
    }
}
```

Операторы управления

Типы вычислительных процессов

Вычислительные процессы делятся на три основных типа: линейные (рис. 7), разветвления (рис. 8), циклические (рис. 9).

Если речь идет о линейном процессе, то говорят о естественном порядке выполнения операторов (инструкций), и никакие специальные приёмы для его реализации не нужны.

Как следует из рис. 8, разветвление программы – это то, что на ассемблере, например совместно, выполняла пара команд сравнения (CMP) и условного перехода (JE, JNE, JL и т. п.). На любом высокоуровневом языке подобные действия выполняет один оператор выбора.

Рис. 7. Блок-схема линейного процесса

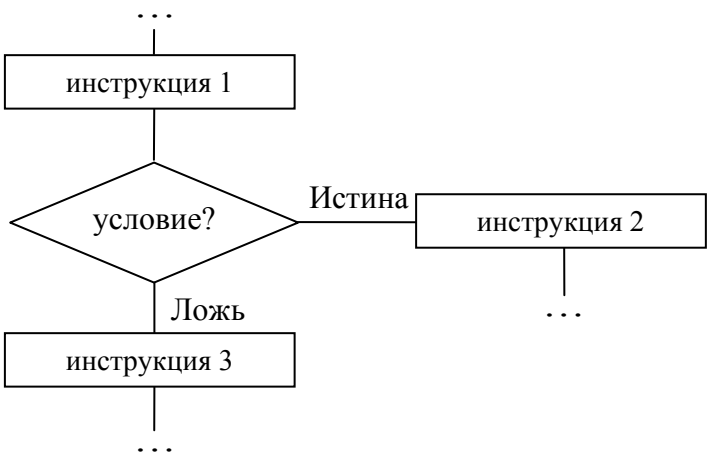


Рис. 8. Блок-схема процесса ветвления

Прикладные программы в среднем на 80–85 % состоят из циклов. В блок-схеме (рис. 9) `инструкция 3` (ещё называемая телом цикла) выполняется до тех пор, пока `условие?` даёт истинный результат. `Инструкция 2` на блок-схеме – это оператор программы, следующий после цикла. Следовательно, `инструкция 1` – это, как правило, действия, которые необходимо выполнить для подготовки к циклу.

Для организации циклов в составе языка также имеются специальные операторы.

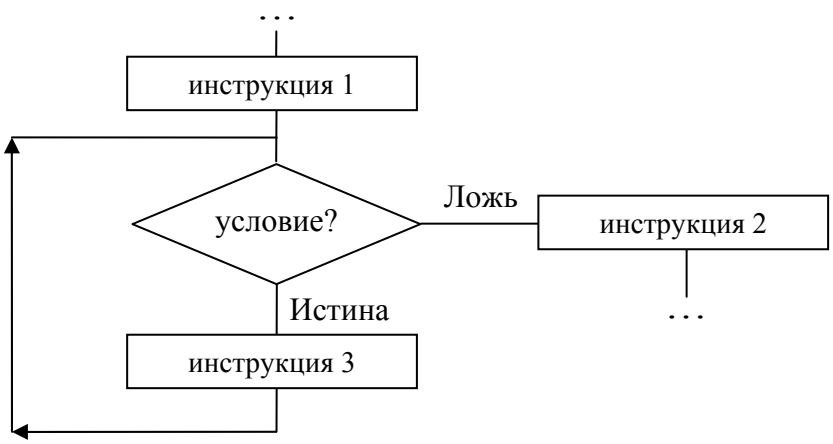


Рис. 9. Блок-схема циклического процесса

Операторы выбора

Для разветвления линейного хода алгоритма в язык C# включены следующие инструкции:

- `if`;
- `if else`;
- `if else if …`;
- `switch`.

Оператор выбора `if`

Синтаксис оператора `if` наиболее простой:

```
if (выражение) operator1;
operator2;
```

Логика работы представлена на рис. 10.

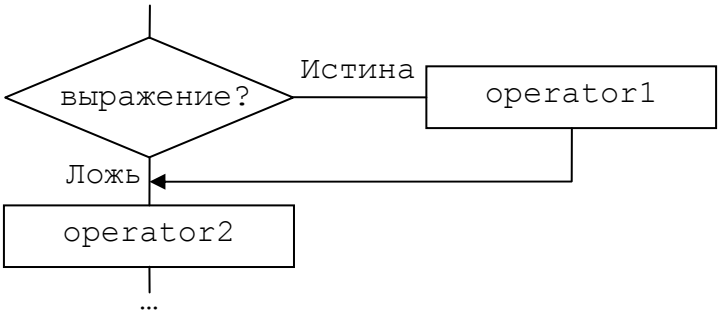


Рис. 10. Блок-схема оператора `if`

Главное условие – значение выражения должно быть булева (логического) типа. Пример на выбор максимального из двух заданных значений с использованием оператора if может быть следующим:

```
using System;
class Primer15
{
        static void Main()
        {
          int  i, j, k;
          string s;
          Console.WriteLine("Задайте первое число!");
           s = Console.ReadLine();
           i = Convert.ToInt32(s);
          Console.WriteLine("Задайте второе число!");
           s = Console.ReadLine();
           j = Convert.ToInt32(s);
           if (i < j ) k = j;
           if (i > j ) k = i;
 Console.WriteLine("Максимум из заданных чисел = {0}",k);
   }
}
```

Оператор выбора if else (рис. 11)

В предшествующем примере оператор if присутствует дважды. Может быть и иной вариант решения поставленной задачи:

```
k = j;
if (i > j ) k = i;
```

Но и в этом случае необходимо два оператора. Инструкция if else позволяет те же действия выполнить проще. Синтаксис оператора if else:

```
if  (выражение) operator1;
else operator2;
    operator3;
```

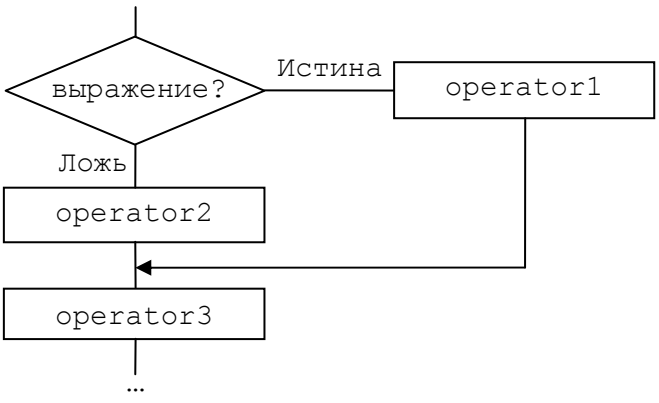


Рис. 11. Блок-схема оператора if else

Модификация примера сводится к одному оператору:

```
if (i > j ) k = i;
    else k = j;
```

Оператор выбора if else if

В ряде случаев выбор из двух возможностей оказывается недостаточным. Рассмотрим реализацию так называемой сигнальной функции:

$$f(x) = \begin{cases} 1, \text{ для всех } x > 0 \\ 0, \text{ для } x = 0 \\ -1, \text{ для всех } x < 0 \end{cases}$$

Оптимальный вариант в этом случае – использование оператора `if else if` (рис. 12):

```
if (выражение1) operator1;
else if (выражение2) operator2;
else operator3;
operator4;
```

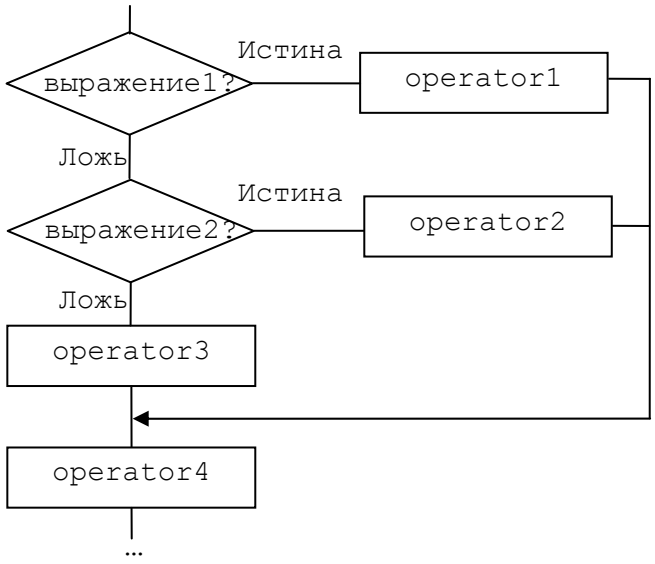


Рис. 12. Блок-схема оператора `if else if`

Рассматриваемая конструкция может иметь произвольное количество ступеней `else if`. При этом любой `else` соответствует предшествующему ему `if`. Следующий пример реализует сигнальную функцию с помощью трёхступенчатой конструкции `if else if`, которую еще называют *лестницей*:

```
using System;
class Primer16
{
   static void Main()
```

```
    {
     int x, f;
     string s;
     Console.WriteLine("\t\tРаботает сигнальная функция");
        Console.WriteLine("Задайте значение x!");
        s = Console.ReadLine();
        x = Convert.ToInt32(s);
        if (x < 0) f = -1;
           else if (x > 0) f = 1;
                    else f = 0;
 Console.WriteLine("Значение сигнальной функции =
{0}", f);
     }
}
```

```
                    Работает сигнальная функция
Задайте значение x!
100
Значение сигнальной функции = 1
```

Блочный оператор

Блочным оператором называют инструкции, размещённые внутри парных фигурных скобок. В этом отношении и тело метода, и тело класса, присутствовавшие во всех рассмотренных примерах, также являются блочными операторами, или просто блоками.

Самостоятельное значение имеют блоки в операторах *выбора* и *циклов*. Так в операторах выбора на месте `operator` может находиться *блок операторов*, инструкции в котором будут выполнены в соответствии с рассматриваемой логикой.

Следующий пример использует блочные операторы для сортировки двух значений.

```
using System;
class Primer19
{   static void Main()
    {   int i, j, max, min;
        string s;
        Console.WriteLine("Задайте первое число!");
        s = Console.ReadLine();
        i = Convert.ToInt32(s);
        Console.WriteLine("Задайте второе число!");
        s = Console.ReadLine();
        j = Convert.ToInt32(s);
        if (i > j) { max = i; min = j; }
        else { max = j; min = i; }
```

```
Console.WriteLine("Максимальное = {0}, минимальное = {1}",
   max, min);
      }
   }
```

```
Задайте первое число!
100
Задайте второе число!
-100
Максимальное  = 100,  минимальное = -100
```

Оператор множественного выбора `switch`

Оператор множественного выбора `switch` (см. рис. 13) выполняет те же действия, что и многоступенчатый `if else`, но более наглядно. Синтаксис оператора:

```
switch (выражение)
{
case константное_выражение1 : operator1; break;
case константное_выражение2 : operator2; break;
case константное_выражение3 : operator3; break;
…
default: operator; break;

}
```

Значение `выражения` в данном случае является аргументом, по которому в случае совпадения его значения со значением константного выражения выполняется так называемое *вхождение в метку* `case`. На практике часто вместо `выражения` используется простая переменная, а вместо `константных выражений` – просто константы. Важно, чтобы тип результата `выражения` соответствовал типу `константных выражений` в метках `case`. Тип может быть только целочисленным или строковым. Можно для этой цели использовать и тип `bool`, но целесообразность этого весьма сомнительна.

На месте любого из операторов может быть любая последовательность операторов (иногда её называют разделом). Однако раздел должен завершаться оператором перехода. Чаще всего для этой цели используется оператор `break`, который всегда передаёт управление за границу блока фигурных скобок. Таким образом исключается ситуация выполнения более чем одного раздела после одного вхождения в `case`. Исключением из этого правила является *пустой* раздел.

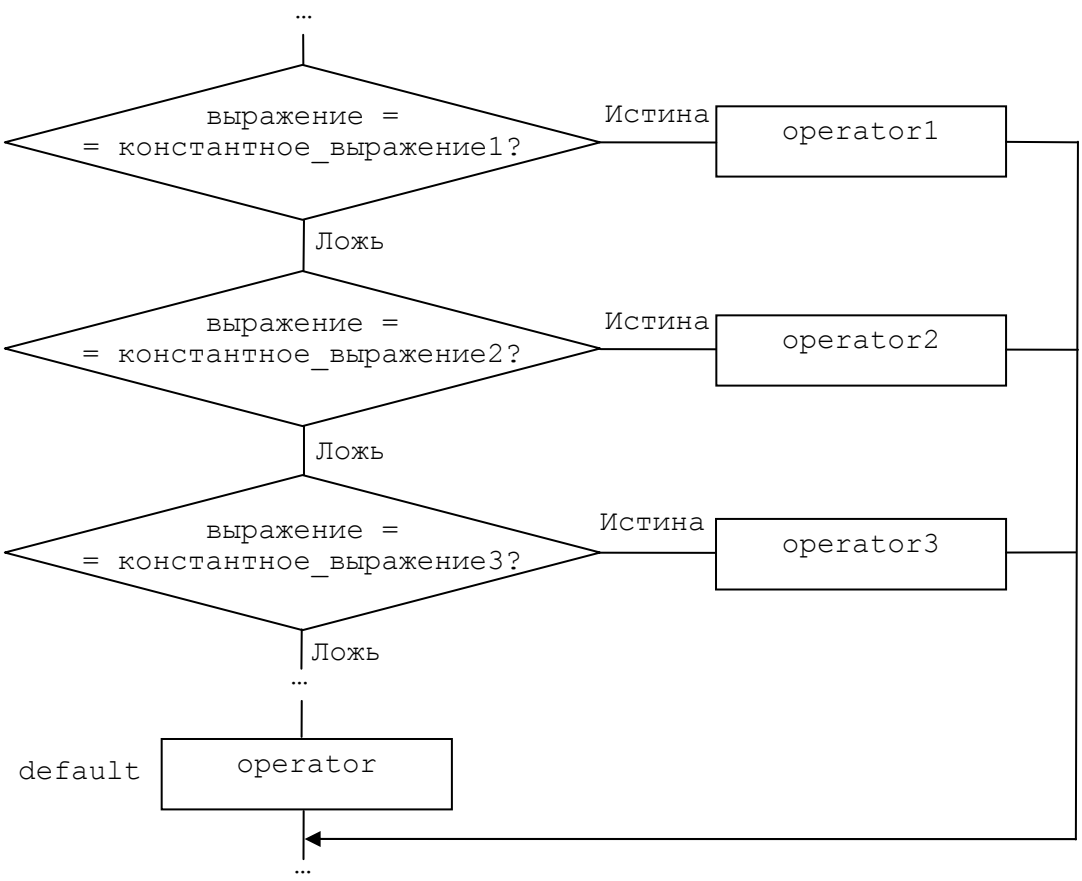
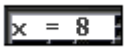


Рис. 13. Блок-схема оператора `switch`

Применение оператора `switch` покажем на примере:

```
using System;
class Primer17
{
    static void Main()
    {
        int x=3;
        switch(x)
        {
            case 2:  x+=2;   break;
            case 1:  x+=1;   break;
            case 3:
            case 4:
            case 5:  x+=5;   break;
            default: x-=10;  break;
        }
        Console.WriteLine("x = {0}", x);
    }
}
```

В результате получим



Следующий пример демонстрирует следование меток внутри блока в произвольном порядке, а также *пустые* разделы.

```
using System;
class Primer18
{
    static void Main()
    {
        int  x= 0xd;
        switch(x)
        {
            default: x += 1; goto case 3;
            case 1:  x += 2; break;
            case 2:  x += 3; goto case 1;
            case 3:
            case 4:  x += 4; goto case 2;
            case 5:  x += 5; break;
            case 6:  x += 6; break;
        }
        Console.WriteLine("x=  {0} ", x);
    }
}
```

Результат будет следующий:

```
x = 23
```

Вместо оператора `break` завершающим в операторах метки может быть оператор перехода `goto`. Он является аналогом команды безусловного перехода JMP ассемблера. Метка, на которую осуществляет переход оператор `goto`, может располагаться или в том же, или во внешнем по сравнению с данными блоке. При использовании `goto` ограничение по выполнению одного раздела после вхождения в метку `case` довольно легко можно обойти:

```
using System;
class Primer18_1
{
  static void Main()
  {   string x ;
```

```
        Console.WriteLine("Нажмите цифровую клавишу!");
        x = Console.ReadLine();
        switch(x)
    {case "0":
     case "1":
     case "2":
     case "3":
     case "4": Console.WriteLine("Клавиша интервала 0-4");
                goto case "end";//имя метки - произвольно
     case "5":
     case "6":
     case "7":
     case "8":
     case "9": Console.WriteLine("Клавиша интервала 5-9");
                goto  case "end";
     case "end": Console.WriteLine("Нажата !!"); break;
default:Console.WriteLine("Нажата нецифровая клавиша!");
break;
     }
    }
}
```

Результатом будет запись:

```
Нажмите цифровую клавишу!
3
Клавиша интервала 0-4
Нажата !!
```

Операторы циклов

Циклами называется повторение некоторого фрагмента кода необходимое количество раз. По способу организации различают циклы с пред- и постусловием. Циклы с предусловием соответствуют блок-схеме (см. рис. 9 на с. 43): условие повтора проверяется *до* тела цикла. В циклах с постусловием сначала выполняется тело цикла, а затем проверяется условие следующего повтора. В общем случае оно может проверяться и внутри тела цикла, но на практике такие циклы обычно не используются.

Любой цикл может быть реализован с помощью рассмотренных операторов выбора и передачи управления. Следующий пример демонстрирует такой вариант для программы, подсчитывающей сумму ряда натуральных чисел.

```
using System;
class Primer20
```

```
{
    static void Main()
    {
     int i, j = 1 , sum = 0;
     string s;
     Console.WriteLine("Задайте натуральное число!");
        s = Console.ReadLine();
        i = Convert.ToInt32(s);
ret:     if ( j > i) goto end;
    sum+=j;
    j++;
    goto ret;
end:
 Console.WriteLine("Сумма ряда от 1 до {0} = {1}", i, sum);
    }
}
```

```
Задайте натуральное число!
10
Сумма ряда от 1 до 10  = 55
```

Вопрос: какой тип цикла реализован в рассмотренном примере?

Для реализации циклов удобнее использовать специальные операторы. Циклы с предусловием реализуются операторами `while` и `for`, а цикл с постусловием – оператором `do while`.

Оператор цикла `while`

Цикл `while` (рис. 14) (в переводе на русский *пока*) по синтаксису наиболее прост:

```
while (выражение) оператор1;
оператор2;
```

Результат выражения должен быть булева типа, а на месте `оператора1` может быть простой (единственный), пустой (символ «`;`», перед которым ничего нет) или блочный оператор. Здесь `оператор2` – следующий после цикла оператор.

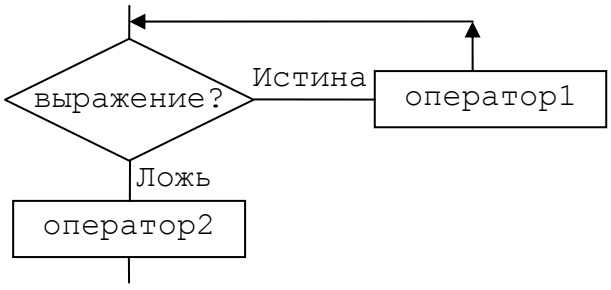


Рис. 14. Алгоритм работы цикла `while`

В следующем примере цикл while используется для вычисления факториала заданного натурального числа.

```
using System;
class Primer21
{ static void Main()
    {  uint i, j = 1 ;
       uint f = 1;
       string s;
      Console.WriteLine("Задайте натуральное число!");
       s = Console.ReadLine();
       i = Convert.ToUInt32(s);
       while ( j <= i)
    {
    f*=j;
    j++;
    }
    Console.WriteLine("Факториал  от  {0}  = {1}", i, f);
    }
}
```

```
Задайте натуральное число!
10
10!  = 3628800
```

Тип uint был использован для расширения диапазона значений, что, в свою очередь, потребовало заменить метод для преобразования введённой с клавиатуры строки.

Оператор цикла do while

Оператор do while (*делай, пока*…) используется для организации цикла с постусловием. Синтаксис в наибольшей степени отражает логику его работы:

```
do оператор; while выражение;
```

Как и в предшествующем случае, на месте оператора может быть и простой, и пустой, и блочный оператор, а результат выражения должен быть булева типа. В следующем примере использован оператор do while также для расчёта факториала.

```
using System;
class Primer22
{
    static void Main()
    {
       uint i, j = 1;
```

```
            uint f = 1;
            string s;
            Console.WriteLine("Задайте натуральное число!");
            s = Console.ReadLine();
            i = Convert.ToUInt32(s);
            do
            {
                f *= j;
                j++;
            }
            while (j <= i);
            Console.WriteLine("{0}!  = {1}", i, f);
        }
    }
```

Приведенный далее пример с помощью цикла `do while` разворачивает заданное число в обратном порядке:

```
using System;
class Primer22_1
{
    static void Main()
    {
        int num=12345, next;
        do
        {
            next = num % 10;
            Console.Write(next);
            num = num / 10;
        }
        while (num>0);
    }
}
```

Результат таков:

Оператор цикла `for`

Оператор цикла for (рис. 15) является наиболее универсальным и представляет собой своего рода шаблон для типичного цикла. Синтаксис оператора:

```
for (выражение1; выражение2; выражение3) оператор ;
```

Иногда конструкцию цикла `for` иллюстрируют следующим образом:

```
for (инициализация; условие; итерация) оператор ;
```

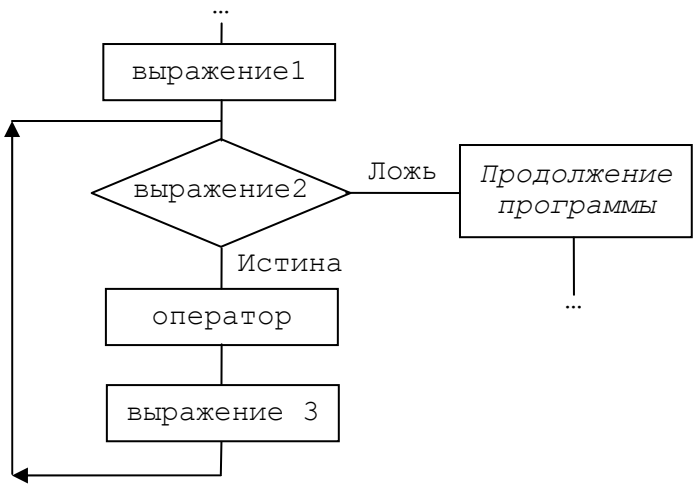


Рис. 15. Алгоритм работы цикла `for`

Порядок выполнения оператора `for` максимально соответствует общему алгоритму циклического процесса.

Пример на вычисление факториала с использованием `for` выглядит наиболее компактно:

```
using System;
class Primer23
{ static void Main()
    {   uint i, j = 1;
        uint f = 1;
        string s;
       Console.WriteLine("Задайте натуральное число!");
        s = Console.ReadLine();
        i = Convert.ToUInt32(s);
        for (f = j = 1; j < i; j++) f *= j;
        Console.WriteLine("{0}!  = {1}", i, f);
    }
}
```

Как следует из данного примера, `выражение1` (рис. 15) – это подготовка (инициализация) цикла, `выражение2` – условие повторения (его результат должен быть булева типа), `выражение3` – действия, которые выполняются в конце прохода тела цикла. Часто в `выражении1` размещают объявление переменной, с помощью которой управляют повторениями цикла (так называемая *переменная цикла*). Область видимости этой переменной в таком случае ограничена телом цикла.

Поскольку и `оператор`, и `выражение3` выполняются последовательно в одной логической цепи, несложные циклические действия могут быть выполнены и так:

```
for (f = j = 1; j < i; f*=j++);
```

Тогда формально тело цикла пустое. Отсутствовать могут любые выражения. При этом отсутствующее выражение2 считается истинным. Таким образом, следующий цикл *пустой*, но *бесконечный*:

```
for (;;);
```

Иногда цикл, в котором отсутствует тело, называют *бестелесным*. Чаще всего в качестве подобного цикла используют цикл for, встроенных возможностей которого бывает достаточно для решения несложных задач:

```
using System;
class Primer23_0
{
    static void Main()
    {
      int i, sum = 0 ;
      for (i = 1; i < 10; i+=2, sum++) ;
      Console.WriteLine("i={0}, sum = {1}", i, sum);
    }
}
```

Получим следующий результат:

```
i=11, sum = 5
```

Следует обратить внимание на значение переменной i, выведенное на консоль после окончания выполнения цикла.

При необходимости цикл for может содержать более одной управляющей переменной:

```
using System;
class Primer23_1
{
    static void Main()
    {
     int i, j,k=0 ;
     for (i = 0, j = 10; i <= j; i++, j--) k++;
     Console.WriteLine("i={0}, j = {1},  k = {2}", i,j,k);
    }
}
```

```
i=6, j = 4,  k = 6
```

Операторы `goto`, `break`, `continue` в циклах

Как следует из сказанного выше, количество повторений тела цикла зависит от результата выражения, который проверяется либо до, либо после очередного прохода. В ряде случаев необходимо досрочно завершить выполнение цикла. Для этой цели можно использовать операторы `goto` (если метка перехода расположена вне блока цикла) или `break`. Так в следующем примере факториал определяется для ограниченного диапазона натуральных чисел (поскольку уже 8! = 40 320):

```
using System;
class Primer24
{  static void Main()
    {  short i, j;
       short f ;
       string s;
      Console.WriteLine("Задайте натуральное число!");
       s = Console.ReadLine();
       i = Convert.ToInt16(s);
       for (f = j = 1; j <= i; j++)
       {
           f *= j;
           if (j == 7) break;//*
       }
       Console.WriteLine("{0}!  = {1}",i,f);//*
    }
}
```

После трёх запусков примера получаем

```
Задайте натуральное число!
6
6!  = 720
Задайте натуральное число!
7
7!  = 5040
Задайте натуральное число!
8
8!  = 5040
```

Как видно из результата, при выходе на ограничение строка на экране не вполне соответствует истине. Оператор `goto` решает данную проблему, но приводит при этом к появлению так называемого *макаронного кода*.

```
using System;
class Primer24_1
{
    static void Main()
    {
       short i, j;
       short f;
       string s;
        Console.WriteLine("Задайте натуральное число!");
       s = Console.ReadLine();
       i = Convert.ToInt16(s);
       for (f = j = 1; j <= i; j++)
       {
           f *= j;
           if (j == 7) goto m1;
       }
       Console.WriteLine("{0}!  = {1}", i, f);
       return;
   m1: Console.WriteLine("Наибольшее  {0}!  = {1}", j, f);
     }
}
```

Оператор `continue` предназначен для пропуска всех операторов в теле цикла, которые расположены *после* него. Таким образом, в цикле `for` следующим после оператора `continue` выполняется `выражение3`, а в операторах `while` и `do while` сразу после `continue` вычисляется `выражение-условие`.

```
using System;
class Primer24_2
{
    static void Main()
    {
        short i, j;
        short f;
        string s;
        Console.WriteLine("Задайте натуральное число!");
        s = Console.ReadLine();
        i = Convert.ToInt16(s);
        for (f = j = 1; j <= i; j++)
        {
            if (j > 7) continue;
            f *= j;
        }
```

```
    Console.WriteLine("{0}!  = {1}", (i<=7)?i:(short)7, f);
     }
}
```

Ещё один пример на использование оператора `continue`:

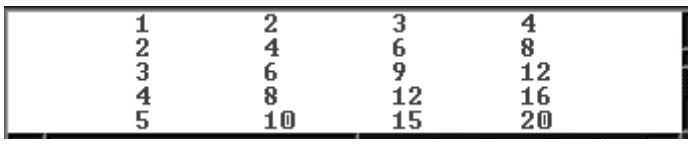
```
using System;
class Primer25
{  static void Main()
    {   short i, j;
        short f =0;
        string s;
       Console.WriteLine("Задайте натуральное число!");
        s = Console.ReadLine();
        i = Convert.ToInt16(s);
        for ( j = 1; j <= i; j++)
        {
            if ( j%2 == 0) continue;
            f += j;
        }
Console.WriteLine
("Сумма нечётных интервала 1 - {0}= {1}", j,f);
    }
}
```

```
Задайте натуральное число!
10
Сумма нечётных интервала  1 - 11  = 25
```

Вложенные циклы

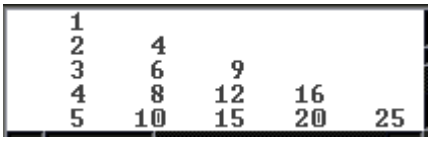
Зачастую в программах на месте тела одного цикла находится другой цикл. В этом случае первый из них называется *внешним*, второй – *внутренним*, или вложенным, а сама конструкция – *двойным циклом*. Вложенность циклов может быть тройной, четверной и так далее. Следующий пример демонстрирует двойной цикл, в котором повторения внутреннего цикла не зависят от номера прохода внешнего.

```
using System;
class Primer26
{
    static void Main()
    {
     for (short j = 1; j <= 5; Console.WriteLine(), j++)
            for (short i = 1; i < 5; i++)
                Console.Write("\t{0}", i * j);
    }
}
```



В следующем примере количество повторений внутреннего цикла зависит от номера прохода внешнего:

```
using System;
class Primer27
{
    static void Main()
    {
     for (short j = 1; j <= 5; Console.WriteLine(), j++)
            for (short i = 1; i <= j; i++)
                Console.Write("{0,5}", i * j);
     }
}
```



Массивы

Массивы – это агрегированные объекты, состоящие из заданного количества однотипных элементов. Все массивы относятся к ссылочным типам и являются производными от системного типа `SystemArray`. Массивы различают по размерности и типу составляющих его элементов. Основное преимущество массива – возможность выбора элемента по его индексу, то есть порядковому номеру в массиве. Для каждой из размерностей массива индексация элементов по умолчанию начинается с нуля.

Одномерные массивы

Объявление одномерного массива выполняется в соответствии с синтаксисом:

```
тип[] ИмяМассива=new тип[КоличествоЭлементов]
{Блок инициализаторов};
```

Оператор объявления одномерного массива может выглядеть заметно проще:

```
using System;
class Массивы1
```

```
{
static void Main()
 {
int[] m1=new int[4], m2 = new int[] {2,4,6,8}, m3 =
{1,3,5,7 };
for (short j = 0; j < 4;) m1[j] = ++j;
int сумма = 0 ;
for (short i = 0 ; i <= 3; i++) сумма += m1[i] +
m2[i] + m3[i];
Console.WriteLine("{0:d}", сумма);
 }
}
```

Оператор объявления массива в общем случае может состоять из трёх частей. Но уже и первая часть, например `int[] m1`, представляет собой завершённый оператор. Его исполнение транслятором сводится к созданию ссылки с именем `m1`. *Ссылка* – это объект особого типа, предназначенный для размещения адреса. Ссылка должна соответствовать типу объекта, адрес которого она может потенциально содержать. Однако одно объявление ссылки значение конкретного адреса не обеспечивает.

Для инициализации ссылки (задание значения) в языке C# используется оператор `new`. Его действие сводится к запросу у исполняющей системы необходимого фрагмента области динамической памяти. В общем случае область динамической памяти состоит из двух разделов: стека и кучи (`heap`). Место, где физически будет выделена память, определяется категорией объекта: ссылочные объекты размещаются в области `heap`, а значащие – в стеке.

Инициализирующий блок может содержать константы, или константные выражения. Если размер массива определён в операции `new`**,** то количество инициализаторов должно точно соответствовать количеству элементов массива. При отсутствии инициализирующего блока элементы массива получают значения по умолчанию (для числовых массивов это – нуль). Оператор `new` возвращает адрес выделенной области, и затем этим адресом инициализируется ссылка. Аргументами оператора `new` являются тип и количество элементов, которые необходимо разместить в динамической памяти. В языке C# нет необходимости заботиться об удалении объектов из динамической области после завершения их обработки. Освобождением памя-

ти занимается специальная программа *Garbage Collector (сборщик мусора)*.

Как следует из примера (на с. 60), при наличии блока инициализаторов в операторе объявления массива может быть опущено как количество элементов в операции `new` (массив `m2`), так и сама эта операция (массив `m3`). Типы констант в блоке инициализации либо должны соответствовать типу элементов массива, либо должны быть неявно приводимыми к этому типу (расширяющее неявное преобразование).

Переменная `сумма` является сумматором. Чаще всего сумматор инициализируется нулем.

Рис. 16 демонстрирует схему размещения массивов.

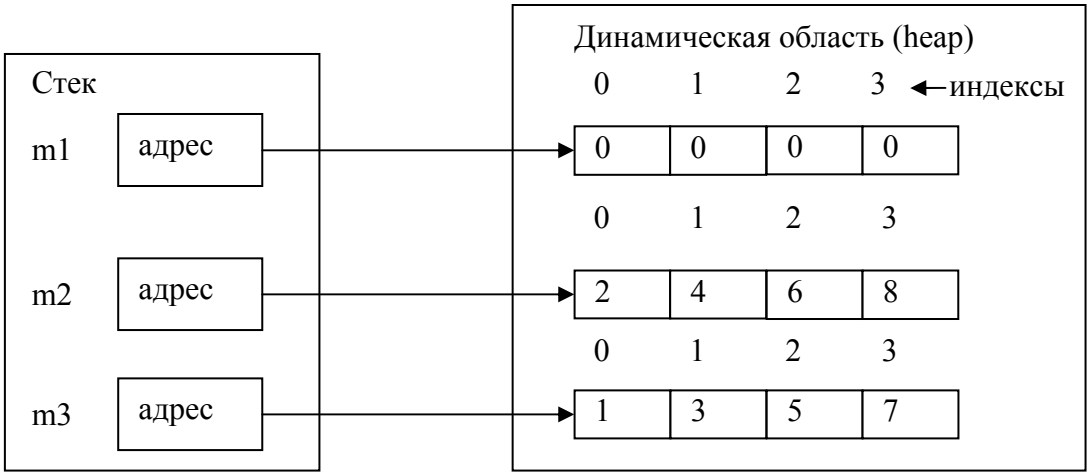


Рис. 16. Схема размещения массивов

Оператор цикла `foreach`

Оператор `foreach` (*для каждого*) (рис. 17) предназначен для работы с объектами, состоящими из некоторого набора элементов (их ещё называют коллекциями). Он также может быть использован для массивов. Синтаксис оператора `foreach`:

```
foreach  ( Элемент in ИмяМассива ) оператор ;
```

Тип `элемента` должен совпадать с типом массива. Предыдущий пример (см. с. 60) с использованием оператора `foreach` выглядит несколько проще. Нужно помнить, что `foreach` может быть использован только для *чтения* значений элементов массива, но не для их установки или изменения.

```
using System;
class Массивы2
```

```
{
static void Main()
{
int[] m1 = new int[4], m2 = new int[] {2,4,6,8},
m3 = {1,3,5,7};
   for (short j = 0; j < 4; ) m1[j] = ++j;
   int сумма = 0 ;
   foreach (int j in m1) сумма += j;
   foreach (int j in m2) сумма += j;
   foreach (int j in m3) сумма += j;
    Console.WriteLine("{0:d}", сумма);
    }
}
```

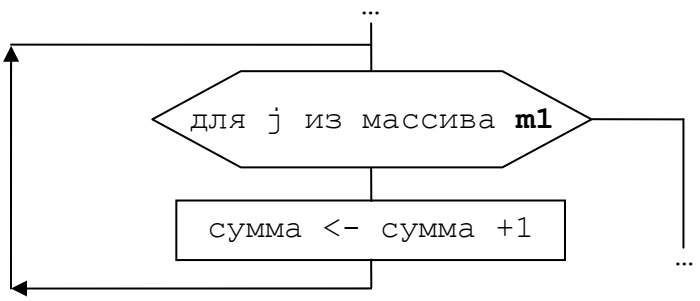


Рис. 17. Блок-схема цикла `foreach`

Базовые приёмы работы с одномерными массивами

Инициализация массивов датчиком случайных чисел

На практике (к примеру, на этапе отладки какого-либо алгоритма) зачастую бывает необходимо наполнить массив произвольными числовыми значениями. Для этого существует специальный класс `Random` (пространство имён `System`) генератора псевдослучайных значений. Технология его использования включает следующие этапы: создание объекта класса `Random`; вызов необходимого метода для созданного ранее объекта.

Наиболее часто используемые методы класса `Random`:

- `int Next()` – возвращает очередное псевдослучайное целое число в диапазоне от 0 до 0x7FFFFFFF;
- `int Next(int Max)` – то же в диапазоне от 0 до max;

- `int Next(int Min, int Max)` – в диапазоне от min до max;
- `double NextDouble()` – возвращает очередное псевдослучайное вещественное число в диапазоне от 0,0 до 1,0.

Пример:

```
using System;
class Массивы3
{
static void Main()
{
    Random Gen = new Random();
    int[] m1 = new int[10];
    for (int i = 0; i < 10; i++) m1[i] = Gen.Next(100);
    int Счётчик = 0 ;
    foreach (int j in m1)
    if (j % 2 == 0) Счётчик++;
    Console.WriteLine("Массив случайных значений");
    for (int i = 0; i < 10; i++ )
    Console.WriteLine("m1[{0}] = {1:d}", i, m1[i]);
  Console.WriteLine("Количество четных = {0}", Счётчик);
}
}
```

```
Массив случайных значений
m1[0] = 24
m1[1] = 13
m1[2] = 48
m1[3] = 42
m1[4] = 8
m1[5] = 37
m1[6] = 20
m1[7] = 98
m1[8] = 22
m1[9] = 78
Количество четных = 8
```

В данном примере в первый раз продемонстрирован вызов нестатического метода. *Методом* называется функция, являющаяся элементом класса. Функция `WriteLine` является статическим методом класса `Console`, и её вызов технологически проще: необходимо через соединитель-точку указать имя класса и имя метода. Нестатические методы класса могут быть вызваны только в привязке к экземпляру (объекту) класса. Для этого приходится объявлять объект `Gen` и выделять ему память (экземпляры класса, как и массивы, являются ссылочными объектами).

Листинг предыдущего примера умышленно выполнен без структурирования. В качестве самостоятельного упражнения нарисуйте блок-схему к этому примеру.

Экстремальные значения и статистические характеристики элементов массива

Предположим, что в массиве целых случайных значений, размер которого задаётся пользователем, требуется найти минимум, максимум и такие статистические характеристики, как среднее значение, дисперсия, среднеквадратичное отклонение:

```
using System;
class Массивы4
{
   static void Main()
  {
   string s;
   Console.WriteLine
("Задайте количество элементов массива");
   s = Console.ReadLine();
   int i = Convert.ToInt32(s);
   int[] mas = new int[i];

   Random Gen = new Random();
   for (int k = 0; k < mas.Length; k++)
   mas[k] = Gen.Next(1,10);
   Console.WriteLine("Элементы массива\n");
   foreach (int j in mas) Console.Write("{0,8}", j);
   int max = mas[0], min = mas [0];
   foreach (int j in mas)
   {
       if (max < j) max = j;
       if (min > j) min = j;
   }
   Console.WriteLine("\n Максимум= {0}, Минимум ={1}",
    max, min);
   float  среднее=0, дисперсия=0;
   foreach (int j in mas)  среднее+=j;
   среднее/=mas.Length;
   Console.WriteLine
```

```
("\nСреднее арифметическое= {0:f5}", среднее);
   foreach (int j in mas)
дисперсия+= (j-среднее)*(j-среднее);
   дисперсия/=mas.Length;
   Console.WriteLine("Дисперсия = {0:f5}",дисперсия);
   Console.WriteLine
("Среднеквадратичное отклонение= {0:f5}",Math.Sqrt(дисперсия));
  }
}
```

```
Задайте количество элементов массива
20
Элементы массива

       8       3       6       8       9       2       4       2       5       8
       1       6       9       4       5       3       3       5       5       2

 Максимум= 9, Минимум =1

Среднее арифметическое= 4,90000
Дисперсия = 5,89000
Среднеквадратичное отклонение= 2,42693
Для продолжения нажмите любую клавишу . . .
```

В примере продемонстрировано то обстоятельство, что любой массив является потомком класса `System.Array`, поэтому в составе любого массива есть элементы указанного класса. Здесь используется свойство `Length`. Пока достаточно сказать, что это – элемент объекта, с помощью которого можно получить размерность (количество элементов) данного массива. Иным способом может быть использование значение переменной `i`, ранее введённой с клавиатуры, но использование свойства `Length` является универсальным решением.

В этом же примере используется статический метод класса `Math` с именем `Sqrt`, который возвращает квадратный корень аргумента.

Методы сортировки одномерных массивов

Сортировкой принято называть процесс, позволяющий упорядочить множество подобных данных в возрастающем или убывающем порядке. При этом часть данных, определяющая место элемента в общем списке, называется ключом. При сравнении используется только значение ключа, а остальные данные, входящие в состав элемента, переносятся на новое место целиком. Ниже будут рассмотрены простейшие методы сортировки числовых массивов, в которых данные и ключ – это одно и то же.

Среди наиболее востребованных методов сортировки можно выделить:

- метод отбора (еще известен как метод минимакса) (см. рис. 18);
- перестановки (или пузырьковой сортировки) (рис. 19);
- вставки (рис. 20);
- Шелла (рис. 21);
- компаранда (рис. 22).

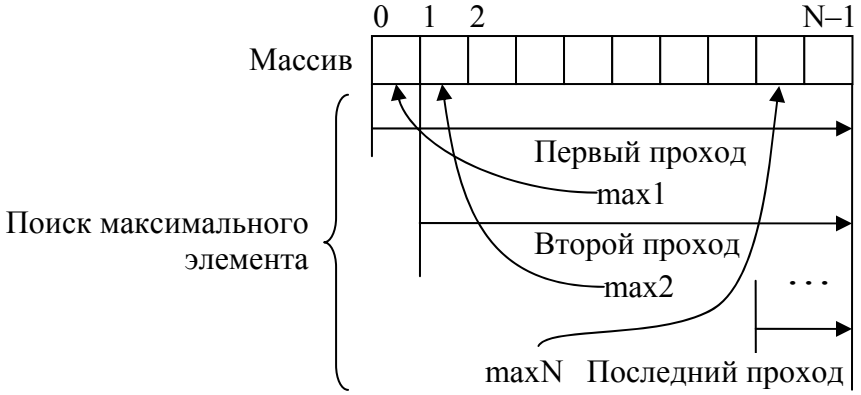


Рис. 18. Сортировка массива по убыванию методом *минимакса*

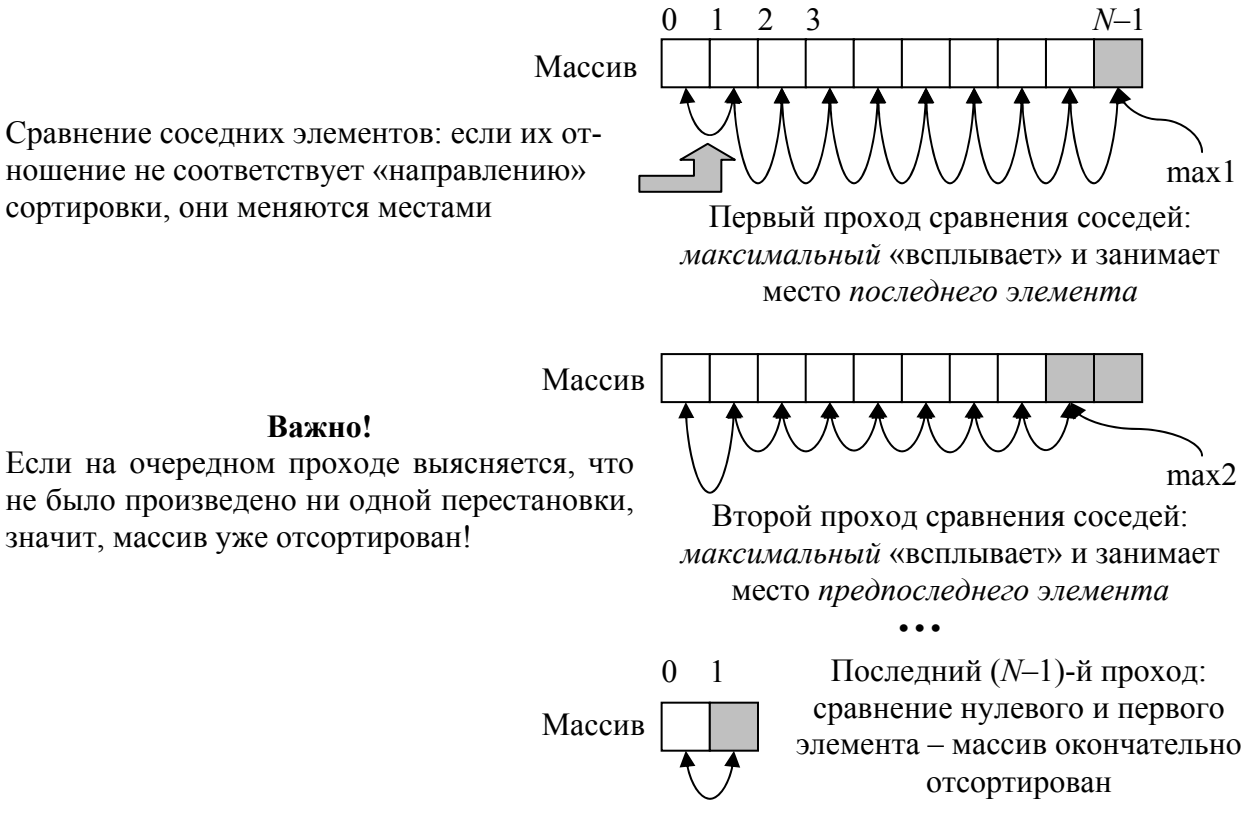


Рис. 19. Сортировка массива по возрастанию методом *пузырька*

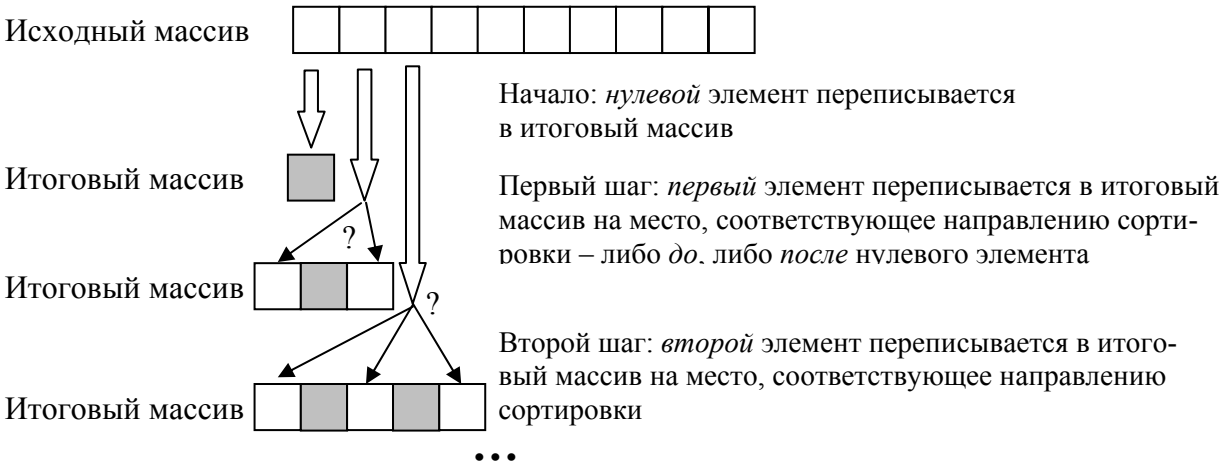


Рис. 20. Сортировка массива методом *вставки*

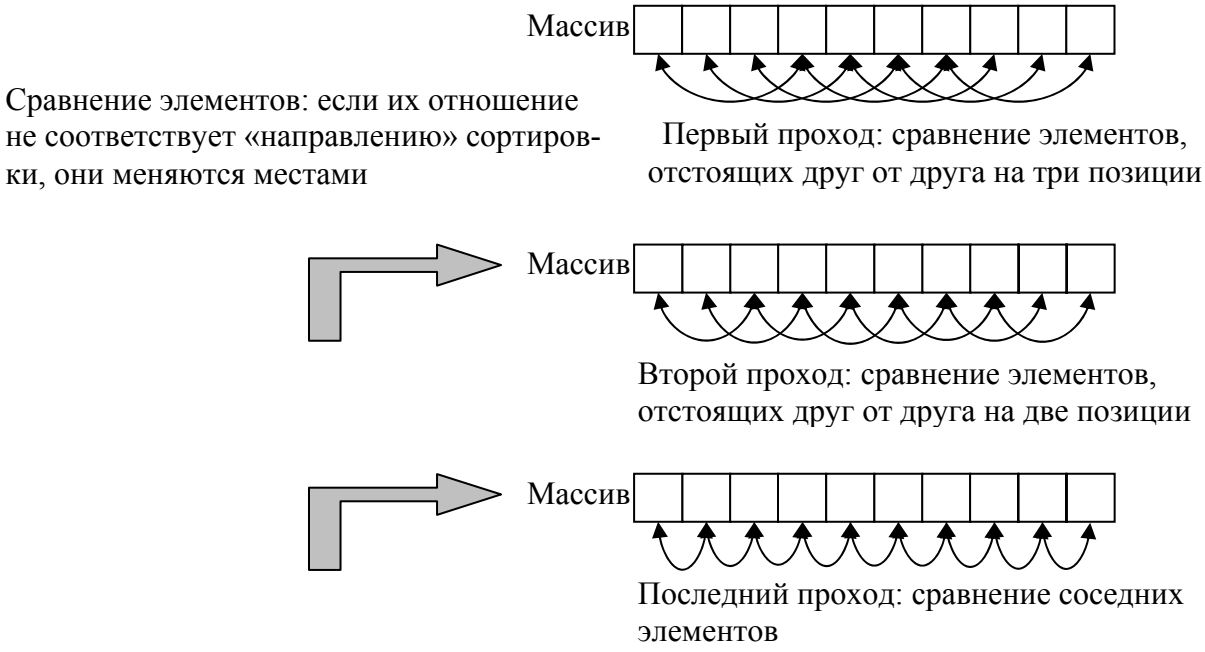
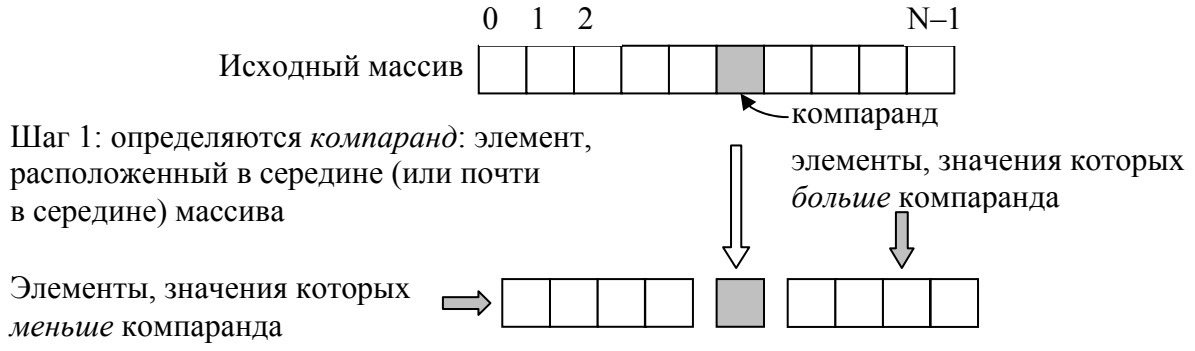


Рис. 21. Сортировка массива методом *Шелла*



Шаг 2: из исходного массива формируются два раздела: в *первый* переписываются элементы, значение которых меньше компаранда, во *второй* – больше

Для каждого из полученных разделов выполняется шаг 1 и шаг 2, пока количество элементов в получаемых разделах не станет равным единице

Рис. 22. Сортировка массива по возрастанию методом *компаранда*

Пример реализации сортировки методом минимакса:

```
using System;
class Сортировка1
{
    static void Main()
 {
    string s;
Console.WriteLine
("Задайте количество элементов массива");
    s = Console.ReadLine();
    int k = Convert.ToInt32(s);
    int[] mas = new int[k];
    Random Gen = new Random();
  for (int i=0; i<mas.Length; i++)mas[i]=Gen.Next(1,100);
    Console.WriteLine("Элементы массива");
    foreach (int j in mas) Console.Write("{0,8}", j);
    int max , imax;
    for (int i = 0; i < mas.Length - 1; i++)
    {
        max = mas[imax = i];
        for ( int j = i + 1 ; j < mas.Length; j++)
            if (max < mas[j]) max = mas[imax = j];
        mas[imax] = mas[i];
        mas[i] = max;

    }
Console.WriteLine
("\nЭлементы массива после сортировки по убыванию");
    foreach (int j in mas) Console.Write("{0,8}", j);
    }
}
```

```
Задайте количество элементов массива
30
Элементы массива
      54       1      84      15      85      28       7      20      79      85
      11      42      85      97      86      92      33      71      43      47
      11      92      36      77       2      95      55      39      88      97

Элементы массива после сортировки по убыванию
      97      97      95      92      92      88      86      85      85      85
      84      79      77      71      55      54      47      43      42      39
      36      33      28      20      15      11      11       7       2       1
```

Пример реализации пузырьковой сортировки:

```
using System;
class Сортировка2
{
  static void Main()
```

```
  {
   string s;
Console.WriteLine("Задайте количество элементов массива");
   s = Console.ReadLine();
   int k = Convert.ToInt32(s);
   int[] mas = new int[k];
   Random Gen = new Random();
 for (int i=0; i<mas.Length; i++)mas[i]=Gen.Next(1,100);
     Console.WriteLine("Элементы массива");
     foreach (int j in mas) Console.Write("{0,8}", j);
     int r ;
     bool flag = false;
     for (int i = 0; i < mas.Length - 1; i++)
     {
       flag = false;
       for ( int j = 0 ; j < mas.Length-i-1 ; j++)
            if (mas[j] < mas[j + 1]) continue;
            else
            {
               r = mas[j];
               mas[j] = mas[j + 1];
               mas[j+1] = r;
               flag = true;
             }
         if (flag == false) break;
     }
Console.WriteLine
("\nМассив после сортировки по возрастанию");
    foreach (int j in mas) Console.Write("{0,8}", j);
    }
}
```

Результат аналогичен результату выполнения примера со с. 69.

Двумерные массивы

Объявление двумерных массивов

В общем случае массивы в программе на C# могут быть произвольной размерности. На практике используются массивы с размерностью не более трех, чаще всего двумерные. При этом есть возмож-

ность использовать как прямоугольные (или квадратные) массивы, так и ломаные – со строками переменной длины.

Прямоугольные массивы объявляются в соответствии с одним из следующих вариантов синтаксиса:

```
тип[ , ] ИмяМассива = new тип [ КолСтр, КолСтолб ];
тип[ , ] ИмяМассива = { {Блок инициализаторов для строки0},
                        {Блок инициализаторов для строки1}, …};
```

В первом случае размерности массива задаются явно, а исполняющая система после выделения памяти инициализирует значения элементов массива нулями. Во втором случае опущена операция new и конфигурация массива определяется по содержимому блока инициализаторов для конкретной строки. Правила использования констант в блоке инициализаторов аналогичны одномерным массивам. При этом количество элементов в блоке инициализаторов должно быть исчерпывающим.

По своей конструкции двумерный массив является массивом ссылок (количество элементов – это число строк), каждая из которых должна быть проинициализирована адресом соответствующего фрагмента памяти из кучи.

В следующем примере демонстрируются варианты объявления прямоугольных двумерных целочисленных массивов:

```
using System;
class Массивы5
{
    static void Main()
    {
        int[,] mas1 = {{1,2,3},{4,5,6},{7,8,9}};
        int[,] mas2 = new int[2,3];
        int c1=0;

foreach (int i in mas1)Console.WriteLine("элементы
mas1={0}",i);
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++ )
                if(j==i)c1 += mas1[i,j];
Console.WriteLine
("Сумма диагональных элементов mas1={0}",c1);
    }
    }
```

```
элементы mas1=1
элементы mas1=2
элементы mas1=3
элементы mas1=4
элементы mas1=5
элементы mas1=6
элементы mas1=7
элементы mas1=8
элементы mas1=9
Сумма диагональных элементов mas1=15
```

Массив `mas2` в примере демонстрирует вариант объявления без использования инициализирующего блока. Значения его элементов нулевые. Цикл `foreach` достаточно удобен в случаях, когда осуществляется полный перебор элементов массива. Порядок этого перебора соответствует размещению элементов любого многомерного массива в памяти: первым меняется ближайший к правому краю индекс. Когда требуется иной порядок получения элементов массива (не говоря уже об изменении их значений), применяют оператор `for`.

Порядок перебора двумерного массива в порядке его размещения в памяти показан в следующем примере:

```
using System;
class Массивы5
{
    static void Main()
    {
        int[,] mas1 = {{1,2,3},{4,5,6},{7,8,9}};
        int j = 5, s = 0; ;
        foreach (int i in mas1)
        {
            s += i;
            if (--j < 0) break;
        }
        Console.WriteLine("j = {0},  s= {1}",j , s);
    }
}
```

Результатом будет

```
j = -1,  s= 21
```

Простейшие приёмы работы с двумерными массивами

При обработке двумерной квадратной матрицы удобно использовать её деление на характерные части (рис. 23).

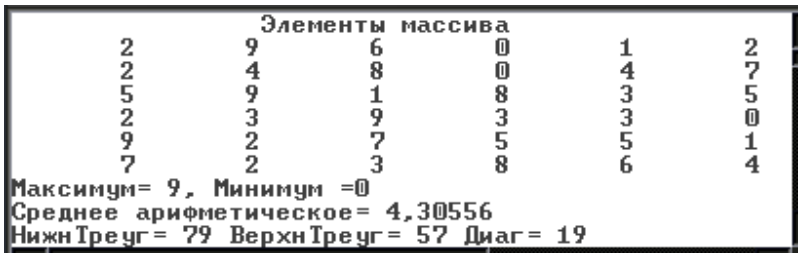
Рис. 23. Характерные части квадратной матрицы

В следующем примере приведены объявление и инициализация квадратной числовой матрицы, поиск экстремальных элементов, расчёт среднеарифметического значения, а также сумм нижне- и верхнетреугольных частей и суммы элементов главной диагонали. Для получения количества элементов по соответствующей размерности массива использован метод `GetLength`, аргументом которого является номер размерности.

```
using System;
class Массивы6
{
    static void Main()
    {
        int[,] mas = new int[6,6];
        Random Gen = new Random();
        for (int i = 0; i < mas.GetLength(0); i++)
           for (int j = 0; j < mas.GetLength(1); j++)
                mas[i,j] = Gen.Next(0,10);
        Console.WriteLine("\t\tЭлементы массива");
        int k = 0;
        foreach (int j in mas)
        {
            Console.Write("{0,8}", j);
            k++;
            if (k % 6 == 0) Console.WriteLine();
        }
        int max = mas[0,0], min = mas [0,0];
        foreach (int j in mas)
        {
            if (max < j) max = j;
            if (min > j) min = j;
        }
  Console.WriteLine
```

```
("Максимум= {0}, Минимум ={1}",max,min);
        float  среднее=0 ;
        foreach (int j in mas)  среднее+=j;
        среднее/=mas.Length;
  Console.WriteLine
("Ср. арифметическое= {0:f5}",среднее);
        int Диаг = 0, НижнТреуг = 0, ВерхнТреуг = 0;
        for (int i = 0; i < mas.GetLength(0); i++)
           for (int j = 0; j < mas.GetLength(1); j++)
                if(i>j) НижнТреуг+=mas[i,j];
                   else if (i<j)ВерхнТреуг+=mas[i,j];
                        else Диаг+=mas[i,j];
Console.WriteLine("НижнТреуг= {0:d} ВерхнТреуг= {1:d}
Диаг= {2:d}",
НижнТреуг,ВерхнТреуг,Диаг );
   }
}
```

Как следует из примера, цикл `foreach` может быть применим к массивам различной размерности.



Умножение массивов

Зачастую в программировании для именования массивов используются алгебраические термины. Так одномерный массив часто называется вектором, а двумерный – матрицей.

Типичной процедурой обработки массивов различных размерностей является их умножение:

- вектора на вектор (так называемое скалярное произведение векторов);
- матрицы на вектор;
- матрицы на матрицу.

В первом случае результатом является число (скаляр), равное сумме попарных произведений соответствующих элементов массивов. Количество элементов умножаемых массивов должно быть одинаково

$$c = \sum_{i=0}^{N-1} \boldsymbol{a}_i \cdot \boldsymbol{b}_i ,$$

где N – количество элементов массивов.

В следующем примере выполнено скалярное произведение векторов ***a*** и ***b***, проинициализированных датчиком случайных чисел:

```
using System;
class Массивы7
{
    static void Main()
    {
        int[] a = new int[10], b = new int[10];
        Random Gen = new Random();
        for (int i = 0; i < 10; i++)
        {
            a[i] = Gen.Next(0, 5);
            b[i] = Gen.Next(0, 5);
        }
        Console.WriteLine("\t\tЭлементы массивов");
        for (int i = 0; i < 10; i++)
        {
Console.WriteLine("a[{0}]={1,8} \t b[{2}]= {3,8}",
i,a[i],i,b[i]);
        }
     int c = 0;
     for (int i = 0; i < 10; i++) c +=a[i]*b[i];
     Console.WriteLine
("Скалярное произведение = {0}",c);
    }
}
```

```
                Элементы массивов
a[0]=       4   b[0]=       0
a[1]=       4   b[1]=       2
a[2]=       1   b[2]=       1
a[3]=       1   b[3]=       3
a[4]=       1   b[4]=       3
a[5]=       3   b[5]=       4
a[6]=       1   b[6]=       3
a[7]=       0   b[7]=       2
a[8]=       1   b[8]=       3
a[9]=       1   b[9]=       4
Скалярное произведение = 37
```

Произведения матрицы на вектор и матрицы на матрицу основаны на процедуре скалярного произведения векторов. Для произведения матрицы на вектор рассматриваем только случаи совпадения количества столбцов матрицы и количества элементов массива

$$c_i = \sum_{j=0}^{M-1} \boldsymbol{a}_{ij} \cdot \boldsymbol{b}_j\,,\ \forall i = \overline{0, N-1},$$

где M – количество столбцов матрицы; N – количество строк.

Произведение же матриц рассматриваем для случаев, когда количество столбцов M первой из умножаемых матриц совпадает с количеством строк второй,

$$c_{ij} = \sum_{k=0}^{M-1} \boldsymbol{a}_{ik} \cdot \boldsymbol{b}_{kj}\,,\ \forall i = \overline{0, N-1},\ \forall j = \overline{0, L-1},$$

где L – количество столбцов матрицы $\boldsymbol{B}$.

В следующем примере показано скалярное произведение матрицы $\boldsymbol{a}$ на вектор $\boldsymbol{b}$ и матрицу $\boldsymbol{d}$.

```
    using System;
    class Массивы8
{
    static void Main()
    {
int[,]  a  =  {{1,0,0,0,0},  {1,2,0,0,0},  {1,2,3,0,0},
{1,2,3,4,0}};
int[,] d = {{1,2,3}, {1,2,3}, {1,2,3}, {1,2,3}, {1,2,3}};
int[] b = {5,4,3,2,1};
int l = 0;
        Console.WriteLine("\t\tЭлементы массива a");
        foreach (int i in a)
        {
            Console.Write("{0,8}", i);
            l++;
            if (l % 5 == 0) Console.WriteLine();
        }
        l = 0;
        Console.WriteLine("\t\tЭлементы массива d");
        foreach (int i in d)
        {
            Console.Write("{0,8}", i);
            l++;
            if (l % 3 == 0) Console.WriteLine();
        }
        Console.WriteLine("\t\tЭлементы массива b");
        foreach (int j in b)
```

```
            Console.Write("{0,8}", j);
        int[] c1 = new int[4];
        int[,] c2 = new int[4,3];
        for (int i = 0; i < 4; i++)
            for (int j=0; j < 5; j++ ) c1[i] += a[i,j] *
b[j];
     Console.WriteLine
("\nПроизведение матрицы a на вектор b");
        foreach (int j in c1)Console.WriteLine(j);
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 3; j++)
               for(int k = 0; k < 5; k++)
                c2[i,j] += a[i, k] * d[k,j];
    Console.WriteLine
("\nПроизведение матрицы a на матрицу d");
    l = 0;
    foreach (int i in c2)
    {
        Console.Write("{0,8}", i);
        l++;
        if (l % 3 == 0) Console.WriteLine();
    }
  }
}
```

```
                Элементы массива a
       1       0       0       0       0
       1       2       0       0       0
       1       2       3       0       0
       1       2       3       4       0
                Элементы массива d
       1       2       3
       1       2       3
       1       2       3
       1       2       3
       1       2       3
                Элементы массива b
       5       4       3       2       1
Произведение матрицы a на вектор b
5
13
22
30

Произведение матрицы a на матрицу d
       1       2       3
       3       6       9
       6      12      18
      10      20      30
```

Двумерные ломаные (ступенчатые, зубчатые) массивы

В языке C# предусмотрена возможность задания массивов, в которых размерность последней координаты переменная. В двумерном

массиве это – длина строки (количество столбцов). В примере Массивы9 организуется такого рода массив для сохранения в нём нижнетреугольной части матрицы ***a***.

```
using System;
class Массивы9
{
    static void Main()
    {
        int l=0;
        int[,] a = { { 1, 2, 3, 4, 5 },
                     { 1, 2, 3, 4, 5 },
                     { 1, 2, 3, 4, 5 },
                     { 1, 2, 3, 4, 5 },
                     { 1, 2, 3, 4, 5 }};
        Console.WriteLine("\tЭлементы массива a");
        foreach (int i in a)
        {
            Console.Write("{0,8}", i);
            l++;
            if (l % 5 == 0) Console.WriteLine();
        }
        int[][] b = new int[4][];
        for (int i = 0; i < b.Length; i++)
            b[i] = new int[i + 1];
        for (int i = 0; i < b.Length; i++)
                for( int j=0; j<b[i].Length; j++)
                    b[i][j] = a[i+1,j];
     Console.WriteLine("\n\tЭлементы ломаного массива ");
        for (int i = 0; i < b.Length; i++)
        {
            Console.WriteLine();
            for (int j = 0; j < b[i].Length; j++)
                Console.Write("{0,8}", b[i][j]);
        }
   Console.WriteLine();
  }
}
```

Вывод в консольное окно ломаного массива ***b*** может быть организован также с помощью цикла `foreach`:

```
foreach (int[] i in b)
{
    Console.WriteLine();
    foreach (int j in i)
        Console.Write("{0,8}", j);
}
```

МЕТОДЫ (ПОДПРОГРАММЫ)

Как правило, во всех языках программирования существует возможность оформлять *часть программного кода* в форме *подпрограмм*. Обычно подпрограмма выполняет некоторый логически завершённый этап обработки данных. При этом повышается наглядность программ, упрощается процесс отладки, а также появляется возможность повторного использования ранее разработанного кода. Возможностью реализовать этапы обработки данных в виде подпрограмм пользуются программисты, пишущие на языках высокого уровня. Подпрограммы называются функциями или процедурами в зависимости от языка программирования и вида подпрограммы. В частности, язык программирования Си известен как процедурно ориентированный язык, поскольку центральной объектом в нём является как раз процедура (называемая функцией).

В языке C# также существует возможность работы с подпрограммами. В нём подпрограммы могут быть только в форме методов. Метод – это функция, являющаяся составной частью (элементом) типа (структуры или класса). Так, функция `Main`, присутствующая во всех ранее рассмотренных примерах, в то же время является методом какого-либо класса.

Синтаксис объявления и *определения* метода:

[**Модификаторы**] **Тип Имя** (*список входных параметров*) {**Тело метода**}

Здесь модификаторы – специальные ключевые слова, обеспечивающие, как правило, режим использования метода.

Тип метода – это всегда тип единственного значения, которое метод может возвращать в точку вызова. Конкретное возвращаемое значение определяется аргументом оператора `return`. После выполнения этого оператора управление передаётся в точку вызова. Если тип метода `void`, то оператор `return` в теле может отсутствовать, а возврат в вызывающую программу произойдёт после достижения потоком выполнения закрывающей скобки.

Список входных параметров представляет собой перечисление объявлений объектов, локализуемых в теле метода и используемых прежде всего для передачи значений в метод (иногда они называются

формальными). В отличие от обычных объявлений объектов, в списке входных аргументов после каждого типа следует единственное имя.

Тело метода – это фрагмент логически завершённого кода. Тело метода может иметь произвольное количество операторов `return`, каждый из которых должен возвращать значение, соответствующее типу метода.

Для передачи управления методу (для начала его работы) необходим оператор вызова, который имеет предельно простой синтаксис:

`Имя_объекта = Имя_метода``(`*`список фактических аргументов`*`)`**`;`**

Если метод объявлен типом `void` или если возвращаемое методом значение не используется, то часть оператора вызова «`Имя_объекта=`» опускается. Элементами списка фактических параметров могут быть различного рода объекты при соблюдении одного условия: тип и порядок их следования должен точно соответствовать списку входных (формальных) параметров.

Типы и аргументы методов

Примером простейшего метода – без возвращаемого значения и без входных аргументов – может служить любая из функций `Main` рассмотренных ранее примеров. Следующий пример демонстрирует вариант использования метода `среднее` также без возвращаемого значения, но уже с входными аргументами в виде трёх переменных целого типа:

```
using System;
class Методы1
{  static void среднее(int a, int b, int c)//заголовок
метода
    {// Тело метода
Console.WriteLine("Ср. арифм. значение =
{0:f5}",(a+b+c)/3.0);
    }
    static void Main()
     {
    среднее(1, 5, 10); //оператор вызова метода
}
}
```

```
Среднеарифметическое значение = 5,33333
```

В отличие, например, от Си и Си++, в C# метод может быть определён и после вызова. Это делает программу несколько проще, а для транслятора не имеет значения, так как в данном случае и `Main` и `среднее` – это методы одного класса `Методы1`:

```
using System;
class Методы1
{
    static void Main()
        {
        среднее(1, 5, 10);
        }
    static void среднее(int a, int b, int c)
        {
 Console.WriteLine
("Среднеарифметическое значение = {0:f5}",
                (a + b + c) / 3.0 );
        }
}
```

Несколько видоизменим предыдущий пример:

```
using System;
class Методы2
{
    static void Main()
    {
     int a = 1, b = 5, c = 10;
Console.WriteLine
("Аргументы до вызова = {0}, {1}, {2}",a,b,c);
     среднее(a, b, c);
Console.WriteLine
("Аргументы после вызова={0}, {1}, {2}",a,b,c);
    }
static void среднее(int a, int b, int c)
    {
Console.WriteLine
("Ср.арифм. значение = {0:f5}",(a+b+c)/3.0);
    a = b = c = 0;
 Console.WriteLine
("Аргументы в методе = {0}, {1}, {2}",a,b,c);
    }
    }
```

```
Аргументы до вызова = 1,  5,  10
Среднеарифметическое значение = 5,33333
Аргументы в методе = 0,  0,  0
Аргументы после вызова = 1,  5,  10
```

Результат на экране объясняется тем, что в C#, как впрочем и в родственных ему языках, передача данных в метод выполняется в соответствии с принципом *передачи по значению*:

- по списку входных аргументов в стеке выделяется соответствующее количество ячеек;
- в каждую из выделенных ячеек *копируется* значение объекта, присутствующего в операторе вызова метода;
- каждая из выделенных ячеек становится доступной в теле метода под именем, заданным в списке входных аргументов;
- после завершения работы метода ячейки удаляются из стека.

Как следует из рис. 21, оператором `a = b = c = 0` обнуляются объекты в стеке, в то время как значения фактических аргументов в данном случае остаются неизменными.

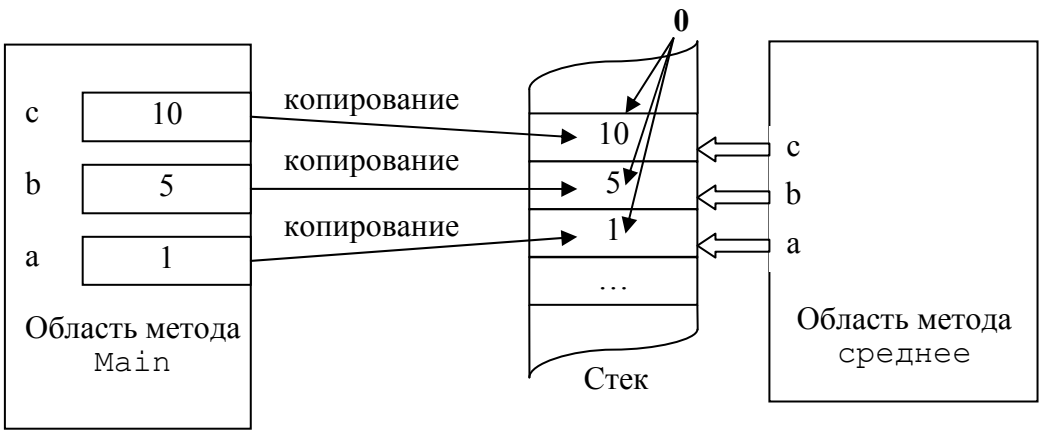


Рис. 21. Схема передачи входных аргументов

Модификаторы `ref` и `out`

Существует простой и быстрый способ передать вызываемому методу объекты, поименованные в списке входных аргументов и изменённые в методе – сделать их типы ссылочными. Для этого используется модификатор `ref`. В таком случае используется он дважды: в точке вызова и в заголовке метода.

```
using System;
class Методы3
{
  static void Main()
    {
    int a = 1, b = 5, c = 10;
```

```
Console.WriteLine
("Аргументы до вызова = {0},  {1},  {2}",a, b,c);
    среднее(ref a, ref b, ref c);
Console.WriteLine
("Аргументы после вызова={0}, {1}, {2}",a,b,c);
    }
    static void среднее
(ref int a, ref  int b, ref  int c)
    {
Console.WriteLine
("Ср.арифм. значение = {0:f5}", (a+b+c)/3.0);
        a = b = c = 0;
Console.WriteLine
("Аргументы в методе={0}, {1}, {2}", a,b,c);
    }
}
```

```
Аргументы до вызова = 1,  5,  10
Среднеарифметическое значение = 5,33333
Аргументы в методе = 0,  0,  0
Аргументы после вызова = 0,  0,  0
```

Как следует из примера, изменение переменных в методе `среднее` отразилось и на переменных в вызывающей программе, поскольку в момент вызова метода в стек помещаются адреса входных аргументов (а не копии их значений, как в предыдущем случае).

Выполним еще одно изменение метода `среднее`: рассчитанное в нём значение будем возвращать в точку вызова:

```
using System;
class Методы4
{
    static void Main()
    {
     int a = 1, b = 5, c = 10;
     float cp_ap;
     cp_ap = среднее(a, b, c);
Console.WriteLine
("Среднеарифметическое значение={0:f5}",cp_ap);
    }
static float среднее(int a, int b, int c)
    {
        return (a + b + c) / (float)3;
    }
}
```

```
Среднеарифметическое значение = 5,33333
```

Следует обратить внимание на принудительное преобразование типа в операторе `return`. Деление выполняется в вещественном типе, и результат получается требуемой точности.

С помощью оператора `return` в точку вызова можно вернуть единственное значение. В случае когда метод модифицирует значения нескольких объектов вызывающей программы, можно использовать модификатор `ref`. Существует ещё одна возможность решения подобной задачи – объявить входные параметры метода с использованием модификатора `out` (выходной). По сути ничего не меняется: вызываемому методу так же, как и в случае с модификатором `ref`, передаются адреса объектов (ссылки). При этом параметр, помеченный как выходной, с одной стороны, может не получать значения до вызова метода, но, с другой стороны, обязательно должен получить значение во время исполнения кода вызываемого метода:

```
using System;
class Методы5
{
 static void Main()
 {
 int a = 1, b = 5, c = 10;
 float cp_ap;
 среднее( a, b, c, out cp_ap);
Console.WriteLine("Среднеарифметическое
значение={0:f5}",cp_ap);
 }
static void среднее(int a, int b, int c, out float
cp_ap)
    {
         cp_ap = (a + b + c) / (float)3;
    }
}
```

Аргументы методов, объявленные с использованием модификаторов (`ref`, `out`), получили название *декорированных аргументов*.

Массив в качестве входного аргумента

Передать адрес вместо копии значения объекта можно и в случае, когда в качестве входного аргумента используется массив. Действительно, любой массив – это ссылочный объект, и его значением является адрес памяти, в которой размещены элементы массива.

При этом так же, как и в случае с параметрами `ref` и `out`, метод получает возможность модификации объектов вызывающего метода.

```
class Методы6
{    static void Main()
    {int[] a = { 1, 5, 10 };
     float cp_ap;
     среднее(a, out cp_ap);
Console.WriteLine
("Среднеарифметическое значение={0:f5}",cp_ap);
     Console.WriteLine("a[0] = {0}", a[0]);
    }
static void среднее(int[] mass, out float cp_ap)
{       cp_ap = 0;
        foreach( int i in mass)
                cp_ap += i;
        cp_ap /= mass.Length;
        mass[0] = 0;
 }
}
```

```
Среднеарифметическое значение = 5,33333
a[0] = 0
```

Модификатор `params`

Если в точке вызова метода используется несколько входных аргументов одного типа, то в самом методе они могут быть использованы как элементы массива. Для этого в заголовке метода объявляется массив соответствующего типа с модификатором `params`:

```
using System;
class Методы6
{
   static void Main()
   {
    int a = 1, b = 5, c = 10;
    float cp_ap;
    среднее( out cp_ap, a, b, c);
Console.WriteLine
("Среднеарифметическое значение={0:f5}", cp_ap);
    }
static void среднее(out float cp_ap, params int[]
mass)
{
        cp_ap = 0;
```

```
        foreach( int i in mass)
                cp_ap += i;
        cp_ap /= (float)3;
    }
}
```

Поскольку массив, объявленный в заголовке метода с модификатором `params`, объединяет в себя неограниченное количество входных аргументов *своего* типа, постольку размещаться он может только в *конце* списка формальных аргументов.

Следующий пример подчёркивает отличие параметрирования от использования массива в качестве входных аргументов:

```
using System;
class Методы6_1
{ static void Main()
    {int a =  1  , b= 5, c= 10 ;
     float cp_ap;
     cp_ap =  среднее( a, b, c);
Console.WriteLine
("Среднеарифметическое значение={0:f5}",cp_ap);
    Console.WriteLine(a);
    }
static float среднее(params int[] mass)
{       float cp_ap = 0;
        foreach( int i in mass)
               cp_ap += i;
        cp_ap /= mass.Length;
        mass[0] = 0;
        return cp_ap; }  }
```

```
Среднеарифметическое значение = 5,33333
1
```

Как следует из последней строчки результата, обнуление нулевого элемента массива `mass` в методе `среднее` никоим образом не изменило содержимого переменной `a` в методе `Main`.

Можно отметить, что объединение в массив набора однотипных входных аргументов только изменяет способ использования ячеек стека, которые содержат копии значений объектов из списка в инструкции вызова.

Перегрузка методов

Набор таких характеристик, как тип, количество и порядок следования аргументов метода, получил название *сигнатуры*. Язык C# позволяет разрабатывать и использовать методы с одинаковыми именами, если они различаются по сигнатуре. Это бывает удобно, когда единообразная обработка выполняется для различных типов данных.

Следует отметить, что тип самого метода в сигнатуру *не входит*.

```
using System;
class Методы7
{  static void Main()
    {
        bool a = true, b = false;
        int  c = 10, d = 15;
        andor(a, b);
        andor(c, d);
    }
 static void andor(bool a, bool b)
 {   Console.WriteLine("ИЛИ {0}", a || b);
    Console.WriteLine("И   {0}", a && b);
 }
 static void andor(int a, int b)
 {  Console.WriteLine("ИЛИ {0}", a | b);
    Console.WriteLine("И   {0}", a & b);
 }
}
```

```
ИЛИ True
И   False
ИЛИ 15
И   10
```

Рекурсивные методы

Рекурсивными называют методы, которые непосредственно или косвенно вызывают самих себя. Классическим примером может служить метод, вычисляющий факториал:

```
using System;
class Методы8
{
    static void Main()
    {
        int i;
        string s;
        Console.WriteLine("Задайте число!");
        s = Console.ReadLine();
        i = Convert.ToInt32(s);
```

```
        Console.WriteLine("    {0}!    =    {1}",    i,
факториал(i));
    }
    static long факториал(int i)
    {
        if (i == 1) return 1;
        return i * факториал(i - 1);
    }
}
```

```
Задайте число!
10
 10! = 3628800
```

Еще одним примером применения рекурсии является метод вычисления значения показательной функции:

```
using System;
class Методы9
{
    static void Main()
    {
        int n;
        double a;
        string s;
Console.WriteLine("Возведение в степень числа a");
Console.WriteLine("Задайте число a!");
        s = Console.ReadLine();
        a = Convert.ToDouble(s);
Console.WriteLine("Задайте число n!");
        s = Console.ReadLine();
        n = Convert.ToInt32(s);
Console.WriteLine(" Число {0} в степени {1} = {2}",
                a,n,экспонента(a,n));
    }
    static double экспонента(double a, int n)
    {
        if (n < 0 || a == 0 ) return 0;
        if (n == 0) return 1;
        return a * экспонента(a, n-1);
    }
}
```

```
Возведение в степень числа a
Задайте число a!
2
Задайте число n!
16
 Число 2 в степени 16 = 65536
```

Предложенный в примере вариант вычисления показательной функции модифицируйте для работы с отрицательными степенями.

СТРОКИ

Строки в языке C# являются объектами встроенного класса `System.String` и поэтому относятся к ссылочным типам: при объявлении локальной строки в стеке создаётся ссылка на объект динамической области.

Для работы обычно используют псевдоним `string`. Фактически рассмотренные ранее литеральные строки (строковые константы) также являются объектами класса `System.String`.

Содержимым строкового объекта является массив `char`-элементов. В отличие от *терминальных* (нуль-ограниченные) строк в некоторых других языках программирования, в частности в C и C++, нуль-символ (символ с кодом 0) в конце `char`-массива отсутствует.

Объявление строк

Объявление объекта класса `string` во многом похоже на объявление массивов с начальной инициализацией. Как и у массивов, у любой строки есть свойство `Length` и операция индексирования.

```
using System;
class Строки1
{
static void Main()
    {
string строка1="Содержимое строки";
Console.WriteLine
("строка1 \"{0}\" длиной= {1} символов",
строка1, строка1.Length);
char[] массив = {'Т','о','ж','е',' 
',' с','т','р','о','к','а'};
string строка2 = new string(массив);
Console.WriteLine
("строка2 \"{0}\" длиной= {1} символов",
строка2, строка2.Length);
Console.WriteLine
("Начальные символы  {0} и {1}",
строка1[0], строка2[0]);
string строка3 = new string('A',10);
string строка4 = new string(массив, 5, 6);
Console.WriteLine
("строка3 ={0}, строка4 = {1}",
строка3, строка4);
    }
}
```

```
строка1 "Содержимое строки" длиной= 17 символов
строка2 "Тоже строка" длиной= 11 символов
Начальные символы  C и T
строка3 =AAAAAAAAAA, строка4 = строка
```

В предшествующем примере используются четыре варианта начальной инициализации строки. В первом случае (`строка1`) это происходит с помощью строкового литерала. Для объекта `строка2` используется `char`-массив, все элементы которого объединяются в строку. Для объекта `строка3` используется одиночный `char`-символ и коэффициент его повторения в формируемой строке. В последнем случае из массива `char`-элементов выбирается фрагмент: третий параметр – индекс начального символа фрагмента, четвёртый параметр – количество элементов массива, используемых для инициализации строки. Фактически это использование четырёх (перегруженных) вариантов `конструктора` строки – специального метода, предназначенного для её создания.

Для вставки в строки символа кавычек был использован обратный слеш, так как без использования этого случая первая парная кавычка закрывает управляющую строку метода `Console.WriteLine`.

Следующий пример демонстрирует ещё одно проявление *родственности* строк и массивов – использование для перебора элементов цикла `foreach`:

```
using System;
class Строки1_1
{
    static void Main()
    {
    string строка1="Столбец!";
    foreach( char a in строка1)
      Console.WriteLine(a);

    }
 }
```

```
С
т
о
л
б
е
ц
!
```

Принципиальное отличие строки `string` от других объектов заключается в том, что строка является неизменяемым объектом – строковой константой. Иначе говоря, однажды созданную (и наполненную содержимым) строку изменить никаким легитимным способом нельзя! На практике это правило проявляется следующим образом:

- в ходе трансляции не создается новая строка, если содержимое инициализирующей её константы совпадает с содержимым ранее объявленной строки. Ссылка инициализируется адресом уже размещённых данных;
- в ходе выполнения не создается новая строка, если для её создания используется метод `String.Intern(string st)`, а содержимое строки `st` совпадает с одной из уже имеющихся в области видимости строк;
- при любом изменении содержимого строки создается новый экземпляр класса `string` с модифицированным содержимым;
- в операции индексирования символы строки доступны только в режиме чтения, так как размещённая в строке информация может быть использована произвольным количеством строковых объектов.

Операции и методы для работы со строками

Поскольку строки `string` являются экземплярами (объектами) класса `System.String`, постольку для работы с ними имеются такие инструменты этого класса, как свойства, операции и методы. Наиболее часто используемые строковые операции и методы содержит табл. 3.

Таблица 3

Элементы класса `System.String`

Инструмент	Назначение	Примечание
Операции		
+	Конкатенация (слияние) строк. Количество объединяемых строк не ограничено	–
==	Посимвольное сравнение содержимого двух строк на предмет их совпадения (`true`)	Операции выполняются с фактическим содержимым строк
!=	Посимвольное сравнение содержимого двух строк на предмет их несовпадения (`true`)	

Окончание табл. 3

Инструмент	Назначение	Примечание
Методы		
int CompareTo(string str)	Сравнение вызывающей строки со строкой `str`: если значение вызывающей строки меньше, то возвращает отрицательное число, если больше – положительное, если равно – нуль	Выполняется сравнение кодов символов
static string Copy(string str)	Копирование строки	Возвращает ссылку на «глубокую копию» строки `str`
int IndexOf(string str)	Поиск подстроки	Возвращает индекс позиции, где первый раз в вызывающей подстроке была обнаружена строка `str`
int LastIndexOf(string str)	Поиск последней подстроки	Возвращает индекс позиции, где последний раз в вызывающей подстроке была обнаружена строка `str`
string Insert(int index, string str)	Вставка в вызывающую строку подстроки `str` с индекса `index`	–
static string Join(string str, params string[] mas)	Объединение разделителем `str` строк из массива `mas`	
string[] str Split(params char[] del)	Расщепление вызывающей строки на массив строк `str`, с помощью набора разделителей `del`	
string Replace(string strold, string strnew)	Заменяет в вызывающей строке подстроку `strold` на `strnew`	
string SubString(int begin, int end)	Выделение из вызывающей подстроки, которая начинается с индекса `begin` и заканчивается индексом `end`	

Следующий пример демонстрирует приёмы объединения и сравнения строк.

```
using System;
class Строки2
{
 static void Main()
 {
  string строка1="Первая";
```

```
  string строка2 ="Первая";
  Console.WriteLine("Строки 1и2 совпадают  ? = {0}",
                    строка1==строка2);
  строка1 += строка2;
Console.WriteLine("Строки 1и2 совпадают  ? = {0}",
                    строка1 == строка2);
  строка2 += "строка";
Console.WriteLine("Строки 1и2 совпадают  ? = {0}",
                    строка1 == строка2);
  int res = строка1.CompareTo(строка2);
      if(res==0) Console.WriteLine("Строка 1 == 2");
      else if (res < 0) Console.WriteLine
("Строка 1 < 2");
      else Console.WriteLine("Строка 1 > 2");
      Console.WriteLine("строка1 = {0}", строка1);
      Console.WriteLine("строка2 = {0}", строка2);
 }
}
```

```
Строки 1и2 совпадают  ? = True
Строки 1и2 совпадают  ? = False
Строки 1и2 совпадают  ? = False
Строка 1 < 2
строка1 = ПерваяПервая
строка2 = Перваястрока
```

Как следует из полученного результата `строка1` оказалась меньше `строки2`, так как `П` < `с`.

Операция поразрядного `исключающего ИЛИ` обладает свойством цикличности: дважды выполненная с одним и тем же операндом, она позволяет восстановить первоначальное значение второго операнда. Это позволяет использовать данную операцию для простейшего шифрования, например, текстовых данных:

```
using System;
class Строки2_1
{
 static void Main()
 {
     string s ="Пин-код";
     char  c = 'C';
     int j = 0;
     char[] mas = new char[s.Length];
     foreach (char i in s) mas[j++] = (char) (i ^ c);
     string s1 =  new string(mas);
```

```
        j = 0;
       foreach (char i in s1) mas[j++] = (char) (i ^ c);
        string s2 = new string(mas);
        Console.WriteLine("Исходная строка:\t\t"+s);
        Console.WriteLine("Закодированная строка:\t\t" +
s1);
        Console.WriteLine("Декодированная строка:\t\t" +
s2);
 }
}
```

```
Исходная строка:                Пин-код
Закодированная строка:          ???n???
Декодированная строка:          Пин-код
```

Массивы строк

Подобно другим типам данных, строки могут быть объединены в массивы. Правила объявления, инициализации и использования массивов строк мало чем отличаются от аналогичных для числовых массивов:

```
using System;
class Строки3
{
 static void Main()
 {
     string[] мс1 ={"один", "два", "три"};
     string[] мс2 = new string[3];
     мс2[0] = "четыре";
     мс2[1] = "пять";
     мс2[2] = "шесть";
     Console.WriteLine("Массив мс1:");
     foreach( string i in мс1)
           Console.Write(i+" ");
       Console.WriteLine("\nМассив мс2:");
     foreach (string i in мс2)
           Console.Write(i + " ");
 }
}
```

```
Массив мс1:
один два три
Массив мс2:
четыре пять шесть
```

В примере `Строки4` массив строк `mass` используется для размещения расщеплённой строки:

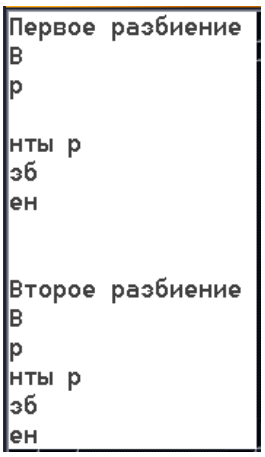
```
using System;
class Строки4
{
 static void Main()
 {
  string строка1="29.01.09";
  Console.WriteLine("строка1 = {0}", строка1);
  строка1 = строка1.Replace("09", "2009");
  string[] mass = строка1.Split('.');
  строка1 = string.Join(":", mass);
  Console.WriteLine("строка3 = {0}", строка1);
 }
}
```

```
строка1 = 29.01.09
строка3 = 29:01:2009
```

Как и конструкторы, большинство методов класса `string` имеют вариации. Следующий пример демонстрирует использование двух перегрузок метода `Split`:

- `Split(char[],StringSplitOptions)` – использование для разбиения массива `char`-разделителей;
- `Split(string[],StringSplitOptions)` – то же, но строк-разделителей.

```
using System;
class Строки5
{
 static void Main()
 {
  string строка1 = "Варианты разбиения";
  string[] ms = { "а","и","я"};
  string[] mass = строка1.Split(ms,
StringSplitOptions.None);
  Console.WriteLine("Первое разбиение");
  foreach (string s in mass) Console.WriteLine(s);
string[]
mass1 = строка1.Split(ms,
StringSplitOptions.RemoveEmptyEntries);
  Console.WriteLine("Второе разбиение");
  foreach (string s in mass1) Console.WriteLine(s);
 }
}
```



Метод `Split` возвращает массив подстрок заданной строки, разделенных заданными символами или строками. Если встречаются смежные разделители, то в массив помещаются пустые строки (""). Значения из перечисления `StringSplitOptions` указывают, включается ли в возвращаемый массив элемент, содержащий пустую строку. При задании значения `None` метод `Split` возвращает массив, содержащий как пустые, так и непустые подстроки. При задании значения `RemoveEmptyEntries` метод `Split` возвращает массив, содержащий только непустые подстроки.

СТРУКТУРЫ

Как и большинство современных языков высокого уровня, C# предоставляет программисту возможность создания своих собственных (пользовательских) типов данных. В полномасштабном варианте использование этой возможности как раз и составляет существо объектно ориентированной технологии программирования.

Несколько сокращённые возможности по объявлению и использованию собственных типов могут быть реализованы с помощью структур, которые являются упрощенной разновидностью класса. Перечислим основные особенности структур:

- любая структура на C# является производной от системного класса `ValueType`, который, в свою очередь, производен от `Object` (см. рис. 4 на с. 11). На практике это означает, что, во-первых, в любом объекте структурного типа скрыто присутствуют такие методы, как `ToString()` и `GetType()`, а во-вторых, объекты структурных типов являются *значимыми*. Иначе говоря, объекты структурных типов содержат значения (а не ссылки) и размещаются в `стеке` (а не в `куче`);
- структуры не могут быть использованы в системах наследования, состоящих из структур и классов, но могут наследовать интерфейсам;
- структуры могут иметь элементы-данные (поля) и элементы-функции (методы, свойства, операции).

В следующем примере показано объявление нового типа с именем `A`, объявление экземпляра этого типа с именем `obj`, наполнение его данными и вывод содержимого на консоль.

```
using System;
class Структуры1
{
 struct A {
            public string s;
            public bool a;
            public int b;
          }
 static void Main()
  {
     A obj;// поля не инициализируются!
```

```
        obj.a = true;
        obj.b = 10;
        obj.s = "Объект типа A";
        Console.WriteLine("{0} a={1}
b={2}",obj.s,obj.a,obj.b);
      }
  }
```

```
Объект типа A a=True b=10
```

Блок с описанием состава структуры иногда называют *тегом*. Именно тег задаёт конструкцию пользовательского типа. В данном случае (`struct A`) он состоит из трёх полей с именами `s`, `a`, `b`. Все поля имеют модификатор `public`, который регламентирует их *доступность* в любой точке области видимости экземпляра типа. Реализуется тип `A` в операторе объявления `A obj`. Доступ к содержимому полей объекта осуществляется с помощью выражения `ИмяОбъекта.ИмяПоля`.

Конструкторы

В общем случае все методы класса или структуры можно разделить на *особые* и *неособые*. Особые, в свою очередь, делятся на конструкторы и деструкторы. У структур деструктора быть не может. А конструкторы, которые могут входить в состав структуры, – это специальные методы, вызываемые в момент создания экземпляра этого типа. Один конструктор есть у структуры всегда. Его создаёт транслятор, и поэтому он называется *конструктором по умолчанию*. Действие конструктора по умолчанию – разместить элементы структуры, перечисленные в теге (структурный тип значащий, следовательно, объект содержит данные). Перегрузить конструктор (то есть заменить каким-либо иным) нельзя. В то же время никто не запрещает иметь произвольное количество конструкторов с непустым списком входных аргументов (с различной сигнатурой). Для разработки собственного конструктора нужно руководствоваться следующими правилами:

- конструктор не имеет типа, а следовательно, не может возвращать какого-либо значения;
- имя конструктора всегда *совпадает* с именем структуры;
- неявно конструктор по умолчанию структур вызывается оператором объявления объекта данного типа;

- управление конструктору в случае явного вызова передаётся с помощью оператора `new ИмяТипа(аргументы)`.

Следующий пример демонстрирует *расширение* уже рассмотренного типа `A` с помощью включения в него трёх конструкторов.

```
using System;
class Структуры2
{
 struct A {
            public string s;
            public bool a;
            public int b;
     public A(int inb)
 { b = inb; a = true; s = "Создан конструктором A(int)"; }
     public A(bool ina)
 { b = 10; a = ina; s = "Создан  конструктором A(bool)"; }
  public A(string ins)
  { b = 10;
    a = true;
    s = ins + " создан конструктором A(string)"; }
  }
 static void Main()
    {
    A obj=new A(),
      obj1=new A(100),
      obj2=new A(true),
      obj3=new A("obj3");
    Console.WriteLine
("s={0} a={1} b={2}",obj.s,obj.a,obj.b);
    Console.WriteLine
("s={0} a={1} b={2}",obj1.s,obj1.a,obj1.b);
    Console.WriteLine
("s={0} a={1} b={2}",obj2.s,obj2.a,obj2.b);
    Console.WriteLine
("s={0} a={1} b={2}",obj3.s,obj3.a,obj3.b);
 }
}
```

```
s= a=False b=0
s=Создан конструктором A(int) a=True b=100
s=Создан  конструктором A(bool) a=True b=10
s=obj3 создан конструктором A(string) a=True b=10
```

Объект `obj`, как и в предыдущем примере, создаётся с помощью конструктора по умолчанию. Явный способ вызова данного конструктора приводит к тому, что поля создаваемого конструктором по умолчанию объекта `obj` будут проинициализированы значениями *по умолчанию*. В предыдущем примере конструктор по умолчанию не инициализировал поля объекта, что приводило к необходимости определения их значений до использования в методе `Console.WriteLine`. В рассматриваемом примере значения, полученные полями в результате инициализации значениями по умолчанию, выведены на консоль в первой строке консольного экрана. Все объекты размещаются в стеке.

Конструкторы копии

Для создания объекта, являющегося точной копией уже существующего экземпляра структурного типа, достаточно вполне очевидного оператора:

```
Тип НовыйОбъект = СтарыйОбъект;
```

В этом случае действия, необходимые для создания объекта, (так называемое *поверхностное копирование*) выполняет конструктор копии по умолчанию. В состав структуры можно добавить и собственный эксклюзивный конструктор копии, тогда конструктор копии по умолчанию (как и простой конструктор по умолчанию) в типе остаётся и может быть использован, например:

```
using System;
class Структуры3
{
 struct A
    {
            public string s;
            public bool a;
            public int b;
     public A(int inb)
 { b = inb; a = true; s = "Создан конструктором A(int)"; }
     public A(A inA)
       { b = inA.b+1;
         a = !inA.a;
         s = inA.s.Replace("A(int)","копии");
       }
    }
 static void Main()
```

```
    {
    A obj=new A(1),
      obj1=obj, obj2 = new A(obj);
Console.WriteLine("s={0} a={1} b={2}",obj.s,obj.a,obj.b);
Console.WriteLine("s={0} a={1} b={2}",obj1.s,obj1.a,obj1.b);
Console.WriteLine("s={0} a={1} b={2}",obj2.s,obj2.a,obj2.b);
   }
}
```

```
s=Создан конструктором A(int) a=True b=1
s=Создан конструктором A(int) a=True b=1
s=Создан конструктором копии a=False b=2
```

При автовызове конструктора копии управление передаётся конструктору *копии по умолчанию.* Все объекты в примере существуют независимо друг от друга, так как являются объектами значащего (структурного) типа.

Неособые методы

Объекты, используемые в предыдущем примере, агрегируют соответствующий набор данных (и этим повышается *прозрачность* кода), но не обеспечивают их защиты, так как поля объявлены с модификатором `public`.

При отсутствии какого-либо модификатора элемента структуры или класса (режим *по умолчанию*) устанавливается защита `private` (частный), которая разрешает использование элемента только своим методам. Среди этих методов программист должен предусмотреть такие, которые объявлены с модификатором `public` и позволяют обрабатывать должным образом поля или вызывать частные методы класса.

Следующий пример демонстрирует определение и использование метода `type` для работы с комплексными числами.

```
using System;
class Структуры4
{
 struct Complex
  {
    double Real,Image;
    public Complex(double inR, double inI)
              { Real = inR; Image = inI;}
    public Complex(int inR, int inI)
              { Real = inR; Image = inI; }
    public void type()
```

```
{Console.WriteLine("{0}+j{1}",Real,Image);}
  }
 static void Main()
    {
     Complex a = new Complex(1.5, 2.5),
b = new Complex(15, 25);
     a.type();
     b.type();
    }
}
```

```
1,5+j2,5
15+j25
```

В типе `Complex` имеется два явных конструктора, каждый под соответствующий набор входных данных, и метод `type` для демонстрации текущего содержимого объекта.

Переопределение методов

Предложенный в последнем примере вариант получения данных, инкапсулированных внутри объекта, носит учебный характер. Любой объект имеет в своём распоряжении унаследованный от `Object` метод `ToString()`, выполняющий преобразование содержимого объекта из внутреннего представления в строку. Для пользовательских типов имеется возможность разработать свой (переопределённый) вариант метода `ToString()` (эта возможность обеспечивается модификатором `virtual`, с которым метод объявлен в классе `Object`). В следующем примере метод `ToString()` в своём изначальном виде используется как для встроенного, так и для нового типа:

```
using System;
class Структуры5
{
  struct Complex  {
                   double Real,Image;
                  }
  static void Main()
    {
     int a = 100;
     Complex b = new Complex();
     string s = a.ToString();
     Console.WriteLine("Объект a = " + s);
     s = b.ToString();
     Console.WriteLine("Объект b = " + s);
    }
```

```
}
```

```
Объект a = 100
Объект b = Структуры5+Complex
```

Содержимое строки `s` для объекта `b`, как следует из примера, определяет не содержимое экземпляра класса, а его принадлежность типу.

В следующем примере метод `ToString()` переопределяется для работы с типом `Complex`. Переопределение метода (в отличие от перегрузки) характеризуется следующими особенностями:

- метод может быть переопределён только в производном классе или структуре;
- в базовом классе переопределяемый метод должен иметь модификатор `virtual`;
- переопределяемый метод должен иметь модификатор `override`;
- сигнатуры виртуального и переопределяемого метода должны полностью совпадать.

```
using System;
class Структуры6
{
 struct Complex
      {   double Real,Image;
          public Complex(int inR, int inI)
                         { Real = inR; Image = inI; }
          public override string ToString()
            {
             string s;
             if (Image >= 0) s = Real + "+j" + Image;
                   else s = Real + "-j" + Image*(-1);
                return s;
            }
 }
 static void Main()
    {
     Complex a = new Complex(3,-4);
     Console.WriteLine("Объект a = " + a);

 }
}
```

```
Объект a = 3-j4
```

Операции для пользовательских типов

Для улучшения функциональности пользовательских типов можно разработать (правильнее сказать – перегрузить) достаточно большое количество операций:

- все унарные (+, -, !, ~, ++, --);
- бинарные, кроме логических И(&&) и ИЛИ(||);
- операции true и false для использования в выражениях проверки (условная операция, операторы if else);
- операции приведения типов.

Перегрузка операции для типа осуществляется с помощью специального вида методов – операторного. Его особенности:

- операторный метод должен быть обязательно public static;
- он должен сохранять арность операции;
- один из входных аргументов операторного метода должен быть включающего типа (то есть того самого, для которого перегрузка операции и осуществляется);
- операторный метод не должен (а чаще всего и не может) изменять значение входного аргумента;
- он может возвращать включающий тип.

Следующий пример демонстрирует перегрузку для типа Complex операции ++ и сложения:

```
using System;
class Структуры7
{ struct Complex  {
        double Real,Image;
     public Complex(int inR, int inI)
                    { Real = inR; Image = inI; }
          public override string ToString()
            {
             string s;
             if (Image >= 0) s = Real + "+j" + Image;
                   else s = Real + "-j" + Image*(-1);
                return s;
            }
    public static Complex operator ++ (Complex argin)
     {   Complex argout;
         argout.Real = argin.Real  + 1;
         argout.Image = argin.Image + 1;
         return argout;
```

```
     }
    public static Complex
operator +(Complex arg1,Complex arg2)
     {   Complex argout;
         argout.Real = arg1.Real + arg2.Real;
         argout.Image = arg1.Image + arg2.Image;
         return argout;
     }
 }
 static void Main()
    {
        Complex a = new Complex(1, 2),
b = new Complex(3, 4);
        a++;
        Complex c = a + b;
        string s = c.ToString();
        Console.WriteLine("Объект c = " + s);
    }
}
```

```
Объект c = 5+j7
```

Свойства

Свойством называется средство доступа к закрытым полям типа (для структуры это поля `private`). Похожую функциональность могут обеспечить и методы, но с помощью свойств то же самое нагляднее и проще. По функциональности свойства близки методам, а по использованию – полям.

В структурных типах свойства могут быть объявлены только `public` (в классном типе могут быть и `protected` свойства):

```
public Тип Имя // Имя – это имя свойства
{
 get{// метод доступа на чтение закрытых полей
   }// должен содержать хотя бы один return со значением Тип
 set{// метод для установки значений закрытых полей
     }// имеет одну входную автопеременную value
}
```

Свойства запускаются по принципу автовызова:

```
Имя = значение;  // вызов set – метода,
Объект = Имя; //  вызов get – метода.
```

Методы `get` и `set` могут иметь самостоятельные модификаторы режима защиты, которые должны быть не ниже модификатора самого свойства.

Следующий пример демонстрирует использование свойств в типе `Complex`.

```
using System;
class Свойства // вместо конструктора инициализируем
поля через set свойства
{
 struct Complex
    {
     double Real,Image;
     public double real
   {
  get { return Real;}
//тривиальный вариант get - свойства
  set // set - свойство работает на два фронта!
  {Real = value;
   Image = 2 * value;
// мнимая часть всегда в 2 раза больше!}
  }
  public double image // get свойство у Image - своё
    {
     get { return Image; }
//тривиальный вариант get - свойства
     }
  }
static void Main()
  {
   int re1=10, re2=20;
   Complex a = new Complex(), b = new Complex();
//объявление и вызов конструктора по умолчанию
   a.real = re1;// работает set  - свойство
   b.real = re2; // работает set  - свойство
Console.WriteLine("Комплекс a: {0} +j {1}",
               a.real,a.image);//работают get
Console.WriteLine("Комплекс b: {0} +j {1}",
               b.real,b.image);//работают get
  }
}
```

```
Комплекс a: 10 +j 20
Комплекс b: 20 +j 40
```

В случае когда нет необходимости работы с полями, можно воспользоваться так называемыми *автосвойствами*:

```
using System;
class Свойства
{
 struct Complex
{
 public double real  {get; set;}
 public double image  {get; set;}
 public override string ToString()
  {return String.Format("{0} + j {1}", real, image);}
}
static void Main()
 {
  Complex a = new Complex()
{real = 2.5, image = 10.5};
  Console.WriteLine(a);
 }
}
```

Обратите внимание на то, что инициализация полей объекта выполняется именованным образом.

МЕТОДЫ И АЛГОРИТМЫ ЧИСЛЕННОГО ИНТЕГРИРОВАНИЯ

Постановка задачи

Для некоторых подынтегральных функций интеграл можно вычислить аналитически или найти в справочниках. Однако в общем случае первообразная функции:

- может быть *неопределенной*;
- может не иметь выражения через элементарные функции.

Кроме того, сами подынтегральные функции в отдельных случаях не являются элементарными. В конечном счёте это приводит к необходимости разработки приближенных методов вычисления определенных интегралов. Наиболее простыми среди них являются так называемые *классические методы численного интегрирования*:

- прямоугольников;
- трапеций;
- парабол.

Каждый из этих методов основан на суммировании элементарных площадей, на которые разбивается вся площадь под функцией. Так в методе прямоугольников площадь под графиком функции (а значит, и определенный интеграл от a до b) может быть определен по одной из формул:

а) для входящих прямоугольников

$$\int_a^b f(x)\,dx = \frac{b-a}{n}(y_0 + y_1 + \cdots + y_{n-1});$$

б) для выходящих прямоугольников

$$\int_a^b f(x)\,dx = \frac{b-a}{n}(y_1 + y_2 + \cdots + y_n),$$

где n – кратность (количество шагов) интегрирования функции $y_i = f(x_i)$ в точке, определяющей либо вписанный, либо описанный по отношению к графику интегрируемой функции прямоугольник.

Для уточнения значения интеграла, полученного по формулам суммирования площадей прямоугольников, существует формула остаточного члена

$$H = \frac{(b-a)^3}{24n^2}\xi,$$

где ξ – максимум значения второй производной на рассматриваемом интервале интегрирования, $\xi = \max_{x\in[a,b]}|f''(x)|$.

В методе трапеций площадь криволинейной трапеции и интеграл могут быть вычислены по формуле

$$\int_a^b f(x)\,dx \approx \frac{b-a}{n}\left(\frac{y_0+y_n}{2} + y_1 + y_2 + \cdots + y_{n-1}\right).$$

Формула остаточного члена для метода трапеций имеет вид

$$H = \frac{b-a}{12}\xi\, h^2.$$

В методе парабол (Симпсона) определение площади под графиком интегрируемой функции основано на замене двух смежных фрагментов участком параболы второго порядка (параболической трапецией)

$$\int_a^b f(x)\,dx \approx$$
$$\approx \frac{n}{3}\left(y_0 + y_{2m} + 2(y_2 + y_4 + \cdots + y_{2m-2}) + 4(y_1 + y_3 + \cdots y_{2m-1})\right).$$

Остаточный член для этого метода находится по формуле

$$H = \frac{(b-a)^5}{180n^4}\xi,$$

где ξ – максимум значения четвёртой производной на рассматриваемом интервале, $\xi = \max_{x\in[a,b]}\left|f^{(4)}(x)\right|$.

Пример программной реализации численного интегрирования

Решение задачи предусматривает выполнение нескольких этапов.

Постановка задачи. Рассчитать определенный интеграл приближённым и точным методом, оценить погрешность и вывести результаты на консоль для функции $f(x) = 4x^3 + 3x^2 + 2x$ на интервале [0, 10]. Для приближённого вычисления определённого интеграла использовать метод трапеций с кратностью 1000 без дополнительного члена.

Проектирование типов данных. Для решения поставленной задачи целесообразно использовать структурный тип конструкции, приведенной на рис. 25.

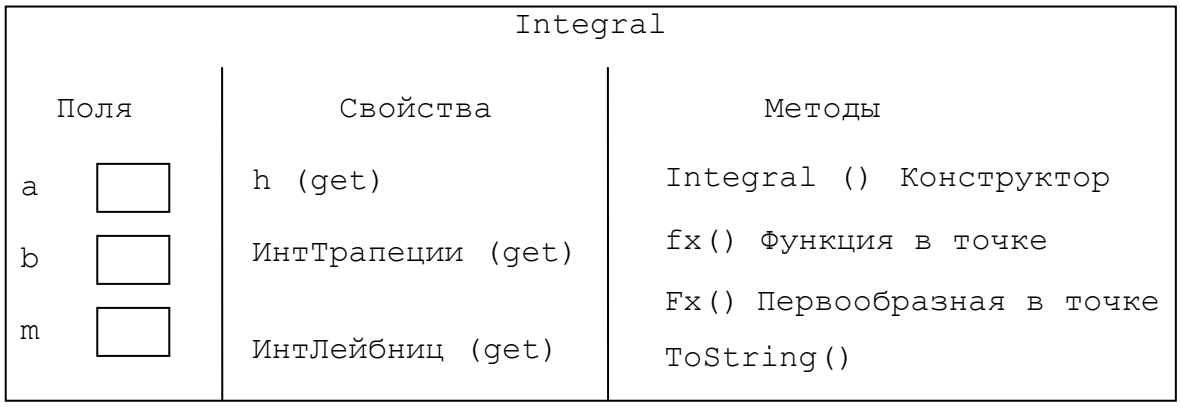


Рис. 25. Схема пользовательского типа

Листинг программы

```
using System;
class Интеграл1
{
 struct Integral
 {
  double a, b;
  int m;
  public Integral(double ina, double inb, int inm)
  {
      a = ina < inb ? ina : inb;
      b = ina < inb ? inb : ina;
      m = inm;
  }
  public override string ToString()
  {
  string s;
s = String.Format("Точное значение=
{0:f3}\nПриближённое значение={1:f3}",
ИнтТрапеции, ИнтЛейбниц);
double Delta = (ИнтТрапеции -
ИнтЛейбниц)/ИнтЛейбниц*100;
Delta = Math.Abs(Delta);
s += String.Format("\nПогрешность = {0:f3} %",Delta);
return s;
  }

public double fx(double x)
  {
```

```
   return 4 * x * x * x + 3 * Math.Pow(x, 2) + 2 * x;
  }
  public double Fx(double x)
  {
      return x * x * x * x +  Math.Pow(x, 3) + x * x;
  }
  public double h
  {
   get
   {
       return (b - a) / m;
   }
  }
  public double ИнтЛейбниц
   {
       get
          {
              return Fx(b) - Fx(a);
          }
      }
  public double ИнтТрапеции
     {
         get
         {
    double sum = (fx(a)+fx(b))/2;
    for (double i = a + h; i < b; i += h)
sum += fx(i);
    sum *= h;
    return sum;
         }
     }
    }
  static void Main()
    {
        Integral obj=new Integral(0,10,1000);
        Console.WriteLine(obj.ToString() );
  }
 }
```

```
Точное значение= 11143,210
Приближённое значение= 11100,000
Погрешность = 0,389 %
```

Пример тестового задания № 1 приведен в прил. 1.

НЕКОТОРЫЕ ВЫВОДЫ

Наиболее существенное из рассмотренного можно свести к следующему.

- Среда .NET – это набор программных технологий для разработки Web- и Windows-приложений. Основная особенность данной среды – *многоязыковость*. В так называемое *решение* могут быть объединены модули на языках Visual Basic, Visual C++, J#, Jscript и других Но наиболее эффективно новые технологии программирования могут быть использованы в модулях на языке C#, который и был разработан специально для .NET**.** Для поддержки многоязыковости трансляция исходных модулей осуществляется в промежуточный IL-код (формируется самодокументированный модуль-сборка с расширением .exe), а преобразование в машинный код исполняет специальный компилятор времени выполнения (JIT) во время загрузки сборки в память.
- Стандартные типы C# (более 4 000) сгруппированы в группы по функциональному признаку, которые называются *пространствами имён*. Пространства имён организованы в единую древовидную структуру, в которой имена классов могут повторяться в разных пространствах, оставаясь при этом уникальными. Вложение одних классов в другие встречается крайне редко.
- Корневым пространством имён является `System`. В частности, оно включает такие пространства, как `Collection`, `Windows`, `Drawing`, `IO` и другие. Кроме того, в пространстве `System` объявлены встроенные типы языка:
  - 12 числовых (по степени возрастания старшинства) – `sbyte`, `byte`, `short`, `ushort`, `char`, `int`, `uint long`, `ulong`, `float`, `double decimal` – все размерные, значащие;
  - один логический тип `bool`, тоже размерный;
  - строковый тип `string` – ссылочный;

Любой тип в C# является прямым или косвенным потомком от *супербазового* типа `System.Object` (или `object`). Это позволяет использовать, например, методы `GetType()`, `ToString()`, методы проверки на эквивалентность и другие элементы, объявленные в классе `object`. Значимые типы являются прямыми потомками

класса System.ValueType. Они определены с помощью конструкции struct и всегда (!) размещаются в стеке. Здесь наследственностью обеспечивается возможность использования, например, полей MaxValue, MinValue.

Встроенные числовые типы могут быть использованы для размещения скалярных данных. Наиболее удобными инструментами их обработки являются операции (порядка 40). Операторы управления и циклов позволяют реализовать соответствующую логику вычислений (if, if else, switch, while, do while, for).

Встроенные (а также пользовательские) типы могут быть организованы в массивы, которые всегда наследуют классу System.Array. Основной выигрыш при работе с массивами даёт использование операции индексирования. Кроме того, наследственность от System.Array расширяет функциональность любого массива:

- на свойства Length, Rank;
- статические методы Reverse(), Sort(), GetLength() и другие;
- возможность использования цикла foreach.

В языке предусмотрена возможность разработки пользовательских типов:

- значимых (с помощью инструкции struct), которые неявно, но прямо наследуются от класса ValueType. Пользовательские структуры не могут быть производными от классов или других структур. При этом могут наследовать интерфейсам (или говорят: поддерживать интерфейсы);
- ссылочных (с помощью инструкции class). Данный тип позволяет использовать всю мощь объектно ориентированной технологии программирования. В вышеизложенном материале оператор class использовался только для организации *обёрточного* класса, содержащего метод Main().

И у структур, и у классов функциональность типу обеспечивают методы, свойства, операции. Классные типы дополнительно могут получать способность реагирования на определённые события.

КЛАССЫ

И классы, и структуры предназначены для объявления типов. При этом использование структур, как правило, ограничено моделированием более простых – геометрических и математических – данных. Понятие класса более глубокое, чем понятие структуры, и является фундаментальным для любых объектно ориентированных языков программирования. Именно классы обеспечивают одну из целей объектно ориентированных технологий – достижение адекватности моделирования предметной среды.

Главное отличие классов от структур состоит в том, что классы между собой могут поддерживать отношения наследования. Разновидностью класса являются специализированные типы: интерфейсы и делегаты. Первые ориентированы на расширение только функциональности разрабатываемого типа (методы и свойства), но при этом позволяют реализовать множественное наследование. На основе делегатов программируется реакция систем на разнообразные внешние воздействия или события.

Объявление класса и создание его экземпляров

Простейший *невложенный* класс (входит только в пространство имён, в какой-либо другой класс не входит) можно объявить следующим образом:

```
class A{},–
```

при этом само объявление по умолчанию становится `internal`, то есть может быть использовано только внутри данной сборки. При указании спецификатора доступа `public` класс становится *видимым* (наблюдаемым и используемым) из других сборок.

При объявлении вложенного класса спецификатором по умолчанию является `private` (режим *по умолчанию* для элементов класса или структуры). Дополнительно для класса, который может стать базовым, для объявления элементов используют спецификатор `protected`. Данное определение обеспечивает непосредственный доступ к `protected` – элементу со стороны методов производного класса.

Использование класса начинается с создания его экземпляров:

```
A o1 = new A();
A o2 = new A();
A o3 = o2;
```

Операция `new` выделяет память (в области кучи – `heap`) и в данном случае вызывает конструктор *по умолчанию*. Для объекта `o3` новой ссылке присваивается адрес существующего объекта, содержащийся в ссылке с именем `o2`. Данный факт подтверждает пример:

```
namespace Class1
{  class A
    {   int a;
        public A(int ia)
        {
            a = ia;
        }
        public override string ToString()
        {
            return String.Format("a={0} ", a++);
        }
      static void Main()
        {
            A o1 = new A(1), o2 = new A(1), o3 = o2;
            Console.WriteLine(o1);
            Console.WriteLine(o2);
            Console.WriteLine(o3);
        }
    }
```

```
a=1
a=1
a=2
```

Результат примера свидетельствует о том, что третий и второй объект – одно и то же.

Объекты классных типов *ссылочные* и сами по себе содержат только ссылку (по сути адрес) в куче, где размещается данный объект.

В следующем фрагменте объявление объектов и их инициализация для наглядности разделены:

```
A o1;
A o2;
A o3;
o1 = new A();
o2 = new A();
o3 = o2;
```

По объявлению конструкторов класса приняты те же правила, что и по конструкторам структур:

- имя должно совпадать с именем класса;
- возвращаемого значения нет;
- конструкторы классов могут быть перегружены, в том числе и конструктор по умолчанию.

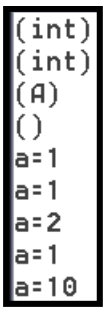
Конструктор класса *по умолчанию* выполняет инициализацию полей объекта с помощью предустановленных значений *по умолчанию*. Для числовых полей это – нуль, а для ссылочных – нуль-адрес, или null. Конструктор *по умолчанию* структур такого рода инициализацию не выполняет. Для полей экземпляров классного типа такого рода *обнуление по умолчанию* выполняется всегда, когда значения полей не определены явно.

Конструктор класса *по умолчанию*, в отличие от конструктора структуры, можно перегрузить, и, более того, при определении в классе хотя бы одного явного конструктора конструктор *по умолчанию* автоматически отключается. При этом класс, как и структура, может иметь неограниченное количество перегруженных конструкторов.

Следующий пример демонстрирует различные варианты конструкторов класса, в том числе перегрузку конструктора по умолчанию:

```
class A
  {
      int a;
      public A(int ia)
      {   a = ia;
          Console.WriteLine("(int)");
      }
      public A()
      {
          a = 10;
          Console.WriteLine("()");
      }
      public A(A ia)
     {
        a = ia.a;
        Console.WriteLine("(A)");
     }
      public override string ToString()
```

```
        {     return String.Format("a={0} ", a++); }
    }
     static void Main()
     {    A o1 = new A(1), o2 = new A(1), o3 = o2,
            o4 = new A(o2), o5 = new A();
            Console.WriteLine(o1);
            Console.WriteLine(o2);
            Console.WriteLine(o3);
            Console.WriteLine(o4);
            Console.WriteLine(o5);
     }
}
```



Таким образом, объекты `o2` и `o3` – это одно и то же, объект `o4` – это отдельный объект; для создания объекта `o3` не требуется вызова конструктора.

Ключевое слово `this`

Первым *скрытым* параметром любого нестатического метода класса (в том числе конструктора) является ссылка `this`, содержимым которой является адрес целевого (текущего) экземпляра. Эта ссылка *привязывает* метод к объекту, для которого метод вызван. В большинстве случаев эта ссылка внутри методов используется неявно, при совпадении же имени поля с именем входного аргумента или локальной переменной её использование достаточно удобно. Вторым способом применения `this` является вызов в конструкторе другого конструктора (того же класса), как правило, с более длинным списком аргументов. В этом случае перед `this` требуется двоеточие:

```
class A// невложенный класс
 {   int a;
     double b;
     string s;
     public A(int a, double b, string s)
     {   this.a = a;//Первое применение this
         this.b = b;
```

```
            this.s = s;
            Console.WriteLine("Конструктор 1");
        }
        public A(string s)
          : this(10, 20.5, s) //Второе применение this
        { Console.WriteLine("Конструктор 2"); }
    public void TypeA()
{ Console.WriteLine("a={0}, b= {1}, s ={2}", a, b, s);}
       static void Main()
        {
            A o1 = new A(1,2.5," Первый объект");
            A o2 = new A(" Второй объект");
            o1.TypeA();
            o2.TypeA();
         }
    }
```

```
Конструктор 1
Конструктор 1
Конструктор 2
a=1, b= 2,5, s = Первый объект
a=10, b= 20,5, s = Второй объект
```

Таким образом, `конструктор2` часть полномочий делегировал первому конструктору. Использование `this` правомерно и для структурных типов.

Деструктор

Несмотря на то что зачисткой памяти в .NET занимается специальная служба – `Garbage Collector` (сборщик мусора), в любом объекте есть деструктор *по умолчанию*. Этот метод вызывается *сборщиком мусора* при выходе потока выполнения за область видимости объекта, когда на объект не остаётся ни одной ссылки. В C# существует возможность определять и явные деструкторы *для классов* (для структур деструкторы недопустимы). Правила создания деструктора следующие:

- имя деструктора совпадает с именем класса с точностью до *тильды* (~);
- деструктор не имеет типа (как и конструктор) и выходного значения;
- деструктор не имеет входных параметров;

- деструктор всегда `public`, вызывается автоматически для удаления объекта из памяти.

В следующем примере показана очередность вызовов конструктора и деструктора.

```
namespace Class2
{
    class A
    {
        string s;
        public A(string s)
        {
            this.s = s;
            Console.WriteLine("Создаю = " + s);
        }
        ~A() { Console.WriteLine("Уничтожаю = " + s);
}
       static void Main()
        {
  A object1 = new A("первый"),
object2 = new A("второй");
        }
    }
}
```

```
Создаю = первый
Создаю = второй
Уничтожаю = второй
Уничтожаю = первый
```

Конкретный момент вызова деструктора определяется системой, наличие в классе деструктора несколько замедляет выполнение программы.

Поля класса

Основной элемент класса – поля. В отличие от полей структур, поля класса могут быть проинициализированы сразу при объявлении (и тогда это выполняется до вызова конструктора):

```
class A
    {
        int a=1;
        double b=2;
        string s="Три";
    }
```

Такое решение позволяет отказаться от *конструктора-инициализатора*, так как для любого экземпляра именно эти константы будут инициализировать поля вместо значений *по умолчанию*.

Поля `const` *и* `readonline`

Так же как и локальные переменные, поля класса могут иметь модификаторы `const` (значения должны быть определены при инициализации) и `readonly`. В последнем случае значения не обязательно задавать при объявлении, но они должны быть непременно определены в конструкторе, так как сразу после создания экземпляра вступает в действие режим `readonly`:

```
namespace Class3
{
    class A
    {
         const int a=100;
         readonly double b;
        public A( double ib)
        {  b = ib+a;}
        public override string ToString()
        { return String.Format("a={0} b= {1}",a,b); }
      }
     static void Main()
        {   A o1 = new A(2.5);
   Console.WriteLine(o1);
// o1.ToString() необязательно
        }
}
```

```
a=100 b= 102,5
```

Значение поля `b` свидетельствует о том, что к моменту вызова конструктора инициализация поля `a` уже состоялась.

Статические элементы класса

В начальных главах пособия упоминание статических элементов было связано, во-первых, с методом `Main` (`static void Main`) и, во-вторых, с операторными методами (они могут быть только статическими). Тем не менее основное предназначение модификатора `static` – это обеспечение *статизма* для полей класса.

В примере со с. 119 экземпляры класса `A` получали свои эксклюзивные наборы полей. Однако в ряде случаев бывает необходимо, чтобы некоторые поля класса были *общими* для всех экземпляров. Для этого элементы объявляют с модификатором `static`. Примером использования статического поля является счётчик количества экземпляров класса:

```
namespace Class4
{
    class A
    {
        int a;
        double b;
        string s;
        static int Counter;//статический
        public A(int ia, double ib, string is)
        {
            a = ia;
            b = ib;
            s = is;
            Counter++;
        }
        public static int GetCounter()
        {  return Counter; }
    }
     static void Main()
        {
Console.WriteLine("Объектов={0}", A.GetCounter());
A o1 = new A(1,2.5," Первый объект");
A o2 = new A(2,3.5," Второй объект");
Console.WriteLine("Объектов={0}", A.GetCounter());
            A o3 = new A(3,4.5," Третий объект");
Console.WriteLine("Объектов={0}", A.GetCounter());
        }
    }
```

```
Объектов=0
Объектов=2
Объектов=3
```

Особые свойства статических полей определяет механизм их реализации (рис. 26). Организационно они являются элементами класса, но фактически размещаются в отдельном сегменте – сегменте данных, определяемом регистром DS. Размещение сегмента данных в оперативной памяти компьютера (а, значит, и статических полей) выполняется в первую очередь, другие сегменты – кода, стека, дина-

мических данных – размещаются позже. Следовательно, к началу работы программы статические поля уже присутствуют в памяти. После завершения программы сегмент данных (включая статические элементы) освобождает память одним из последних.

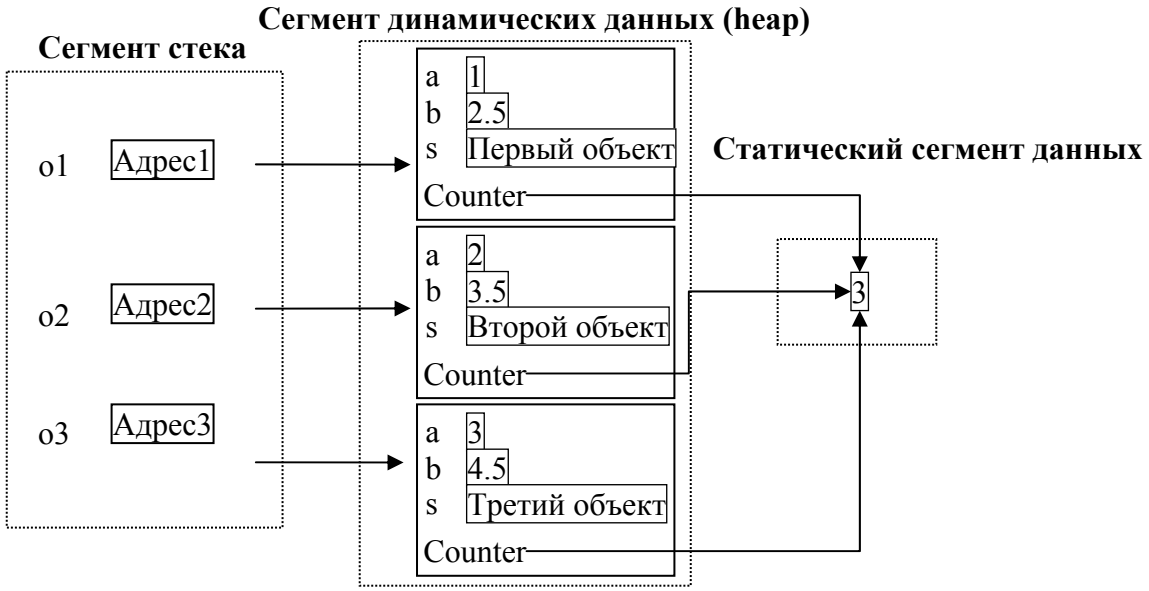


Рис. 26. Схема объектов для примера со с. 121

Доступ к статическому полю может получить и нестатический метод (обратное невозможно: статический метод не может получить доступ к нестатическому полю) (рис. 27). При этом изменяется формат оператора вызова (полное имя поля включает имя класса, а не имя объекта) и исключается возможность использования метода до момента объявления объекта.

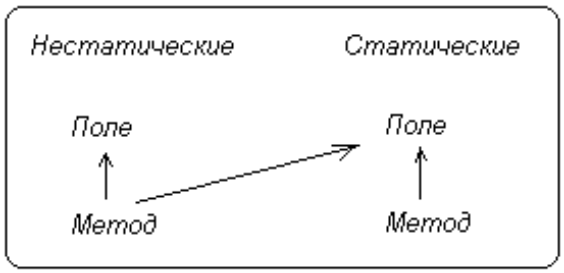


Рис. 27. Возможности доступа нестатических и статических методов

Следующий пример демонстрирует использование явного деструктора, а также нестатического свойства для доступа к статическому полю:

```
class A
 {   static int Counter;
     public A()
        {
            Console.WriteLine("Конструирую объект");
            Counter++;
        }
    ~A() { //Деструктор
            Counter--;
            Console.WriteLine("Уничтожаю объект");
            }
    public int counter{get{return Counter;}}
    static void Main()
     {
      A o1 = new A();
  Console.WriteLine("Объектов = {0}", o1.counter);
      A o2 = new A();
  Console.WriteLine("Объектов = {0}", o2.counter);
       { // вложенный блок
         A o3 = new A();
  Console.WriteLine("Объектов = {0}", o3.counter);
       }
  Console.WriteLine("Объектов = {0}", o1.counter);
     }
 }
```

Вывод в консольное окно (рис. 28) подтверждает следующее:

- статическое поле существует в единственном экземпляре;
- момент запуска деструктора (уничтожение объекта) выбирает не программист, а сборщик мусора.

Для статических полей может быть предусмотрен статический конструктор (единственный, без перегрузки): без аргументов и даже без спецификатора `public`. Он вызывается автоматически до создания первого экземпляра типа или до первого обращения к одному из статических полей.

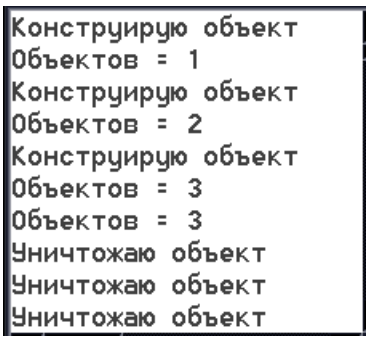


Рис. 28. Результат
выполнения примера

```
class A
 {
    static int Counter;
    public A()
    { Counter--;
    }
    static A() //Статический конструктор
    { Counter = 10; }
    public static int counter
    {
       get { return Counter; }
    }
    static void Main()
     {
      Console.WriteLine("Объектов={0}", A.counter);
      A[] o1 =
{ new A(), new A(), new A(), new A(), new A() };
      A o2 = new A();
      Console.WriteLine("Объектов={0}", A.counter);
     }
 }
```

```
Объектов=10
Объектов=4
```

Статические поля часто называют *полями класса*, в отличие от нестатических, которые называют *полями экземпляра* (или экземплярными). То же относится и к методам.

Если класс содержит только статические элементы, то его можно также объявить статическим. При этом его экземпляры с помощью оператора `new` создавать нельзя!

Если количество полей типа зависит от количества объектов типа и статизма их полей, то любые методы типа в любом случае присутствуют в памяти в единственном числе и находятся вне фрагмента

экземпляра класса (и статические и нестатические методы). Действительно, методы представляют собой программный код, одинаковый для всех экземпляров, и при использовании (в результате вызова метода) этот код каким-либо образом измениться не может. Отсюда и нет необходимости дублировать содержимое методов в каждом объекте соответствующего типа.

Именно для того чтобы привязать нестатические методы к экземпляру класса, первой в список аргументов нестатического метода добавляется ссылка `this`. Статические методы не имеют скрытого параметра `this`, так как они вызываются от имени класса и их не нужно «привязывать» к экземпляру.

Неявно статическими являются поля класса с модификатором `const`. Очевидна целесообразность такого решения для экономии памяти: значение константного поля изменено быть не может, а значит, и незачем держать в памяти несколько одинаковых значений. Также неявно статическими являются вложенные типы.

В следующих примерах показано отличие в использовании обычных и статических полей класса, а также массива `m` пользовательского типа `A`:

```
class A
{       int Fa;
        static int Fb=0;
        public A(int a) { Fa = a-4; Fb++; }
        public A(int a, int b) { Fa = a - b; Fb+=2; }
        public override string ToString()
        {  return String.Format("{0}",Fa - Fb); }
       static void Main()
 { A[] m ={ new A(6), new A(4, 1), new A(3), new A(3,
2) };
   foreach (A i in m) Console.Write(i);
 }
```

```
-4-3-7-5
```

```
class A
{       int Fa;
        static int Fb=0;
        public A(int a) { Fa = a-4; Fb++; }
        public A(int a, int b) { Fa = a - b; Fb-=2; }
        public override string ToString()
        {  return String.Format("{0}",Fa - Fb); }
        static void Main()
 { A[] m =
{ new A(5), new A(4, 1), new A(3), new A(3, 2) };
```

```
  foreach (A i in m) Console.Write(i);
 }
}
```

3513

```
class A
{     int Fa;
      static int Fb=3;
      public A(int a) { Fa = a-4; Fb++; }
      public A(int a, int b) { Fa = a - b; Fb-=2; }
      public override string ToString()
      {  return String.Format("{0}",Fa + Fb); }
      static void Main()
 { A[] m =
{ new A(4), new A(4, 1), new A(3), new A(3, 2) };
   foreach (A i in m) Console.Write(i);
 }
}
```

1402

Индексаторы

Как и для структур, для классов можно определять свойства, в том числе статические для статических полей. Следующий листинг – пример статического свойства:

```
class A
 {
        static int Counter;//Имя поля - с заглавной!
        public A()
        {
          Counter++;
        }
    public static int counter//Имя свойства - со
строчной!
        {
            get { return Counter; }
        }
  static void Main()
  {
  Console.WriteLine("Объектов={0}", A.counter);
  A[] o1 =
{ new A(), new A(), new A(), new A(), new A() };
  A o2 = new A();
  Console.WriteLine("Объектов={0}", A.counter);
  }
 }
```

```
Объектов=0
Объектов=6
```

Индексатор – это разновидность свойства, с помощью которого для объекта пользовательского *классного* типа можно перегрузить операцию *квадратные скобки*. В стандартном случае пользовательский тип содержит набор элементов, доступ к каждому из которых оказывается возможным по индексу.

В следующем примере класс `IndArray` объявлен для размещения массива целочисленных элементов. Количество элементов в массиве задаётся при вызове конструктора и сохраняется в поле `Len`.

```
class IndArray
        {
           int[] arr ;
           int Len;
           public IndArray(int len)
           {
               arr = new int[Len=len];
           }
           public int this[int ind]
            {
     set { if (ind < Len)arr[ind] = value; }
get { if (ind < Len)return arr[ind]; else return 0; }
            }
   static void Main()
   {
    Random Gen = new Random();
    IndArray mass = new IndArray(2);
    for (int i = 0; i < 4; i++)
    {
     mass[i] = Gen.Next(1,10);
    Console.WriteLine("mass[{0}] = {1} ",i, mass[i]);
    }
   }
 }
```

```
mass[0] = 7
mass[1] = 2
mass[2] = 0
mass[3] = 0
```

Элементов с индексами `2` и `3` в данном случае просто нет, и поэтому индексатор игнорирует обращение к ним. В отсутствие индексатора при выходе индекса за границы массива исключение `System.IndexOutOfRangeException` было бы выброшено с выда-

чей на экран предупредительного сообщения и последующей выгрузкой программы из памяти.

Операция *квадратные скобки* может обеспечивать доступ к элементам и многомерных массивов. В следующем примере индексатор кроме доступа к внутреннему двумерному массиву обслуживает обращения, выходящие за его границы:

```
class M
    {
        int S;
        int[,] Arr;
        int R,C;
        public M(int d)
        {
         Arr = new int[ R = d, C = d];
        }
        public int this[int i, int j]
        {
set{if (i < R && j < C )Arr[i,j]=value;else S += value; }
get { if (i < R && j < C) return Arr[i,j]; else return S; }
        }
  static void Main()
     {
         M mass = new M(3);
         for (int i = 0; i < 4; i++)
for (int j = 0; j < 4; j++) mass[i, j] = i + j;
for (int i = 0; i < 4; Console.WriteLine(), i++)
for(int j=0; j<4; j++)Console.Write("{0:d2} ",mass[i,j]);
    }
 }
```

```
00 01 02 30
01 02 03 30
02 03 04 30
30 30 30 30
```

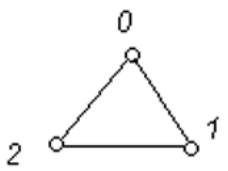
При выходе одного из индексов за реальные границы матрицы (границы – это значения полей `R` и `C` в классе `M`) индексатор выполняет обращение к полю `S` в том же классе.

В примере `матрица сопротивлений` (см. рис. 29) индексатор разработан для заполнения этой матрицы (для простоты сопротивления считаются целочисленными, а нумерация узлов начинается с нуля).

```
class IndArray //Пример матрица сопротивлений
```

```
    {   int[,] arr;
        int Len;
        public IndArray(int len)
        {
            arr = new int[Len = len, len];
        }
        public int this[int row, int col]
        {
         set { if (row < Len && col < Len)
                  arr[row, col] = arr[col, row] = value; }
         get { if (row < Len && col < Len)
                   return arr[row, col]; else return 0; }
        }
   static void Main()
    {
      string s;
      int  Len  =  ReadLine("Задайте  количество  узлов
схемы");
      int n, k, r;
      IndArray R = new IndArray(Len);
      Console.WriteLine("Задайте ветви схемы");
      for (; ; )
      {
          n = ReadLine("Узел  начала ветви");
          k = ReadLine("Узел  окончания ветви");
          if (n + k == 0) break;//окончание ввода
          r = ReadLine("Сопротивление ветви");
          R[n, k] = r;
      }
for (int i = 0; i < Len; i++, Console.WriteLine())
for (int j=0; j<Len; j++)
Console.Write(" {0}", R[i, j]);
    }
        static int ReadLine(string s)
          {
            Console.WriteLine(s);
            s = Console.ReadLine();
            return Convert.ToInt32(s);
          }
    }
```

а



б

```
Задайте количество узлов схемы
3
Задайте ветви схемы
Узел  начала ветви
0
Узел  окончания ветви
1
Сопротивление ветви
1
Узел  начала ветви
1
Узел  окончания ветви
2
Сопротивление ветви
2
Узел  начала ветви
2
Узел  окончания ветви
0
Сопротивление ветви
3
Узел  начала ветви
0
Узел  окончания ветви
0
 0 1 3
 1 0 2
 3 2 0
```

Рис. 29. Пример `Матрица сопротивлений`:
а – схема; *б* – результат в консольном окне

Механизмы наследования

Разработка программы начинается с выделения из окружающего мира так называемой предметной области – набора объектов, для которых разрабатывается программа, или, как говорят, выполняется компьютерное моделирование. Примером предметной среды может служить группа студентов факультета (в программе расчета успеваемость), условные изображения электрических элементов (в программе отображение схем) и другое. После локализации (выделения) предметной области в ней выполняется классификация объектов для выделения общих свойств, параметров, действий. В результате формируется таксономия – структура в виде дерева, которая состоит из классов, связанных отношениями наследования.

Для изображений электрических элементов (предметная область) фрагментом таксономии может служить система классов, приведенная на рис. 30.

Нижний уровень системы представляют классы, описывающие объекты предметной области (для них и разрабатывается программа). Следующий уровень в данном случае можно назвать уровнем примитивов: здесь представлены классы, описывающие графические изоб-

ражения простейших геометрических фигур. Замыкает систему супербазовый класс `Место`, который представляет собой набор характеристик, присущих любому графическому изображению.

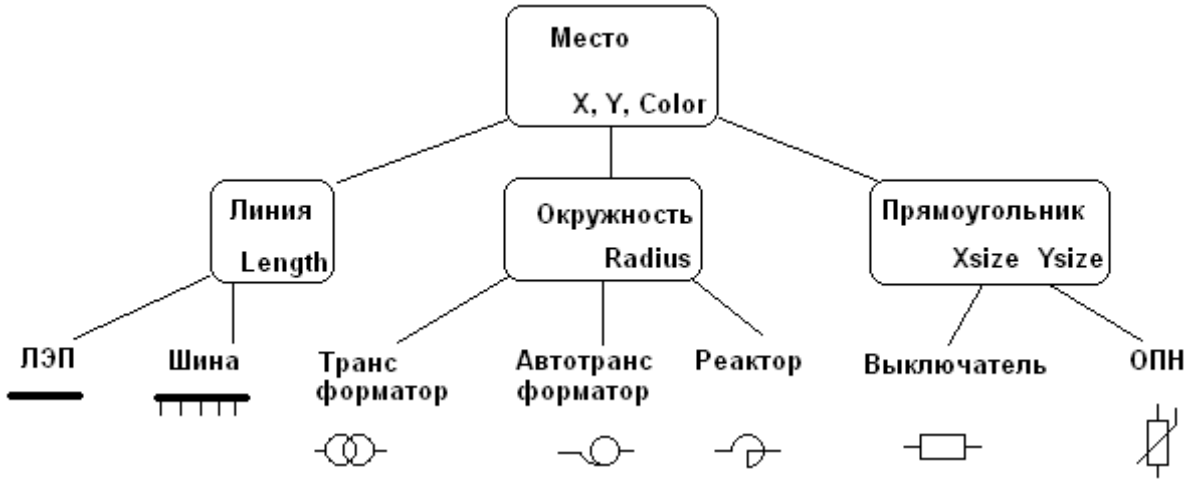


Рис. 30. Система классов:
`X`, `Y` – координаты точки привязки изображения; `Color` – её цвет;
`Length` – длина линии; `Radius` – радиус окружности; `Xsize` и `Ysize` – линейные размеры прямоугольника на соответствующей оси координат

Как видно из таксономии, общие характеристики выносятся в базовые классы, индивидуальные конкретизируют соответствующий объект. Таким образом, класс – это способ моделирования реального объекта предметной области или абстрактного объекта системы классов.

Для объявления классов в общем случае повторяют схему классов (таксономия предметной области). Связями в ней служат отношения наследования. При этом выделяется пара классов: базовый и производный.

Производный класс наследует (получает в распоряжение) все элементы базового класса, кроме конструкторов и деструктора. Производный класс имеет собственные особые методы. Отношения наследования обычно показывает стрелка, направленная от базового класса к производному.

Режимы защиты элементов класса устанавливают спецификаторы доступа:

- `public` – общедоступный элемент класса (не распространяется режим защиты). Данные элементы доступны в любом месте области видимости объекта класса;

- `protected` – защищённый элемент. Данные элементы доступны только в самом классе (собственным элементам) и в производном классе. Спецификатора `protected` в структурах не было, так как структурные типы не поддерживают наследования;
- `private` – частный элемент. Данные элементы доступны только в самом классе.

Отношения наследования формируют таксономию (систему классов) программной системы. Каждый раз наследование связывает производный класс и единственный базовый (простое наследование). Базовый класс потенциально может иметь неограниченное количество производных (рис. 31).

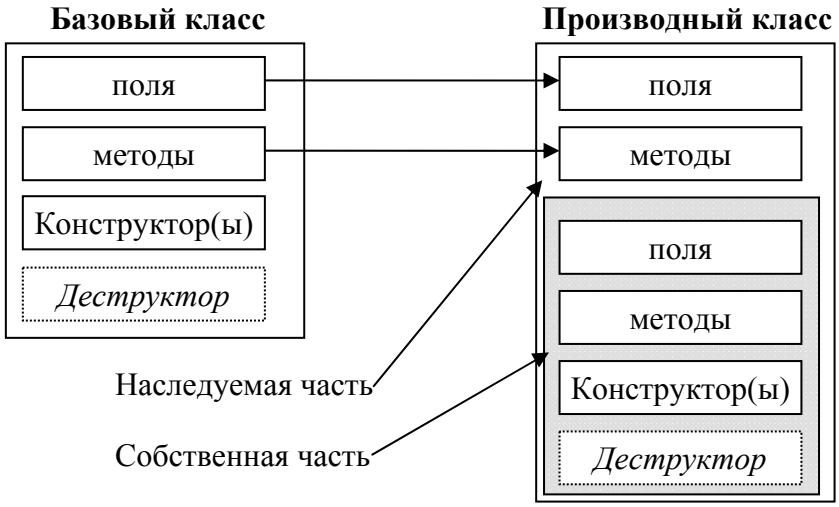


Рис. 31. Схема наследования

Объявление наследования

Наследование всегда объявляется со стороны производного класса, поэтому к моменту объявления производного базовый класс должен быть уже объявлен.

Оператор объявления производного класса имеет следующий состав (рис. 32).

```
class ИмяПроизводногоКласса : ИмяБазовогоКласса
{ < тег производного класса>…};
```

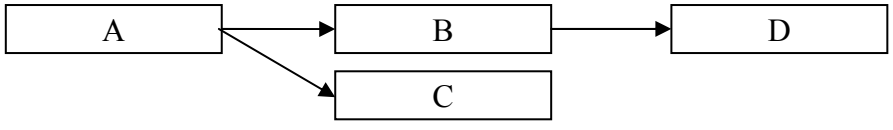


Рис. 32. Пример наследования в системе из четырёх классов

Пример объявления наследования для системы классов рис. 32:

```
class A {элементы класса A};
class B : A {элементы класса B};
class C : A {элементы класса C};
class D : B {элементы класса D};
```

Конструкторы производных и базовых классов

Целью наследования является получение производным классом в свой состав элементов базового класса. Реализуется эта возможность в момент создания объекта (экземпляра) производного класса. Получается, что и здесь необходима деятельность конструктора базового класса: именно конструктор выполняет *материализацию* типа. Очевидно, что при уничтожении объекта производного класса понадобится освобождение памяти, но это выполняет сборщик мусора (GC).

Вызов конструктора базового классов может быть осуществлен как явно, так и автоматически. В следующем примере происходит автовызов конструктора базового класса (рис. 33).

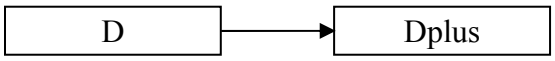


Рис. 33. Схема наследования для следующего примера

```
namespace Class7
    {
        class D
        {
            public D() {Console.WriteLine("Создаю D");}
        }
        class Dplus : D
        {
        public Dplus() { Console.WriteLine("Создаю Dplus"); }
         static void Main()
          {
            Dplus obj = new Dplus();
          }
        }
    }
```

```
Создаю D
Создаю Dplus
```

Вывод на консоль свидетельствует об осуществлённом автовызове конструктора базового класса D для создания наследуемой части

объекта obj. Метод Main можно размещать и в производных классах. Приведённая схема наследования для этого примера (рис. 33) не вполне точна. В языке C# любой тип является прямым или непрямым наследником от супербазового класса object, что легко проверить, если выполнить небольшие изменения в методе Main предыдущего примера:

```
static void Main()
{
Dplus obj = new Dplus();
Console.WriteLine(obj.ToString());//значит ToString
у obj есть!
Console.WriteLine(obj.GetType());// и GetType тоже!
}…
```

```
Соэдаю D
Соэдаю B
Class7.Dplus
Class7.Dplus
```

Методов в классе object всего семь, но наиболее часто используются четыре: Equals, Finalize, GetHashCode, ToString. Поскольку эти методы имеются в любом объекте, постольку обычно их на схемах наследования не показывают, как и само наследование от класса object (рис. 34).

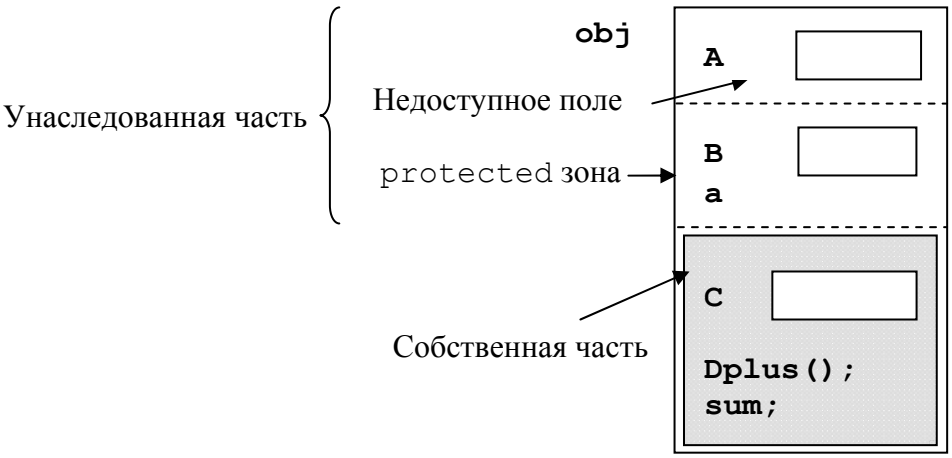


Рис. 34. Схема объекта obj

Дополнительные возможности предоставляет явный (непосредственный) вызов конструкторов базового класса (если они перегружены; если не перегружены, то смысла вызывать нет – автовызов

будет в любом случае). В отличие от C++, в языке C# перед вызовом собственного конструктора в производном классе может быть вызван конструктор только непосредственного базового класса (с помощью встроенного имени `base`).

Рассмотрим схему наследования, также состоящую из двух классов `D` и `Dplus`. В отличие от предыдущего примера (см. последний на с. 134), базовый класс имеет перегруженный конструктор, который явно вызывается перед началом действия собственного конструктора класса `Dplus`:

```
namespace Class8
{
class D
{
  int A;
  protected double B;
  protected int a//защищённое свойство для закрытого
поля A
        { get { return  A;}
        }
  public D(int a, double b) { A = a; B = b;}
    }
class Dplus : D
   {
      double C ;
      public Dplus(int a, double b)
        :base(a+1, b+2)//фактически – это D(a+1, b+2)
        {
          C = b + 3;
        }
        public double sum
        {
            get { return a + B + C; }
        }
    }
  class Program
    {
        static void Main()
        {
            Dplus obj = new Dplus(1,2);
            Console.WriteLine(obj.sum);
        }
    }
}
```

```
11
```

Кроме вызова конструктора базового класса, ключевое слово `base` может быть использовано для доступа к скрытым или переопределённым элементам (методы, свойства) базового класса.

Следующий пример свидетельствует о том, что однажды назначенный режим защиты элемента остаётся неизменным во всех производных классах вне зависимости от количества цепочек наследования:

```
namespace Class9
{
    class D
    {
        int A;
        protected int a
            { get { return  A; }   }
        public D(int a) { A = a;}
    }
    class Dplus : D
    {
        public Dplus(int i) : base(i + 1){ }
    }
    class Dplusplus : Dplus
    {
      public  Dplusplus(int i) : base(i+2){}
        public double sum {  get { return a ; }  }
    static void Main()
        {
            Dplusplus obj = new Dplusplus(1);
            Console.WriteLine(obj.sum);
        }
    }
 }
```

В консольном окне результат 4. В данном случае класс `Dplus` выполнил роль своего рода посредника для передачи поля `A` и свойства `a` от класса `D` классу `Dplusplus`.

Статические элементы в производных классах

Правило единственности статических полей сохраняется и при использовании наследования. В следующем примере статическое поле, объявленное в базовом классе `D`, остается таковым и для производного класса `Dplus`:

```
namespace Class9_1
{
    class D
    {
        protected static int Counter;
        public  D() {Counter++;}
   static public int GetCounter() { return Counter; }
    }
    class Dplus : D
    {
        public Dplus() { Counter++;  }
    }
  class Program
    {
        static void Main()
        {
            D obD = new D();
       Console.WriteLine("1 :{0}", D.GetCounter());
            Dplus obDp = new Dplus();
       Console.WriteLine("2 :{0}", D.GetCounter());
            D obD1 = new D();
       Console.WriteLine("3 :{0}", D.GetCounter());
            Dplus obDp1 = new Dplus();
       Console.WriteLine("4 :{0}", D.GetCounter());
        }
    }
}
```

```
1 :1
2 :3
3 :4
4 :6
```

Результаты в консольном окне свидетельствуют о том, что поле `Counter` общее для всех четырёх объектов.

Скрытие наследуемых элементов

В некоторых ситуациях в производном классе необходимо заменить наследуемый элемент собственной реализацией. Сделать это можно и с полем, и с методом.

Замещение может быть выполнено одним из двух способов:

- с помощью модификатора `new` (в данном случае ключевое слово `new`, ранее известное как операция, выступает в роли модификатора). Это максимально простой способ замещения наследуемого

элемента и далее будет называться *скрытием* (или *сокрытием*). В данном случае не важно, что скрывается – метод, свойство или поле. Не имеют значения какие-либо характеристики скрываемого элемента;

- с помощью модификатора `override` (дословно *отменять, аннулировать*). Называется он переопределением и связан с целым набором условий. Во-первых, замещающий и заменяемый элементы должны быть одного типа. Во-вторых, если они – методы или свойства, то должны совпадать сигнатуры. И, наконец, в-третьих, в базовом классе такая замена должна быть разрешена модификатором `virtual` (*возможный, потенциальный*). На операции *переопределения* наследуемых элементов основан механизм полиморфизма.

В следующем примере демонстрируется сокрытие поля и свойства. Здесь в производном классе содержатся элементы, на первый взгляд, дублирующие и поле (`A`), и свойство (`a`):

```
namespace Class10
{   class D {  int A = 1;
                  protected int a { get { return A; } }
            }
    class Dplus : D
            {
            int   A = 10;
            public int a { get { return A; } }
        static void Main()
        {
            Dplus obj = new Dplus();
            Console.WriteLine("{0}", obj.a);
        }
    }
}
```

Трансляция примера проходит с предупреждением, при запуске на выполнение вполне ожидаемо выводится число 10 на консоль. Транслятор предупреждает следующим образом:

'Class10.Dplus.a' hides inherited member 'Class10.D.a'. Use the new keyword if hiding was intended (в переводе: 'Class10.Dplus.a' скрывает унаследованный элемент 'Class10.D.a'. Используйте ключевое слово **new**, если сокрытие было необходимым)

Представляют интерес ответы на следующие вопросы:

- почему транслятор предупреждает про сокрытие свойства `a` и ничего не говорит про сокрытие поля `A`;
- что даёт использование `new`, рекомендуемое транслятором;
- что происходит с унаследованными элементами, если их скрывают собственные элементы?

Начнём с последнего вопроса. После небольшой доработки примера, связанной с изменением свойства `a` производного класса,

```
public int a { get { return A + base.a; } }
```

в окне вывода получаем число `11`, что свидетельствует о *присутствии* в экземпляре производного класса унаследованных элементов, которые были скрыты собственными.

Чтобы ответить на первый вопрос, обратимся к схеме объекта (рис. 35).

Транслятор ничего не говорит про сокрытие наследуемого поля `A`, так как оно недоступно, в том числе и в производном классе. Если перед полем `A` поместить, например, модификатор `protected`, получим аналогичное предупреждение на его счёт.

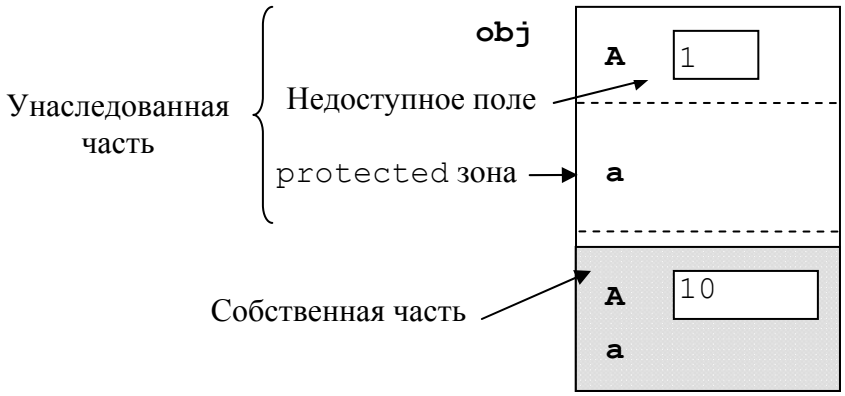


Рис. 35. Схема объекта для примера `Class10`

Теперь добавим `new` в строку, в которой объявляется свойство `a` в производном классе.

```
new public int a { get { return A + base.a; } }
```

Трансляция после этого проходит без предупреждений, и число `11` на экране свидетельствует о том, что ничего не изменилось. Вывод: транслятору важно убедиться, что сокрытие было сделано осознанно.

Обращение с помощью `base` к сокрытым элементам базового класса допустимо только на одной цепи наследования, например, конструкция `base.base.Имя` уже недопустима.

Следующий пример демонстрирует перекрытие наследуемого метода:

```
namespace Class10.1
{
    class D
    {
 public string GetName() {return "Класс D";}
//virtual нет!!
    }
    class Dplus : D
    {
 public new string GetName() {return "Класс Dplus";}
        public string GetName2()
        {
            return base.GetName() + "->" + GetName();
        }

    static void Main()
        {
            Dplus obj = new Dplus();
            Console.WriteLine(obj.GetName());
            Console.WriteLine(obj.GetName2());
        }
    }
}
```

```
Класс Dplus
Класс D->Класс Dplus
```

Перекрытие в классе `Dplus` метода `GetName` осуществляется, хотя базовый класс `D` непосредственно этого не разрешал (нет модификатора `virtual`). В нашем примере тип и сигнатура перекрывающего метода совпадают с аналогичными параметрами оригинального метода, в общем случае они могут быть различными.

Пример на перекрытие поля:

```
namespace Classy10.2
    {
        class D
    {       protected  int A = 1;
            public int GetA() { return A;}
```

```
        }
    class Dplus : D
            {
 public new float A = 10.5f;
 public new  float GetA() { return A + base.GetA(); }
      static void Main()
        {
            Dplus obj = new Dplus();
            Console.WriteLine("{0}", obj.GetA());
            Console.WriteLine("{0}", obj.A);
        }
    }
}
```

```
11,5
10,5
```

Разница в значениях полученного результата косвенно свидетельствует о наличии перекрытого поля, так как напрямую к нему обратиться невозможно. Тип перекрывающего поля также может быть произвольным.

Следующий пример представляет наследование, свойства, конструкции `base` и `this`, а также скрытие наследуемых элементов.

```
namespace class10_3
{   class A
  { protected int Fa;
    protected A(int a){Fa=a;}
    protected A(): this(1){Fa++;}
    protected int fa
                  {
                    get { return Fa--; }
                    set { Fa -= value; }
                  }
  }
    class B : A
    { int Fb;
      public B() : base(3) { Fb = 2 * Fa; }
      public B(int b) { Fb = 2 * b; }
      public new int fa {
                        get{ return Fb + base.fa;   }
                        set{ base.fa = Fb -= value; }
                        }
     static void Main()
        {
            B b1 = new B(),
```

```
                b2= new B(b1.fa),
                    b3= new B ( b2.fa = 3 ) ;
            Console.WriteLine("{0} ", b1.fa);
            Console.WriteLine("{0} ", b2.fa);
            Console.WriteLine("{0} ", b3.fa);
            Console.WriteLine("{0} ", b2.fa);
            Console.WriteLine("{0} ", b1.fa);
        }
    }
}
```

```
8
2
8
1
7
```

Схема объектов с изменениями по ходу выполнения программы показана на рис. 36.

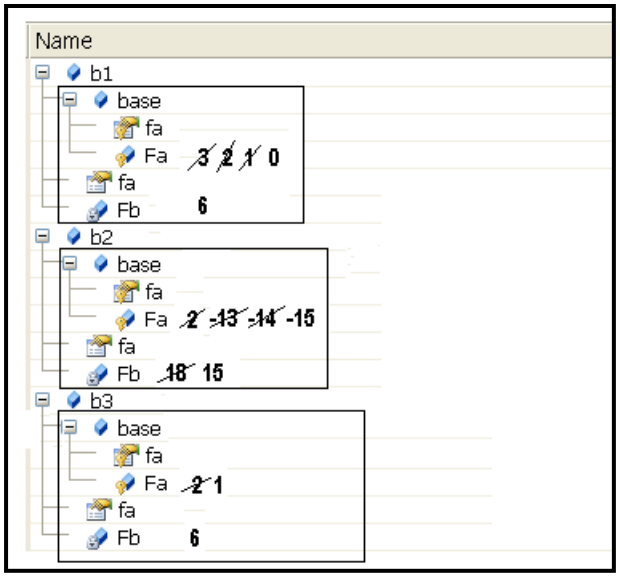


Рис. 36. Схема объектов примера

Переопределение наследуемых элементов

Переопределение методов базового класса реализует полиморфизм в границах таксономии типа. Зачем это нужно, будет обсуждаться далее, а сейчас – о том, *как* это сделать.

Любой производный класс может *переопределить* метод или свойство базового класса, если в базовом они были объявлены с модификатором `virtual`. В производном классе переопределение

должно начинаться с модификатора `override` и абсолютно точно соответствовать типу и сигнатуре заменяемого метода.

Пусть у класса `D` останется только метод, возвращающий его имя, а класс `Dplus` его переопределит:

```
namespace Class11
{ class D
{public  virtual string GetName() {return "Класс D";}
     }
  class Dplus : D
{public override string GetName() {return "Класс Dplus";}
   static void Main()
      { Dplus obj = new Dplus();
        Console.WriteLine(obj.GetName());
      }
    } }
```

В окне фраза `Класс Dplus`.

И после переопределения метод *базового* класса остаётся в составе производного, но доступ к нему должен быть уточнён с помощью конструкции `base`:

```
namespace Class12
{
    class D
    {
 public  virtual string GetName() {return "Класс D";}
    }
    class Dplus : D
    {
public  override  string  GetName()  {return  "Класс
Dplus";}
        public string GetName2()//ещё один метод
        {
return base.GetName()+"->"+GetName();
//переопределённый метод
        }
      static void Main()
        {
            Dplus obj = new Dplus();
            Console.WriteLine(obj.GetName());
            Console.WriteLine(obj.GetName2());
        }
    }
}
```

```
Класс Dplus
Класс D->Класс Dplus
```

Обратим внимание на то, что `base` можно использовать только из *экземплярных* методов, напрямую – от имени объекта – нельзя!

Использование в операторе переопределения метода модификатора `sealed` запрещает переопределение данного метода вниз по иерархии наследования. Объявление класса с данным модификатором запрещает ему иметь производные.

Полиморфное поведение производных классов

Более глубокое использование полиморфизма в границах типа (способность изменять или адаптировать поведение в зависимости от некоторых условий) обеспечивает использование виртуальных методов. Слово «виртуальный» образовано от латинского *virtus* − потенциальный, возможный. Модификатором `virtual` отмечаются методы, которые потенциально могут быть переопределены в производных классах. Таким образом, правильнее говорить *о системе* виртуальных функций, начало которой находится в корневом классе некоторой иерархии типов.

Любой класс, который расположен вниз по иерархии от базового класса, имеющего метод с модификатором `virtual` (в режиме `public` или `protected`), может поступить с этим методом следующим образом:

- ничего не делать и тем самым унаследовать базовую реализацию;
- переопределить метод, используя модификатор `override`;
- перекрыть метод, используя модификатор `new`.

Опыт использования этого механизма уже был (см. пример со с. 116). В *супербазовом* классе `object` имеются виртуальные общедоступные методы `GetType()` и `ToString()`, которые могут быть переопределены в любом пользовательском типе. Важно отметить, что их можно и *не переопределять*, и тогда они возвращают строку, содержащую полное имя типа. Говорят, что в этом случае виртуальные методы сохраняют функциональность базового класса.

Простой пример полиморфизма (рис. 37) разработаем для несколько усечённой таксономии типов графических изображений (для простоты пока без `Color`).

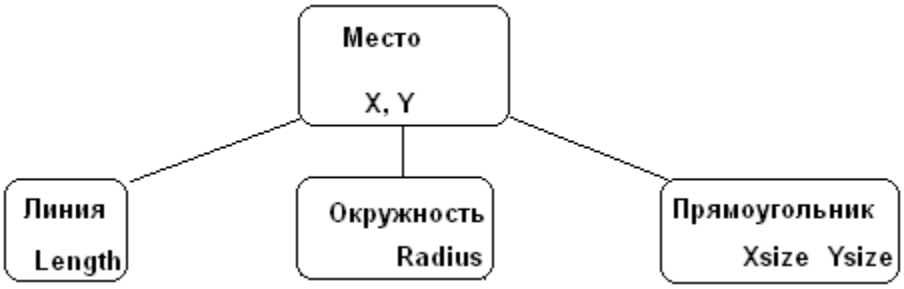


Рис. 37. Сокращённая таксономия

Добавим в класс `Место` метод, возвращающий строку с названием типа объекта и значений его характеристик. Если набор характеристик ограничить полями корневого класса, то не нужно никакой системы виртуальных функций:

```
namespace Элементы1
{
    class Место
    {
        int X,Y;
        public Место(int x,int y){X=x; Y=y;}
        public string type() { return String.Format
                    ("Место: X={0}, Y={1}", X, Y); }
     }
    class Линия : Место
    {
        int Length;
        public Линия(int x, int y, int length) :
                base(x, y) { Length = length; }
    }
    class Окружность : Место
    {
        int Radius;
        public Окружность(int x, int y, int radius) :
               base(x, y) { Radius = radius; }
    }
    class Прямоугольник : Место
    {
        int Xsize, Ysize;
 public Прямоугольник
(int x, int y, int xsize, int ysize) :
          base(x, y) { Xsize = xsize; Ysize = ysize;}
        static void Main()
        {
         Место[] mass = {new Место(10,10),
         new Линия(15,15,15),
```

```
            new Окружность(20,20,20),
            new Прямоугольник(25,25,25,25)};
foreach(Место i in mass)Console.WriteLine(i.type());
                          }
    }
}
```

```
Место: X=10, Y=10
Место: X=15, Y=15
Место: X=20, Y=20
Место: X=25, Y=25
```

Вывод на консоль подтверждает наследование `type` всеми производными от `Место` классами, но сам по себе малоинформативный и где-то даже неправильный (строчки со второй по четвёртую).

В рассмотренном примере используется ещё один аспект *отношений* между базовым и производным классом: тип производного класса всегда и неявно может быть преобразован в тип базового класса, в том числе и в тип любого непрямого базового класса. В данном примере это преобразование выполняется каждый раз при сохранении адреса конкретного объекта в массиве `mass` ссылок на класс `Место`. Массив `mass` можно объявить даже типом `object`

```
object[] mass = ….
```

При этом ничего не изменится. Однако подобная операция с `foreach` не пройдёт: итератор `i` должен иметь тип `Место`.

Усложним задачу: пусть метод `type` возвращает название типа объекта и его эксклюзивные характеристики. Для этого прежде всего метод `type` в корневом классе необходимо сделать виртуальным. Затем с модификатором `override` в производных классах определить новый вариант этого метода:

```
namespace Элементы2
{
    class Место
    {
      int X,Y;
      public Место(int x,int y){X=x; Y=y;}
      virtual public string type()
{ return String.Format("Место: X={0}, Y={1}", X, Y);}
     }
  class Линия : Место
    {
     int Length;
     public Линия(int x, int y, int length)
```

```
      : base(x, y) { Length = length; }
      override public string type()
{return String.Format("Линия: Length= {0}", Length);}
    }
  class Окружность : Место
    {
        int Radius;
  public Окружность(int x, int y, int radius)
   : base(x, y) { Radius = radius; }
  override public string type()
{return String.Format("Окружность: Radius= {0}",
Radius);}
    }
  class Прямоугольник : Место
    {
        int Xsize, Ysize;
public Прямоугольник
(int x, int y, int xsize, int ysize)
        : base(x, y) { Xsize = xsize; Ysize = ysize;}
        static void Main()
        {
          Место[] mass = {new Место(10,10),
          new Линия(15,15,15),
          new Окружность(20,20,20),
          new Прямоугольник(25,25,25,25)};
   foreach (Место i in mass)
Console.WriteLine(i.type());
        }
    }
}
```

```
Место: X=10, Y=10
Линия: Length= 15
Окружность: Radius= 20
Место: X=25, Y=25
```

Поскольку в данном примере не переопределён метод `type` в классе `Прямоугольник`, постольку для него осталась реализация базового класса (последняя строка окна).

Поясним отличия в переопределении (`override`) и сокрытии (`new`) базовой реализации виртуального метода относительно полиморфного поведения типа.

Добавим в систему ещё один класс – `Трансформатор` – сразу с переопределением метода `type` (рис. 38).

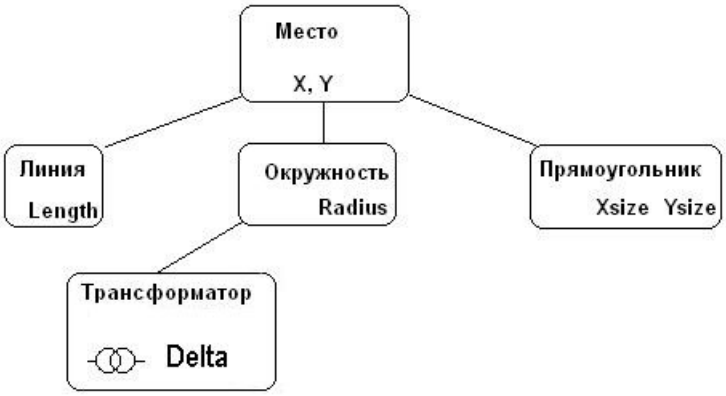


Рис. 38. Система классов после добавления
класса Трансформатор

```
class Трансформатор : Окружность
 {
  int Delta;
public Трансформатор(int x, int y, int radius, int delta)
             : base(x, y, radius)
        { Delta = delta; }
 override public string type()
{return String.Format("Трансформатор: Delta= {0}", Delta);}
     }
```

Переопределим type в классе Прямоугольник и изменим Main:

```
Место[] mass = {      new Место(10,10),
                          new Линия(15,15,15),
                          new Окружность(20,20,20),
                          new Прямоугольник(25,25,25,25),
                          new Трансформатор(30,30,30,5)
               };
```

```
Место: X=10, Y=10
Линия: Length= 15
Окружность: Radius= 20
Прямоугольник: Xsize= 25 Ysize= 25
Трансформатор: Delta= 5
```

В случае когда в классе Окружность вместо

```
override public string type(){…}
```

присутствует

```
new public string type(){…}
```

транслятор фиксирует ошибку со следующей формулировкой

'Элементы3.Трансформатор.type()': не может отменить наследованный элемент 'Элементы3.Окружность.type()', потому что этот элемент не отмечен как virtual, abstract или override.

Модификатором abstract отмечаются методы, которые по своей природе также являются виртуальными, но в своём классе не имеют реализации (в других языках они назывались *чистыми виртуальными методами*). Абстрактными могут быть и классы, которые изначально не предназначены для создания экземпляров, а используются только в системе наследования. Иначе говоря, система виртуальных методов type на модификаторе new в классе Окружность прерывается. Выход здесь один: в классе Трансформатор использовать сокрытие с помощью new. Но в этом случае система вызовет для объекта классов Окружность и Трансформатор экземпляр виртуального метода базового класса Место. Чтобы воспользоваться функциональностью собственных методов type в классах Окружность и Трансформатор в Main-метод в самый конец добавим две инструкции:

```
Console.WriteLine(((Окружность)mass[2]).type());
Console.WriteLine(((Трансформатор)mass[4]).type());
```

```
Место: X=10, Y=10
Линия: Length= 15
Место: X=20, Y=20
Прямоугольник: Xsize= 25 Ysize= 25
Место: X=30, Y=30
Окружность: Radius= 20
Трансформатор: Delta= 5
```

Содержимое консольного экрана после запуска на выполнение позволяет сделать следующие выводы:

- в полиморфных вызовах (в данном примере в цикле foreach) для объектов классов Окружность и Трансформатор используется метод базового класса Место, так как своих реализаций они иметь не могут;
- можно заставить *работать* и собственные методы type в экземплярах классов Окружность и Трансформатор (две последние строки), но к полиморфизму это уже никакого отношения не имеет – для этого ссылку приходится явно приводить к требуемому типу.

Если переопределить в производном классе (с помощью override) можно только *методы*, то перекрыть (с помощью new) можно и поля, и вложенные классы.

Абстрактные методы и классы

Метод, который подлежит переопределению в производных классах, в базовом классе может не иметь реализации вовсе. В этом случае вместо модификатора virtual он должен иметь модификатор abstract. Правда, при этом как *абстрактный* должен быть помечен и сам базовый класс, и в таком случае ему запрещено иметь собственные реализации. Взяв за основу пример Элементы2, выполним следующие изменения в коде:

- в объявлении класса Место

```
abstract  class Место
 {
  int X,Y;
  public Место(int x,int y){X=x; Y=y;}
  abstract public string type();//объявление без
реализации
  } //ранее это называлось – прототип, а подобная
конструкция
    // в базовом классе – чисто  виртуальной функцией
```

- в методе Main из массива исключаем объект типа Место

```
static void Main()
        {
         Место[] mass = {new Линия(15,15,15),
                         new Окружность(20,20,20),
                         new Прямоугольник(25,25,25,25)};
   foreach (Место i in mass) Console.WriteLine(i.type());
        }
```

```
Линия: Length= 15
Окружность: Radius= 20
Прямоугольник: Xsize= 25 Ysize= 25
```

ИНТЕРФЕЙСЫ

Интерфейсом называют особый класс, который инкапсулирует абстрактный набор функциональных возможностей. Объявляется интерфейс с помощью специального ключевого слова `interface`:

```
interface  IИмя { содержимое }
```

Как специальный (можно сказать, упрощенный) класс, интерфейс обладает следующими особенностями:

- может содержать только объявления методов или свойств (без определений). Такие элементы называют абстрактными. Иногда говорят, что интерфейс – это чистый протокол. В таком случае модификатор `abstract` явным образом использовать не нужно, так как данный режим установлен по умолчанию;
- элементы интерфейса всегда открыты (`public` по определению) и модификаторы защиты при их объявлении также не указываются;
- имя интерфейса рекомендуют начинать с `I`.

При объявлении класса или структуры, поддерживающих интерфейс, имя интерфейса указывается в списке базовых классов (по сути это тоже наследование). У структур такой список может состоять только из интерфейсов. Если интерфейс поддерживает производный класс, имя его базового класса в списке должно быть первым. В любом случае все элементы поддерживаемого интерфейса (то есть абстрактные методы) должны быть определены (реализованы) в классе или структуре, в чем заключается одно из отличий от виртуальных методов, переопределение которых в производном типе может отсутствовать.

В условиях отсутствия в C# множественного наследования от классов интерфейсы позволяют поддержать неограниченное количество вариаций поведения (то есть *полиморфизм*).

Дополнительные возможности для работы с интерфейсами дают операции `is` и `as`:

- операция `is` бинарная (*двухоперандная*) логическая. Возвращает значение `true`, если тип объекта совместим с указанным интерфейсом, и `false` – в противном случае. Проще говоря, операция проверяет, есть ли в составе данного объекта элементы

указанного интерфейса. Синтаксис: `Name is IName`. Здесь: `Name` – имя объекта; `IName` – имя интерфейса;

- операция `as` также бинарная и возвращает ссылку на элемент-интерфейс в составе объекта

```
IName i = Name as IName;
```

Здесь `i` – ссылка на интерфейс. Если тип данного объекта не поддерживает данный интерфейс, операция `as` возвращает `null`. Говорят, что при использовании данной операции реализуется доступ через интерфейсную ссылку. После этого допустим следующий вызов:

```
i.ИмяЭлементаИнтерфейса;
```

где `ИмяЭлементаИнтерфейса` – имя метода или свойства.

Операции `is` и `as` поддерживают механизм динамической идентификации типов (*RTTI – runtime type identification*), и поэтому их функциональность заметно шире:

- операция `is` в общем случае проверяет объект на совместимость с типом. Наследование классу означает такого рода совместимость для экземпляра класса-потомка (экземпляр класса-потомка в любом случае содержит наследуемую часть);
- операция `as` возвращает ссылку на наследуемую часть, если объект совместим с данным типом, или `null` – в противном случае.

Ссылку на элемент-интерфейс (который входит в состав объекта) можно получить и с помощью операции явного приведения типов (это называется *объектной ссылкой*):

```
…(IName)Name.ИмяЭлементаИнтерфейса …
```

Один и тот же интерфейс может поддерживаться (и реализовываться) типами из различных иерархий наследования. Это даёт возможность использовать полиморфизм для семантически несовместимых объектов, если все они поддерживают один интерфейс. К примеру, можно объявить массив интерфейсных объектов

```
IName[] Имя = { new Type1(), new Type2() …};
```

и тогда в цикле с помощью инструкции

```
… Имя[i].ИмяЭлементаИнтерфейса …
```

можно осуществить доступ к элементу интерфейса конкретного объекта, если `Type1`, `Type2` – типы, реализующие интерфейс `IName`.

Как следует из строки объявления, здесь также работает механизм неявного приведения ссылки производного класса к ссылке базового. Интерфейс чаще всего применяется для типов различных иерархий, и, таким образом, полиморфизм может быть перенесен на набор семантически не связанных типов. В пределах одной иерархии наследования для этих целей удобнее использовать виртуальные функции.

Разработка и использование интерфейсов

В следующем примере интерфейс `IName` реализуется независимыми друг от друга классами `A`, `B`, `C`:

```
namespace Интерфейс
{
    interface IName {
        string  type();
                      }
    class A : IName
    {
        public string type() { return "A"; }
    }
    class B : IName
    {
        public string type() { return "B"; }
    }
    class C : IName
    {
        public string type() { return "C"; }
     static void Main()
     {
 IName[] imass = { new A(), new B(), new C()};
foreach (IName i in imass) Console.WriteLine(i.type());
     }
    }
}
```

```
A
B
C
```

Здесь также при заполнении массива интерфейсных ссылок работает механизм неявного приведения типа производного класса к типу базового.

Реализация интерфейса в системах наследования

Следующий пример демонстрирует объявление и реализацию интерфейса `IDraw` в системе наследования из четырёх классов: `Место`, `Линия`, `Окружность`, `Прямоугольник` (рис. 39). Метод `draw`, объявленный в интерфейсе, имитирует воспроизведение экземпляра типа.

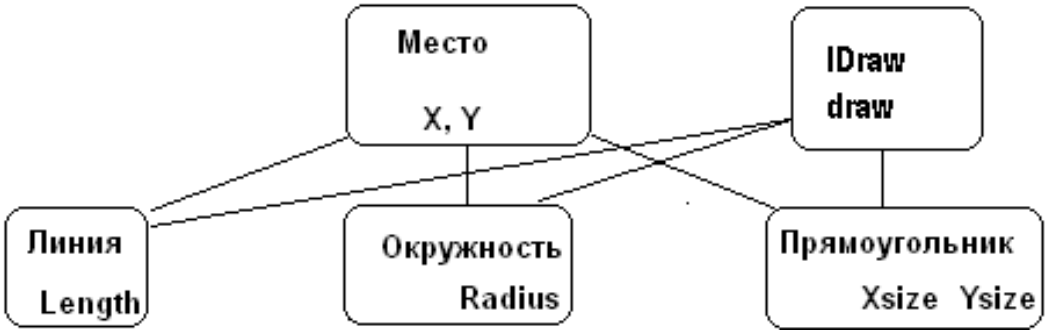


Рис. 39. Схема наследования

```
namespace Интерфейс1
{
    interface IDraw {  void draw();  }
    class Место
    {
        protected int X,Y;//ранее были private
        public Место(int x,int y){X=x; Y=y;}
    }
class Линия : Место, IDraw
 {
  int Length;
  public Линия(int x,int y,int length):base(x, y)
         {Length=length;}
  public void draw() { Console.WriteLine
 ("линия   от   ({0},{1})   до   ({2},{3})",   X,   Y,
X+Length,Y);}
   }
class Окружность : Место , IDraw
{
int Radius;
public Окружность(int x,int y,int radius):base(x,y)
          {Radius=radius;}
public void draw()
```

```
{ Console.WriteLine
 ("окружность в ({0},{1}) радиусом {2}", X, Y,
Radius); }
}
class Прямоугольник : Место, IDraw
 {
 int Xsize, Ysize;
 public Прямоугольник(int x, int y, int xsize, int
ysize):
        base(x, y)
        { Xsize = xsize; Ysize = ysize;}
 public void draw() { Console.WriteLine
   ("прямоугольник с диагональю ({0},{1})-({2},{3})",
       X, Y, X + Xsize, Y + Ysize); }
 static void Main()
        {
            Место[] mass = {new Место(10,10),
            new Линия(15,15,15),
            new Окружность(20,20,20),
            new Прямоугольник(25,25,25,25),
                            };
  foreach (Место i in mass)
  if (i is IDraw) { IDraw j = i as IDraw; j.draw(); }
        }
    }
}
```

```
линия от (15,15) до (30,15)
окружность в (20,20) радиусом 20
прямоугольник с диагональю (25,25)-(50,50)
```

Вместо цикла `foreach` может быть простой `for`, в таком случае можно обойтись одной операцией `as`:

```
IDraw p;
for (int i = 0; i < mass.Length; i++)
{
    p = mass[i] as IDraw;
    if (p != null) p.draw();
}
```

Иной вариант реализации интерфейса: пусть интерфейс поддерживает только *базовый* класс системы, при его реализации интерфейсный метод помечается как `virtual`, а в производных переопределяется обычным способом:

Рис. 40. Поддержка интерфейса `IDraw` классом `Место`

```
namespace Интерфейс1
{
    interface IDraw {  void draw();  }
class Место : IDraw
    {
        protected int X,Y;
        public Место(int x,int y){X=x; Y=y;}
        virtual public void draw()
         { Console.WriteLine(  "Точка в({0},{1})", X, Y);
}
    }
class Линия : Место
    {
      int Length;
      public Линия(int x, int y, int length) : base(x, y)
                { Length = length; }
       override public void draw()
              { Console.WriteLine
   ("линия от ({0},{1}) до ({2},{3})",X, Y, X+Length,Y);}
    }
class Окружность : Место
    {
     int Radius;
   public Окружность(int x, int y, int radius):base(x, y)
         { Radius = radius; }
        override public void draw() { Console.WriteLine
("окружность в ({0},{1}) радиусом {2}",
X,Y,Radius);}
    }
class Прямоугольник : Место
   {
    int Xsize, Ysize;
public Прямоугольник(int x, int y, int xsize, int ysize)
   :base(x, y)
  { Xsize = xsize; Ysize = ysize;}
     override public void draw()
{Console.WriteLine("прям-к с диаг ({0},{1})-({2},{3})",
        X, Y, X + Xsize, Y + Ysize); }
 static void Main()
```

```
        {             Место[] mass = {new Место(10,10),
                      new Линия(15,15,15),
                      new Окружность(20,20,20),
                      new Прямоугольник(25,25,25,25),
                            };
            foreach (Место i in mass)
                if (i is IDraw) ((IDraw)i).draw();
        }
    }
}
```

```
Точка в (10,10)
линия от (15,15) до (30,15)
окружность в (20,20) радиусом 20
прямоугольник с диагональю (25,25)-(50,50)
```

В следующем примере интерфейс поддерживают *все* классы рассматриваемой системы (рис. 41).

```
{
    interface IDraw {
        void draw();
                      }
class Место : IDraw
    {
        protected int X,Y;
        public Место(int x,int y){X=x; Y=y;}
        public void draw()
        { Console.WriteLine( "Точка в ({0},{1})", X, Y);
}
    }
class Линия : Место, IDraw
    {
        int Length;
  public Линия(int x, int y, int length) : base(x, y)
                     { Length = length; }
        public void draw()
   { Console.WriteLine( "линия от ({0},{1}) до
({2},{3})",
         X, Y, X+Length,Y); }
    }
class Окружность : Место, IDraw
    {
        int Radius;
  public Окружность(int x, int y, int radius) :base(x, y)
                    { Radius = radius; }
        public void draw()
```

```
{Console.WriteLine( "окружность в ({0},{1}) радиусом {2}",
        X, Y, Radius); }
    }
class Прямоугольник : Место, IDraw
    {
   int Xsize, Ysize;
 public Прямоугольник(int x, int y, int xsize, int ysize)
         : base(x, y) { Xsize = xsize; Ysize = ysize;}
   public void draw()
{Console.WriteLine("прям-к с диаг ({0},{1})-({2},{3})",
             X, Y, X + Xsize, Y + Ysize); }
 static void Main()
        {
            Место[] mass = {new Место(10,10),
                            new Линия(15,15,15),
                            new Окружность(20,20,20),
                            new
Прямоугольник(25,25,25,25),
                           };
           foreach (Место i in mass)
           if (i is IDraw) ((IDraw)i).draw(); //вместо as

        }
    }
}
```

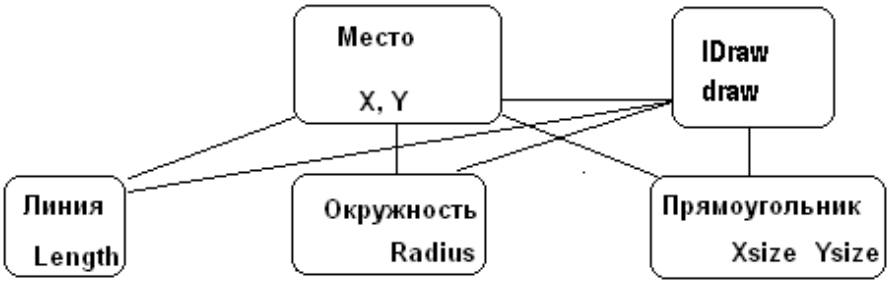


Рис. 41. Поддержка интерфейса `IDraw`
всеми классами системы **`namespace Интерфейс1`**

Вывод на консоль соответствует результатам со с. 157. В рассмотренном примере каждый производный класс *перекрывает* в своём составе наследуемый метод `draw` базового класса (при этом может и не иметь собственной реализации интерфейса, и пользоваться наследуемой). Характерно, что при этом никаких предупреждений от транслятора не поступает (хотя нет модификатора `new`). Убедиться в том, что наследуемые реализации в составе экземпляра производного

имеются, можно на примере класса Прямоугольник. Пусть его метод draw определён следующим образом:

```
public void draw() {
            base.draw();
      Console.WriteLine
      ("прямоугольник с диагональю ({0},{1})-({2},{3})",
             X, Y, X + Xsize, Y + Ysize); }
```

```
Точка в (10,10)
линия от (15,15) до (30,15)
окружность в (20,20) радиусом 20
Точка в (25,25)
прямоугольник с диагональю (25,25)-(50,50)
```

Две последние строчки (вместо одной в предыдущем выводе) свидетельствуют, что базовая реализация интерфейсного метода draw в экземпляре производного класса Прямоугольник имеется.

Стандартные интерфейсы

В библиотеке .NET имеется множество интерфейсов, которые определяют разнообразные «траектории» поведения объектов, в частности:

- IComparable и IComparer – сравнение объектов. Данный интерфейс используется методом стандартной сортировки;
- ICloneable – клонирование объектов;
- IEnumerable и IEnumerator – поддержка цикла foreach.

Встроенные (внутренние) типы также поддерживают стандартные интерфейсы. Например, тип Array из перечисленных реализует интерфейсы IEnumerable и ICloneable.

Интерфейс IComparable

Интерфейс IComparable (компарабельный) – простое сравнение – объявлен в пространстве имён System, содержит всего один метод, возвращающий результат сравнения двух объектов (текущего и obj):

```
interface IComparable { int CompareTo(object obj)}
```

Возвращаемое значение: 0 – объекты равны;
> 0 – текущий больше obj;
< 0 – текущий меньше obj.

Следующий пример для демонстрационного типа `Complex` (структурный тип) реализует поддержку интерфейса `IComparable`, которая необходима для использования стандартной сортировки массива комплексных чисел. Параметром сравнения принята длина радиус-вектора.

```
using System;
class Интерфейс2
{   struct Complex : IComparable // поддерживает интерфейс
    {
        double Real, Image;
        public Complex(int inR, int inI)
        { Real = inR; Image = inI;}
        public override string ToString()
        {
            string s;
            if (Image >= 0) s = Real + "+j" + Image;
            else s = Real + "-j" + Image * (-1);
            return s;
        }
public int CompareTo(object obj) //определение метода
        {
   Complex temp = (Complex)obj;
   double Mthis, Mobj;
Mthis = Math.Sqrt(Real * Real + Image * Image);
Mobj=Math.Sqrt(temp.Real*temp.Real+temp.Image*temp.Im
age);
   if (Mthis == Mobj) return 0;
   else if (Mthis > Mobj) return 1;
   else return -1;
        }
    }
 static void Main()
    {
Complex[] b={ //объявление и инициализация массива
            new Complex(10, 20),
            new Complex(1, 2),
            new Complex(10, 2),
            new Complex(1, 20)
            };
        Console.WriteLine("До сортировки:");
        for( int i=0; i<b.Length;i++)
            Console.Write("
b[{0}]={1}",i,b[i].ToString());
```

```
Array.Sort(b);//Стандартная сортировка выполняется
только с IComparable
Console.WriteLine("\nПосле сортировки:");
for (int i = 0; i < b.Length; i++)
       Console.Write(" b[{0}]={1}", i,
b[i].ToString());
      }
}
```

```
До сортировки:
 b[0]=10+j20 b[1]=1+j2 b[2]=10+j2 b[3]=1+j20
После сортировки:
 b[0]=1+j2 b[1]=10+j2 b[2]=1+j20 b[3]=10+j20
```

Интерфейс `IComparer`

Интерфейс `IComparer` (компаратор) – сравнение по различным критериям – объявлен в пространстве имён `System.Collection`:

```
interface IComparer { int Compare(object ob1, object ob2)}
```

Пример демонстрирует использование интерфейса `IComparer` для сортировки комплексных чисел по действительной и мнимой составляющим. Технология предусматривает создание дополнительных (в данном случае вложенных) классов для каждого критерия сортировки, в которых объявляется поддержка интерфейса `IComparer`. Объекты этих классов, переданные в качестве второго аргумента стандартной сортировки (есть и такая перегрузка), определяют её стратегию.

```
using System;
using System.Collections; // добавленное пространство
имён
class Интерфейс3
{
    struct Complex
    {
        double Real, Image;
        public Complex(int inR, int inI)
        { Real = inR; Image = inI; }
        public override string ToString()
        {
            string s;
            if (Image >= 0) s = Real + "+j" + Image;
            else s = Real + "-j" + Image * (-1);
            return s;
        }
```

```
   public struct SortByReal : IComparer
        {
            public int Compare(object ob1, object ob2)
            {
             Complex temp1 = (Complex)ob1;
             Complex temp2 = (Complex)ob2;
             if (temp1.Real > temp2.Real) return 1;
             else if (temp1.Real < temp2.Real) return -1;
             else return 0;
            }
        }
   public struct SortByImage : IComparer
        {
         public int Compare(object ob1, object ob2)
         {
          Complex temp1 = (Complex)ob1;
          Complex temp2 = (Complex)ob2;
          if (temp1.Image > temp2.Image) return 1;
           else if (temp1.Image < temp2.Image) return -1;
              else return 0;
           }
        }
    }
 static void Main()
    {
        Complex[] b =   {
            new Complex(11, 22),
            new Complex(1, 2),
            new Complex(10, 4),
            new Complex(3, 20)
                        };
        Console.WriteLine("До сортировки:");
        for( int i=0; i<b.Length;i++) Console.Write
                    (" b[{0}]={1}",i,b[i].ToString());
  Array.Sort(b, new Complex.SortByReal());
//сортировка1
      Console.WriteLine("\nПосле сортировки по  Real:");
for (int i = 0; i < b.Length; i++)
      Console.Write(" b[{0}]={1}", i, b[i].ToString());
  Array.Sort(b,new Complex.SortByImage()); //сортировка2
      Console.WriteLine("\nПосле сортировки по  Image:");
  for (int i = 0; i < b.Length; i++)
      Console.Write(" b[{0}]={1}", i, b[i].ToString());
 }
}
```

```
До сортировки:
 b[0]=11+j22 b[1]=1+j2 b[2]=10+j4 b[3]=3+j20
После сортировки по  Real:
 b[0]=1+j2 b[1]=3+j20 b[2]=10+j4 b[3]=11+j22
После сортировки по  Image:
 b[0]=1+j2 b[1]=10+j4 b[2]=3+j20 b[3]=11+j22
```

Интерфейс `ICloneable`

Данный интерфейс предназначен для создания *глубоких* копий объектов. В классе `object` имеется метод `MemberwiseClone()`, который выполняет поверхностное копирование. В случае когда класс или структура состоит только из полей значимых типов, для получения полноценной копии объекта достаточно вызвать `MemberwiseClone()`. Особенность его вызова заключается в том, что объявлен этот метод в `object` с модификатором `protected`, поэтому доступен только метод:

```
protected object MemberwiseClone()
```

Однако для полей *ссылочных* типов результатом такого клонирования будет ссылка на ранее размещённые данные (то есть новый объект не создаётся).

Следующий пример демонстрирует поверхностное копирование. В нём тип `Typ` содержит ссылочный объект – массив, который изначально создаётся по количеству аргументов, переданных конструктору. Объект `clone` создаётся с помощью метода `Clone`, который, в свою очередь, использует метод `MemberwiseClone()`. Вывод на консоль свидетельствует, что на самом деле массив, содержащий вещественные числа, у объектов `obj` и `clone` общий.

```
using System;
class Интерфейс4
{
    struct Typ
    {
        public double[] Params;
        public Typ( params double[] inParams)
        {
            Params = new double[inParams.Length];
            int j = 0;
            foreach (double i in inParams)
                Params[j++] = i;
        }
```

```
        public object Clone()
         { return this.MemberwiseClone(); }
    }
    static void Main()
    {
        Typ obj = new Typ(1.1, 2.2), clone;
        clone = (Typ)obj.Clone();
        Console.WriteLine("После клонирования");
        Console.WriteLine
        ("Оригинал={0},{1}",obj.Params[0],obj.Params[1]);
        Console.WriteLine

("Копия={0},{1}",clone.Params[0],clone.Params[1]);
        clone.Params[0] = 0; //изменяем только клон!
        Console.WriteLine("После изменения клона");
        Console.WriteLine
       ("Оригинал={0},{1}",obj.Params[0],obj.Params[1]);
        Console.WriteLine
      ("Копия={0},{1}", clone.Params[0],clone.Params[1]);
    }
}
```

```
После клонирования
Оригинал = 1,1, 2,2
Копия    = 1,1, 2,2
После изменения клона
Оригинал = 0, 2,2
Копия    = 0, 2,2
```

Если бы объект `clone` создавался конструктором копии по умолчанию

```
Typ clone=obj;
```

результат был бы такой же.

Интерфейс *клонируемый* содержит объявление метода `Clone()`:

```
public interface ICloneable
            { object Clone();}
```

Для создания глубокой копии класс объявляется поддерживающим интерфейс `ICloneable` и реализует метод `Clone()`:

```
using System;
class Интерфейс4
{
    struct Typ : ICloneable
```

```
    {
        public double[] Params;
        public Typ( params double[] inParams)
        {
            Params = new double[inParams.Length];
            int j = 0;
            foreach (double i in inParams)
                Params[j++] = i;
        }
       public object Clone()
       { return new Typ(Params); }
    }
    static void Main()
  {
 Typ obj = new Typ(1.1, 2.2), clone;
 clone=(Typ)obj.Clone();
//необходимо явное приведение типа
 Console.WriteLine("После клонирования");
 Console.WriteLine
 ("Оригинал = {0}, {1}", obj.Params[0],obj.Params[1]);
 Console.WriteLine
 ("Копия    = {0}, {1}",
clone.Params[0],clone.Params[1]);
    clone.Params[0] = 0;
        Console.WriteLine("После изменения клона");
Console.WriteLine
  ("Оригинал = {0}, {1}", obj.Params[0],obj.Params[1]);
Console.WriteLine("Копия    = {0}, {1}",
  clone.Params[0],clone.Params[1]);
  }
}
```

```
После клонирования
Оригинал = 1,1, 2,2
Копия    = 1,1, 2,2
После изменения клона
Оригинал = 1,1, 2,2
Копия    = 0, 2,2
```

Для создания полноценных копий объектов можно обойтись и без этого интерфейса (например, перегрузить конструктор копии), но поддержку полиморфизма проще осуществить с помощью `ICloneable`.

Интерфейсы IEnumerable и IEnumerator и стандартный итератор

Данные конструкции объявлены в пространстве имён System.Collection и предназначены для цикла foreach. Основным из них является IEnumerable. Его объявление предельно простое:

```
public interface IEnumerable
            { IEnumerator GetEnumerator();}
```

Как следует из объявления, единственный метод интерфейса GetEnumerator() должен возвращать ссылку на другой интерфейс IEnumerator. Этот интерфейс содержит уже три элемента: два метода и одно get-свойство –

```
public interface IEnumerator
            {
            void Reset();
            object Current { get;}
            bool MoveNext();
            }
```

Назначение элементов следующее:

- метод Reset() должен установить «движок» на начальную позицию (начальный элемент);
- get-свойство Current возвращает значение текущего элемента;
- метод MoveNext() должен передвинуть «движок», то есть сменить текущий элемент на следующий в наборе и вернуть значение true. Если перемещение движка невозможно, метод должен вернуть false.

Реализация упомянутых интерфейсов показана в следующем примере.

```
using System.Collections;
class Интерфейс7
{
 public   struct Typ : IEnumerable, IEnumerator
    {double[] M;
     int Cur;
     public Typ(params double[] inm)
        {
```

```
            M = new double[Cur = inm.Length];
            int j = 0;
            foreach (double i in inm)M[j++] = i;
        }
    public IEnumerator GetEnumerator()
        {
            return (IEnumerator)this;
        }
     public bool MoveNext()
     {
         if (Cur > 0 )
         {
             Cur--;
             return true;
         }
         else return false;
     }
    public object Current
     {
         get {
             return M[Cur];
             }
     }
    public void Reset()
       {
         Cur = -1;
       }
   }
  static void Main()
    {
        Typ obj = new Typ(1.1, 2.2, 3.3, 4.4, 5.5);
        foreach (double i in obj)
            Console.WriteLine("{0}", i);
    }
}
```

```
5,5
4,4
3,3
2,2
1,1
```

На практике удобнее использовать итератор, появившийся в версии 2.0. Так оформляется блок кода, который задаёт последовательность перебора элементов в коллекции. На каждом проходе цикла `foreach` (а именно он – цель поддержки названных интерфейсов)

выполняется один шаг итератора, заканчивающийся возвращением методом GetEnumerator очередного значения с помощью ключевого слова yield (в дословном переводе *yield* – приводить, приносить). При этом отпадает необходимость в ручном управлении движком с помощью элементов интерфейса IEnumerator.

В примере, который следует далее класс Iter объявлен максимально простым. Его назначение – реализация интерфейса IEnumerable с помощью yield:

```
using System;
using System.Collections;// не забыть !
class Итератор
{
 public   class Iter : IEnumerable
    {
    public IEnumerator GetEnumerator()
        {
            yield return 5;
            yield return 4;
            yield return 4;
            yield return 2;
        }
    }
  static void Main()
    {
foreach (int i in new Iter()) Console.WriteLine("{0}",
i);
    }
}
```

```
5
4
4
2
```

Характерно, что итератор i в цикле foreach должен быть того конкретного типа, который фактически возвращает GetEnumerator.

В следующем примере для уже известного класса Typ выполнена поддержка интерфейса IEnumerable с использованием yield:

```
using System;
using System.Collections;// не забыть !!
class Итератор1
{  public struct Typ : IEnumerable
```

```
    {   double[] M;
        public Typ(params double[] inP)
        {   M = new double[inP.Length];
            int j = 0;
            foreach (double i in inP)M[j++] = i;
        }
     public IEnumerator GetEnumerator()
       {
for(int i=0; i < Params.Length; i++)yield return M[i];
       }
 }
  static void Main()
    {
        Typ obj = new Typ(1.1, 2.2, 3.3, 4.4, 5.5);
       foreach (double i in obj)
               Console.WriteLine("{0}", i);
    }
}
```

```
1,1
2,2
3,3
4,4
5,5
```

КОЛЛЕКЦИИ

Под термином *коллекция* в общем случае понимается группа объектов. В пространстве имён `System.Collection` определено достаточно большое количество классов и интерфейсов, которые позволяют использовать готовые решения для построения достаточно сложных структур (в широком смысле) данных. Стеки, очереди, словари – примеры достаточно часто используемых коллекций.

Основной эффект от использования коллекций – стандартизация способов обработки групп объектов в прикладных программах. Программист имеет возможность использовать уже готовые (встроенные) коллекции или разрабатывать собственные, специализированные.

Интерфейсы пространства имён `System.Collection` (рис. 42) используются в классах коллекций:

- `ICollection` – защищённость и целостность коллекций;
- `IList` – добавление, удаление и индексирования элементов;
- `IDictonary` – возможность представления содержимого в виде «имя – значение»;

- `IEnumerable` – перечислитель, который поддерживает простое перемещение по коллекции для обеспечения функциональности `foreach`;
- `IEnumerator` – простое перемещение по коллекции;
- `IComparer` – сравнение двух объектов.

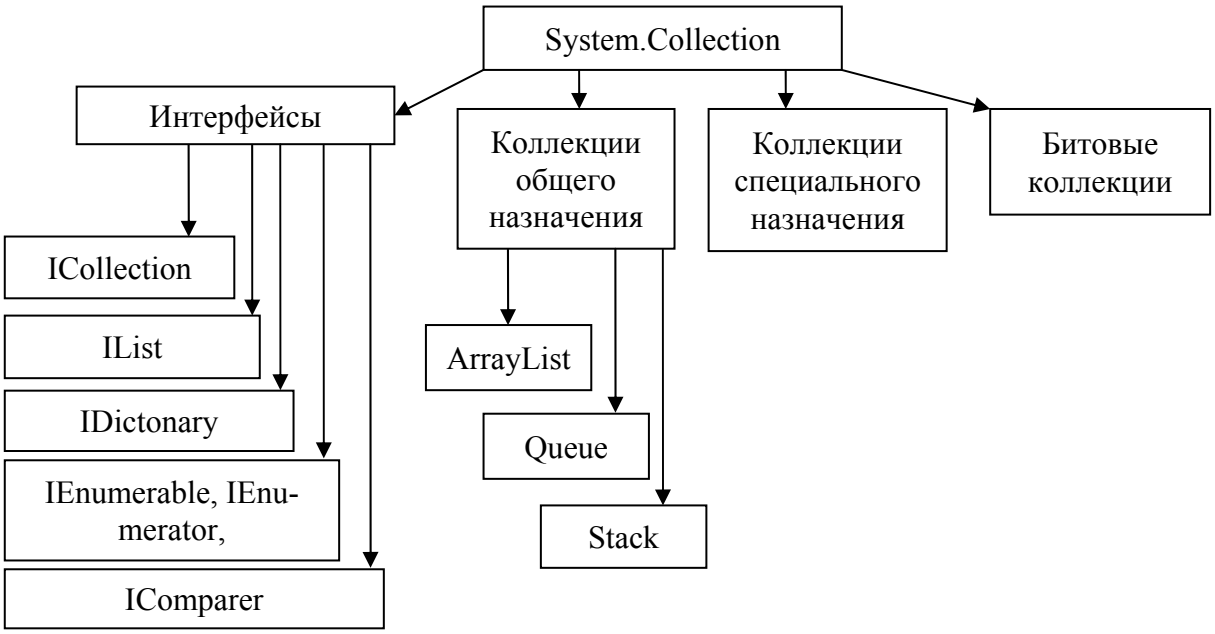


Рис. 42. Стандартные интерфейсы и коллекции

Коллекция ArrayList

Тип `ArrayList` предназначен для создания массива с переопределяемым размером. Тип поддерживает интерфейсы `IList`, `ICollection`, `IEnumerable`, `ICloneable`.

Среди наиболее используемых методов этого типа можно назвать:

- `Add` – добавление объекта в конец списка. Возвращает индекс, соответствующий добавленному объекту;
- `Contains` – проверку вхождения объекта в данный список;
- `CopyTo` – копирование списка в совместимый одномерный массив;
- `IndexOf (LastIndexOf)` – поиск объекта в списке с начала (с конца);

- `Insert` – вставку объекта по указанному индексу. Элементы списка, расположенные после точки вставки, сдвигаются;
- `Remove (RemoveAt)` – удаление элемента из списка (с заданным индексом).

Из свойств этой коллекции можно упомянуть следующие:

- `Count` – количество элементов, которые фактически содержит список (`get`);
- `Item` – получение или установка значения элемента по указанному индексу (`get/set` индексатор).

ДЕЛЕГАТЫ

Делегаты на языке C# являются особым *классом*, объект которого предназначен для размещения *ссылки(ок)* на какой-либо метод(ы). Говорят и так: делегат инкапсулирует ссылку(и) на метод(ы). Добавление такой конструкции в язык позволяет надёжнее и безопаснее осуществлять динамические вызовы методов.

Делегат-класс объявляется в соответствии с синтаксисом:

```
модификатор  delegate  Тип Имя(тип1 имя1, тип2 имя2 …);
```

Это означает, что делегат-объект (делегата-класса `Имя`) может содержать только ссылки на методы типа `Тип` (возвращающее значение), имеющие сигнатуру (`тип1 имя1, тип2 имя2 …`).

Ниже представлены обычные и необычные качества, которыми обладают делегаты-объекты.

Обычные

1. Делегат-объект является ссылочным объектом.
2. Делегату-объекту требуется выделять память.
3. Делегаты-объекты можно использовать:
   - в операции сравнения;
   - в операциях сложения и вычитания (с однотипными объектами);
   - в качестве аргументов методов.

Необычные

1. Содержимое делегата-объекта – ссылка или список ссылок.
2. Делегату-объекту можно присвоить значение, в том числе с помощью автоматически перегружаемого конструктора *по умолчанию.*
3. Добавить ссылку можно операцией `+=` или методом `Combine()`.
4. Удалить ссылку можно операцией `-=` или методом `Remove()`.
5. Основным предназначением объекта-делегата является вызов метода(ов). Для этого в операторе вызова вместо *имени метода* используется *имя объекта-делегата.*
6. Если объект-делегат содержит более одной ссылки, очерёдность вызова определяется очерёдностью присвоения ссылок. Ссылки при этом могут дублироваться.
7. Входные аргументы вызываемым методам передаются одни и те же (если, конечно, в одном из них входные данные не будут изменены по ссылке). Итоговым значением является значение последнего из вызванных методов.

Особые качества делегатам обеспечивает неявное наследование любого из них от системного класса `MultiCastDelegate`. На любое объявление делегата компилятор включает в сборку класс с одноимённым названием, в котором, в частности, имеется перегруженный `Конструктор` с одним входным аргументом – именем метода, удовлетворяющего типу данного делегата.

Следующий пример представляет технологию использования делегатов. Пример демонстрационный: то же самое без делегатов можно сделать значительно проще.

Пример следующий: в обёрточном классе `Program` имеется два метода, выбор между которыми осуществляется с помощью делегата `d` типа `Type`. Можно сказать и так: делегат `d` получает (или ему делегируются) полномочия вызова методов, в данном случае из класса `Program`.

```
     using System;
     class Program
      {
delegate void Type(string str);//объявление делегата типа Type
     static void TypeToConsol(string str)
     //метод1, подходящий для Type
     {  Console.WriteLine("Вывод на консоль=" + str);  }
     static void TypeToMsg(string str) //метод2,
     подходящий для Type
     {System.Windows.Forms.MessageBox.Show("Вывод в окно="
     + str);}
     static void Main()
        { Console.WriteLine("Задайте строку!");
          string instr= Console.ReadLine();
          Type d;// d  - объект-делегат типа Type
      if( (int)instr[0] > 128) d = new Type(TypeToConsol);
            else  d = new Type(TypeToMsg);
     // аргументом для конструктора делегата является один из
     //«подходящих»  методов
          d(instr);
// вызов метода с помощью делегата. Входной аргумент передаётся
// фактически вызываемому методу. Синтаксис вызова = вызов метода
       }
      }
```

В составе делегата-объекта присутствует также метод `GetInvocationList()`, который возвращает массив ссылок на со-

держащиеся в объекте-делегате методы и тем самым позволяет осуществить выборочный вызов методов объекта-делегата.

Объект-делегат является неизменяемым объектом (как строка, например). Каждый раз при его модификации создаётся новый объект, а старый уничтожается.

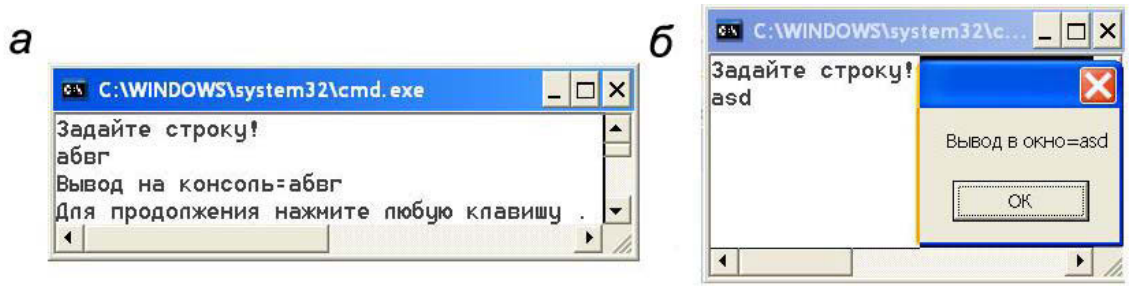


Рис. 43. Вывод на консольное окно:
а – при задании кириллической строки;
б – при задании латинской строки

В следующем примере образован список из двух методов, которым поочерёдно *делегируется* функциональность с помощью делегата-объекта d:

```
using System;
class Program
{
delegate void Type(string str);
static void TypeToConsol(string str)
 {
if((int)str[0]>128)Console.WriteLine("Вывожу кирил.строку="+str);
else Console.WriteLine("Строка - не моя!");
 }
static void TypeToMsg(string str)
 {if((int)str[0]<128)
System.Windows.Forms.MessageBox.Show
("Вывожу лат. строку :" + str);
else System.Windows.Forms.MessageBox.Show
(" Строка не моя!");
}
static void Main(string[] args)
 {
  Console.WriteLine("Задайте строку!");
  string instr= Console.ReadLine();
//объявление объекта с инициализацией :
  Type d = new Type(TypeToConsol);
  d += TypeToMsg; //добавление метода в список
делегата
```

```
    d(instr); //вызов методов из списка делегата
  }
}
```

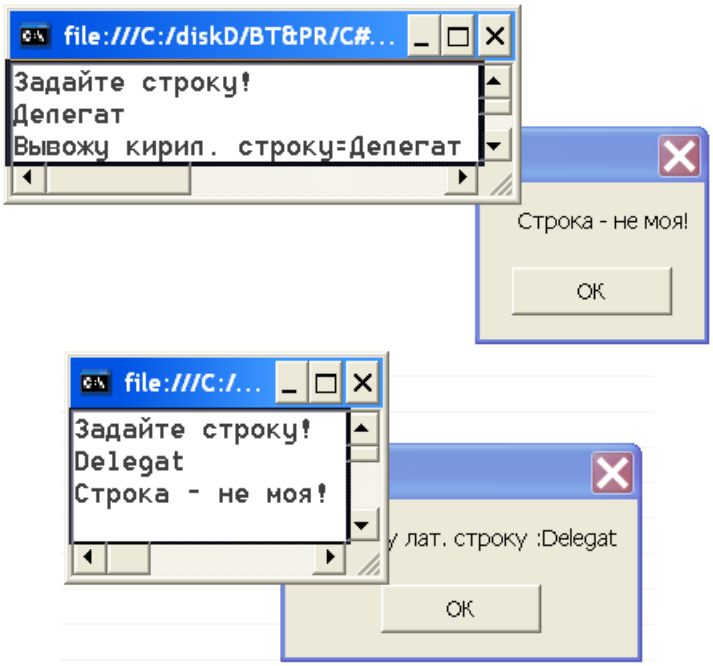


Рис. 44. Варианты вывода

В следующих примерах демонстрируются некоторые возможности использования делегатов:

```
class Program
    {
      delegate void Type(string str);
      static void T1(string str)
        {      Console.WriteLine("T1=" + str);   }
      static void T2(string str)
        {   Console.WriteLine("T2=" + str);      }
      static void T3(string str)
        {     Console.WriteLine("T3=" + str);    }
      static void Main(string[] args)
        {
            string instr= "A";
            Type d  = new Type (T1);
            d += T2;
            d(instr);
            instr += "B";
            d(instr);
            instr += "C";
            d += T3;
            d(instr);
```

```
        }
    }
```

```
T1=A
T2=A
T1=AB
T2=AB
T1=ABC
T2=ABC
T3=ABC
```

```
class Program
    {
    delegate void Type(string str);
    static void T1(string str)
        {  Console.WriteLine("T1=" + str);
        }
    static void T2(string str)
        {  Console.WriteLine("T2=" + str);
        }
    static void T3(string str)
        {  Console.WriteLine("T3=" + str);
        }
    static void Main(string[] args)
        {   string instr= "A";
            Type d  = new Type (T1);
            d += T3;
            d(instr);
            instr += "AB";
            d(instr);
            d += T2;
            d -= T1;
            instr += "CD";
            d(instr);
        }
    }
```

```
T1=A
T3=A
T1=AAB
T3=AAB
T3=AABCD
T2=AABCD
```

СОБЫТИЯ

Общее описание

Механизм делегатов предназначен для использования в более общей технологии программирования событий.

Событием называется особый элемент (поле) класса (типа делегат-класс), с помощью которого он (или объект, который его содержит) может посылать сообщения (уведомления) об изменении своего состояния другим объектам. Фактически передача данных выполняется с помощью вызова соответствующих методов. При этом говорят, что первый объект является источником (`sender`) и *публикует* сообщение, а остальные объекты становятся получателями (`receiver`) и *подписываются* на получение этого сообщения.

Класс источника сообщения (отправитель):

- объявляет *делегат* (объявлением определяются тип и сигнатура вызываемых методов-обработчиков события). Делегат может быть объявлен и вне класса-источника;
- объявляет *событие*;
- определяет метод, инициирующий событие: непосредственно из внешнего кода инициировать событие нельзя!

Класс приёмника (получателя) определяет метод(ы) – обработчик(и) события, тип и сигнатура которых соответствует делегату, которым объявлен элемент-событие.

До момента инициализации события, методы-обработчики должны быть в событии зарегистрированы.

При объявлении события среда создаёт закрытый статический класс, в котором находится экземпляр делегата и операторные методы `+=` и `-=` для добавления или удаления обработчиков. Тип этих операций – `void`. Объявление события состоит из двух этапов: 1) объявление делегата

```
МодификаторДоступа delegate ТипДелегата ИмяДелегата(Тип имя);
```

2) собственно объявление события в классе-источнике события

```
class ИмяКласса
{
 public event  ИмяДелегата ИмяСобытия;
}
```

Как видно из объявлений, событие – это поле типа `Делегат`. При объявлении это поле инициализируется значением `null`.

В следующем примере делегат `Type` используется для связи источника события-объекта `S` (класса `Sender`) с получателями-объектами `R1` и `R2` классов `Receiver1` и `Receiver2` (рис. 43).

Отметим, что элементу-событию не нужно выделять память.

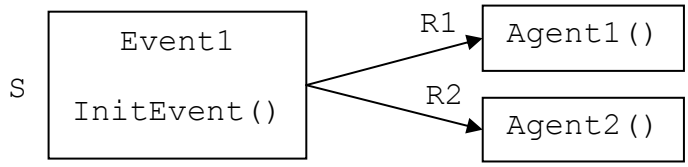


Рис. 43. Схема взаимодействия объектов

```
using System;
class Evnt   //обёрточный!
{
   delegate void Type(string str);
   class Sender
        {   public event Type Event1;
            public void InitEvent()
            {
         if (Event1 != null) Event1(" Свершилось!");
            }
        }
   class Receiver1
   {   public void Agent1(string text)
 { Console.WriteLine("Обработчик1 получил =" + text);
      }
   }
   class Receiver2
   {   public void Agent2(string text)
 { Console.WriteLine("Обработчик2 получил =" + text);
      }
   }
   static void Main()
            {   Sender S = new Sender();
                Receiver1 R1 = new Receiver1();
                Receiver2 R2 = new Receiver2();
                S.Event1 += R1.Agent1;
                S.Event1 += R2.Agent2;
                S.InitEvent();
                Console.ReadLine();
            }
}
```

```
Обработчик1 получил = Свершилось!
Обработчик2 получил = Свершилось!
```

Анонимные обработчики событий

При практическом применении исключений имеется возможность использования *анонимных обработчиков*. Так оформляется несложный код, который размещается сразу в инструкции, в которой выполняется регистрация обработчика события:

```
using System;
    class Evnt1
    {   delegate void Type(string str);
        class Sender1
        {   public event Type Event1;
            public void InitEvent()
            {
        if (Event1 != null) Event1(" Свершилось!");
            }
        }
       static void Main()
        {
                Sender1 S = new Sender1();
       S.Event1 += delegate(string text)
{ Console.WriteLine("Анонимный делегат1 = " + text);};
       S.Event1 += delegate(string text)
{ Console.WriteLine("Анонимный делегат2 = " + text);};
       S.InitEvent();
         Console.ReadLine();
       }
    }
```

```
Анонимный делегат1 =  Свершилось!
Анонимный делегат2 =  Свершилось!
```

Программирование стандартных событий

Представленные выше примеры использования событий преследуют цель познакомить с механизмом событий. В практическом программировании используют стандартные делегаты среды .NET, предназначенные для обработки ситуаций, с помощью которых осуществляется управление программными системами и комплексами (собы-

тия клавиатуры, мыши и тому подобное). Для использования стандартных событий необходимо следовать правилам:

- имя делегата заканчивается суффиксом `EventHandler` (в переводе *event handler* – обработчик события, а в технологии это – *делегат*, инкапсулирующий *ссылки* на обработчики событий);
- делегат определяет два параметра:
  - источник события (предельно общего типа `object`);
  - аргумент события (тип `EventArgs` или производный от него, если требуется дополнительная информация о событии).

  Если дополнительной информации о событии не требуется, то можно не объявлять собственный делегат, а использовать стандартный класс делегата `System.EventHandler`;
- обычно `ИмяОбработчикаСобытия` получается из приставки `On` и имени события.

Предыдущий пример, в котором событие оформлено в соответствии со стандартными правилами, выглядит следующим образом:

```
using System;
    class Event2
    { class Sender2
        {   public event EventHandler Event2; //1
            public void InitEvent()
            {
          if (Event2 != null) Event2(this, null) //2
            }
        }
      class Receiver
       {
 public void OnEvent2(object sender, EventArgs e) //3
           {
             Console.WriteLine
      ("Обработчик получил сообщение
от"+sender.GetType());
           }
        }
   static void Main()
   {
    Sender2 S = new Sender2();
    Receiver R = new Receiver();
    S.Event2 += R.OnEvent2; //4
    S.Event2 += delegate(object sender, EventArgs e)
    //5
```

```
            { Console.WriteLine("Анонимный делегат");
};
     S.InitEvent();
     Console.ReadLine();
    }
  }
```

```
Обработчик получил сообщение от Event2+Sender2
Анонимный делегат
```

Нумерованными комментариями обозначены строки, в которых были произведены изменения.

В следующем примере сообщение, передающееся от источника к получателю, становится более информативным за счёт текстовой строки. Для этого из стандартного класса `EventArgs` наследованием получается пользовательский класс `SenderEventArgs`, в специфицирующей части которого имеется поле `message` (изменение 1). Поскольку этот класс будет использован для обработчиков, изменится их сигнатура, и использовать стандартный делегат `EventHandler` в данном случае нельзя – нужно объявить новый делегат (изменение 2). Третье изменение следует произвести в методе `InitEvent`, который инициирует событие `Event2`. В соответствии с типом этого события (делегат `EventHandler`) необходимо два аргумента. Один из них тот же – ссылка на объект-отправитель сообщения (`this`). Второй аргумент можно создать «на лету». Далее в обработчике события можно использовать этот аргумент (изменение 4).

```
using System;
class Event
    {
class SenderEventArgs : EventArgs
    //1
     {
      public string message;
      public SenderEventArgs(string text)
       { message = text;}
     }
delegate     void     EventHandler(object     sender,
SenderEventArgs e);//2
class Sender2
  {
     public event EventHandler Event2;
     public void InitEvent()
```

```
        {
    if(Event2!= null)Event2(this, new
    SenderEventArgs("Заработало!"));//3
        }
      }
    class Receiver
    {
    public void OnEvent2(object sender, SenderEventArgs e)//4
     {
Console.WriteLine("Обработчик получил сообщение : "+e.message );
     }
    }
    static void Main()
        {   Sender2 S = new Sender2();
          Receiver R = new Receiver();
          S.Event2 += R.OnEvent2;
          S.InitEvent();
          Console.ReadLine();
       }
    }
```

```
Обработчик получил сообщение : Заработало!
```

Необходимо отметить, что имя стандартного делегата `EventHandler` здесь перекрывается.

Расширенные приёмы управления списком обработчиков событий

В предшествующих примерах конструкция `event` использовалась в сокращённом варианте: объявлялось поле-событие (специального обёрточного класса) и с помощью операций +=, −= выполнялось регистрирование или удаление обработчиков из события. Вызов обработчиков после наступления события был возможен исключительно в порядке их регистрации.

Ключевые слова `add` и `remove`, если их использовать в блоке `event`, позволяют управлять процессами регистрации, удаления и в итоге очередностью вызовов обработчиков.

В следующем примере обёрточный класс для события (`Sender`) содержит массив на три делегата, каждый из которых предназначен для размещения ссылки на один метод-обработчик:

```
class Event3
    {
      delegate void Delegate();
      class Sender
      {
          Delegate[] md = new Delegate[3];
          public event Delegate Event
          {
           add
              {
                  int i;
                  for (i = 0; i < 3; i++)
                      if (md[i] == null)
                      {
                          md[i] = value;
                          break;
                      }
      if (i == 3) Console.WriteLine("Список полон!");
              }
              remove
              {
                  int i;
                  for (i = 0; i < 3; i++)
                       if (md[i] == value)
                        {
                          md[i] = null;
                          break;
                        }
     if (i == 3) Console.WriteLine("Метод не найден!");
              }

          }
           public void InitEvent()
            {
                for (int i = 2; i >= 0; i--)
                    if (md[i] != null) md[i]() ;
            }
        }
      class Receiver
        {
            public void OnEvent1()
            {
                Console.WriteLine("Обработчик1");
            }
            public void OnEvent2()
            {
```

```
            Console.WriteLine("Обработчик2");
        }
        public void OnEvent3()
        {
            Console.WriteLine("Обработчик3");
        }
    }
 class Program
    {
        static void Main()
        {
            Sender S = new Sender();
            Receiver R = new Receiver();
            S.Event += R.OnEvent1;
            S.Event += R.OnEvent2;
            S.Event += R.OnEvent3;
            S.Event += R.OnEvent1;
            S.InitEvent();
            S.Event -= R.OnEvent1;
            S.Event -= R.OnEvent1;
            S.InitEvent();
            Console.ReadLine();
        }
    }
}
```

```
Список полон!
Обработчик3
Обработчик2
Обработчик1
Метод не найден!
Обработчик3
Обработчик2
```

Конструкции `add` и `remove` напоминают `set`-свойство и перегружают соответствующие операции.

Имея в составе обёрточного класса массив делегатов, достаточно просто организовать любую очередность их вызовов, в том числе и обратную, как в примере.

В общем случае для размещения ссылок на обработчике можно использовать и такие наборы, как стек, прямая очередь или очереди с учётом приоритетов.

Практическое применение стандартного делегата для управления формой

Формой в большинстве современных инструментальных систем принято называть класс, предназначенный для реализации пользовательского интерфейса. В рассматриваемой системе имя этого класса `Form`, и объявлен он в пространстве имён `System.Windows.Forms`. Экземпляр данного класса представляет собой пустое окно, обладающее набором стандартных функций:

- изменением размеров, свёрткой, развёрткой, перемещением по поверхности рабочего стола;
- управлением с помощью стандартных кнопок или меню;
- строкой заголовка.

Основное назначение класса `Form` – создание пользовательских окон различных типов и назначений (стандартные, инструментальные, всплывающие, диалоговые и другие). Для этого класс используется в качестве базового для некоторого пользовательского класса, который может служить, в частности, контейнером для остальных элементов управления.

Для программного управления состоянием окна в классе `Form` предусмотрены следующие методы:

`Show()` – отображение формы в виде немодального окна;

`ShowDialog()` – отображение формы в виде модального окна;

`Activate()` – передача фокуса окну;

`Hide()` – свертка формы в кнопку панели задач;

`Close()` – завершение работы окна.

Кроме этого, *жизненный цикл* любой формы состоит из событий:

`Load` – генерируется один раз при первой прорисовке окна;

`Activated` – многократно генерируется во время жизни формы каждый раз при получении формой фокуса;

`VisibleChanged` – генерируется каждый раз при изменении свойства `Visible` формы (например, с помощью методов `Show()`, `ShowDialog()`, `Hide()`, `Close()`);

`Deactivated` – многократно генерируется во время жизни формы каждый раз при потери формой фокуса;

`Closing` – событие генерируется непосредственно перед закрытием формы, когда процесс закрытия формы можно отменить (для этого свойство `e.Cancel` необходимо установить в значение

`true`, если `e` – объект типа `CancelEventArgs`, переданный в обработчик данного события);

`Closed` – генерируется после закрытия формы.

Для каждого из перечисленных событий в составе формы имеется соответствующий метод-обработчик, имя которого получается с помощью приставки `On` к имени события. При этом каждый из обработчиков объявлен как виртуальный, и поэтому имеется возможность переопределить его в пользовательском (производном от `Form`) классе.

Представляют практический интерес свойства формы:

`Parent` – ссылка на родительское окно. Данное свойство особо актуально для элементов управления, размещение которых определяется смещением относительно верхнего левого угла клиентской области (свойство `Location` типа `Point`, задающее координаты `X`, `Y` смещения). Любой элемент управления – это тоже форма. Первичные формы не имеют родительского окна;

`Visible` – видимость формы (`true`, `false`);

`Enable` – доступность формы (`true`, `false`). Когда форма невидима, она отображается тусклым цветом и не реагирует на управляющие воздействия;

`AutoSize` – включает или отключает режим подгонки размеров окна (или элемента управления) под размер отображаемых на ней данных (текст, изображение).

Порядок взаимного расположения (перекрытия) элементов на форме определяется так называемым *z-порядком* (по аналогии с третьей координатой трёхмерного пространства). Изначально `z`-порядок определяется порядком добавления элементов, но может быть изменён с помощью методов `BringToTop()` и `SendToBack()`.

Форма является контейнером для размещения произвольного количества элементов управления. Организован этот набор в виде коллекции, а доступен через свойство `Controls` типа `ControlCollection`. С помощью данного свойства можно динамически добавлять или удалять элементы управления, обращаться к ним по индексу, а также осуществлять поиск с помощью метода `Find()`, которому в качестве аргумента указывают имя элемента управления. Собственно номер элемента в коллекции и соответствует `z`-порядку элемента.

Ещё одним свойством, упорядочивающим элементы управления, является `TabIndex`. Номера, присваиваемые этому свойству, опре-

деляют порядок обхода элементов на форме с помощью клавиши Tab. Если один или несколько элементов имеют нулевые значения TabIndex, то очередность обхода определяется по z-индексу.

Часто при разработке главного (или дочернего) окна приложения необходимо обеспечить пользователю возможность отмены операции по закрытию формы (например, на кнопку «крестик» в правом верхнем углу окна). Самое простое решение в таком случае – перегрузка обработчика события OnClosing:

```
using System.Windows.Forms;
using System.ComponentModel;
class Form1 : Form
{
 public Form1()
  {
 this.Closing += new
CancelEventHandler(OnFormClosing);
  }
 void OnFormClosing(object s, CancelEventArgs e)
  {
string message = "Продолжить выполнение ?";
string caption = "Выберите вариант!";
MessageBoxButtons buttons = MessageBoxButtons.YesNo;
DialogResult result =
MessageBox.Show(message,caption,buttons);
if (result == DialogResult.No) return;
e.Cancel = true;
}
 static void Main()
    {
        Form1 Форма;
        Форма = new Form1();
        Форма.Text = "Ручная форма";
        Application.Run(Форма);
    }
}
```

ИСКЛЮЧЕНИЯ

Во время выполнения программы зачастую возникают ситуации, когда предусмотренное программистом действие либо выполниться не может в принципе, либо приводит к неопределённому результату (деление на ноль, переполнение, исчезновение порядка и тому подобное). Для того чтобы избежать при этом аварийного завершения программы, предусмотрен так называемый механизм обработки исключительных ситуаций, или механизм исключений. Он сводится к следующим действиям:

- при возникновении особой *исключительной* ситуации система создаёт специальный объект типа *исключение*, который и соответствует данной ситуации (говорят: выбрасывает исключение). Все типы исключений являются потомками одного специального класса `Exception`, определённого в пространстве имён `System`. Имеется большое количество системных исключений (`DivideByZeroException`, `OutOfMemoryException`, `OverflowException` и другие). Программист может разрабатывать и собственные типы исключений;
- информация, инкапсулированная в объекте-исключении, используется в блоке обработки этого исключения. Стандартными действиями (обработка по умолчанию) являются вывод диагностического сообщения и безопасное (для системы) завершение программы, при работе которой выброшено исключение;
- особую гибкость механизму исключений обеспечивает возможность определения собственной обработки исключений (говорят: перехватить исключение);
- исключения могут быть выброшены программно.

В типе `Exception` имеются свойства, которые удобно использовать на практике:

`HelpLink` – URL файла с описанием ошибки;

`Message` – текстовое описание ошибки (`readonly`). Инициализируется при создании объекта-исключения (входной аргумент конструктора);

`Source` – имя объекта или приложения, выбросившего исключение;

`TargetSite` – метод, выбросивший исключение.

Для использования механизма исключений в языке C# имеется три ключевых слова:

- `throw` – генерация исключения;
- `try` – блок проверки на исключение;
- `catch` – блок обработки исключения.

Оператор `throw`

Синтаксис использования оператора:

```
throw ОбъектИсключение; //
объект должен быть уже размещён в памяти,
```

или

```
throw  new = ТипИсключения();
```

После выполнения оператора `throw` дальнейшее выполнение блока кода прекращается, а управление передаётся обработчику данного исключения. Оператор `throw` может быть использован и без аргументов для повторной генерации того же исключения.

Оператор `catch`

Оператор `catch` является заголовком обработчика исключения. Синтаксис использования:

```
catch (Тип объект){блок кода}
```

`Тип` используется для выбора обработчика: тип исключения, выброшенного оператором `throw` должен полностью совпадать с типом, заданным в `catch` (или быть производным от него). Собственно `объект` является входным аргументом для блока обработчика.

Разновидностью `catch` является оператор `finally`, который не имеет входных аргументов, выполняется всегда (в том числе после любого обработчика исключения).

Оператор `try`

Данный оператор определяет блок кода, в котором выполняется проверка на исключение. Обычно так выделяются фрагменты программы, потенциально опасные из-за возникновения исключительных ситуаций. После блока `try` должны следовать блоки обработчи-

ков исключений. В случае их отсутствия управление после генерации исключения передается обработчику по умолчанию.

В следующем примере выбрасываются и перехватываются исключения как системного, так и пользовательского типа.

В любом варианте работы программы удерживается консольное окно (рис. 44), и курсор в последней строке свидетельствует о запущенном методе `ReadLine`.

```
namespace Exc1
{
    class Program
    {
        class Exc1 : Exception
        {
         public   Exc1(string s) : base(s){}
        }
       static void metod(int a)
        {
       if(a<0) throw new Exc1("отрицательное значение");
               Console.WriteLine("Всё ОК!");
        }
        static void Main()
        {
            string s;
            int Num;
            Console.WriteLine("Задайте целое число!");
            s = Console.ReadLine();
            try
            {
                Num = Convert.ToInt32(s);
                metod(Num);
            }
       catch(Exc1 e)
       { Console.WriteLine("Исключение : " + e.Message);}
       catch (Exception e)
        {Console.WriteLine("Исключение : "+e.Message);}
       finally
        { Console.ReadLine();}
        }
    }
}
```

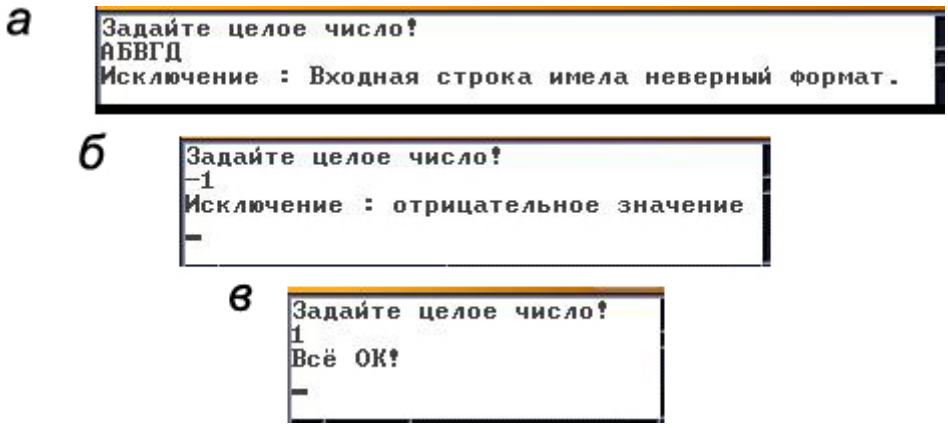


Рис. 44. Консольное окно:
а – при вводе строки, которая не может быть преобразована в целочисленный формат;
б – при задании отрицательного значения;
в – при штатном вводе

Механизм исключений представляет собой альтернативный вариант возврата из методов, освобождая программиста от необходимости увеличения списка входных аргументов и выходных проверок в каждой точке вызова или возврата.

В следующем примере наряду со стандартным (деление на ноль) используется и пользовательское исключение, оформленное в виде класса `B`:

```
namespace Exc2
{
    class A
    {
        static int S = 1;
        public static  string type()
        {
            try
            {
                return String.Format("{0}", 1/S--);
            }
            catch (Exception) { return "E1"; }
        }
    }
    class B : Exception
    {
        public char C;
        public B(string b, char c) : base(b) { C = c;
}
     static void Main()
```

```
        {
            try
            {
            for (int i = 1 ; i >  -2; i--)
                if(i < 0) throw new B("E2",'-');
                else  Console.WriteLine( A.type());
            }
          catch(B b)
            { Console.WriteLine(b.C+b.Message); }
        }
    }
}
```

```
1
E1
-E2
```

До начала работы собственного конструктора класса `B` выполняется явный вызов конструктора базового класса `Exception` с аргументом-строкой, которая используется для инициализации поля `Message`.

СВЯЗНЫЕ СПИСКИ

Понятие об информационных структурах

Обработка больших объёмов данных современными информационными системами возможна исключительно при условии чёткой организации, в том числе и обрабатываемых данных. В общем случае *информацией* считается определённым образом упорядоченный набор структурированных данных. Элементы этого набора связывают так называемые структурные отношения (рис. 45):

- наследования и предшествования;
- начальности и конечности;
- иерархии (подчинения);
- ветвления
- и другие.

.

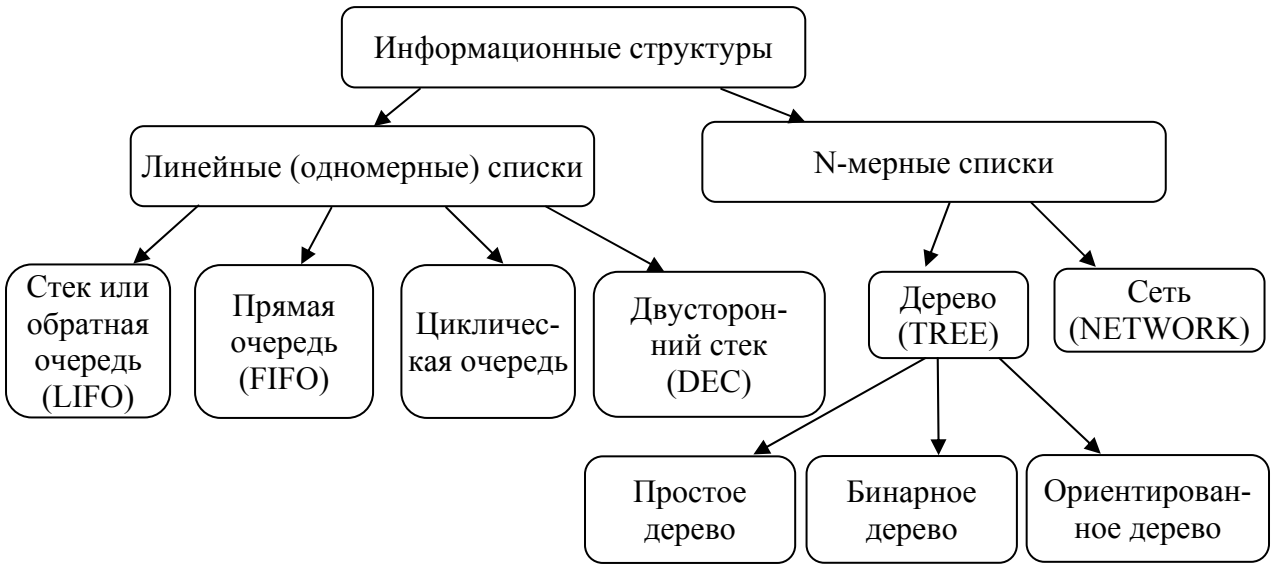


Рис. 45. Структуры данных

Данные, удовлетворяющие тому либо иному типу (типам) структурных отношений, называются *информационными структурами*.

В линейных списках (рис. 46) положение элемента задаётся одной координатой (номер в очереди, сдвиг относительно предыдущего элемента, адрес месторасположения и тому подобное). В N-мерных списках для идентификации элемента, как правило, требуется задать уровень его расположения в иерархии (первая координата), а затем указать N–1 координат, определяющих его расположение на данном уровне.

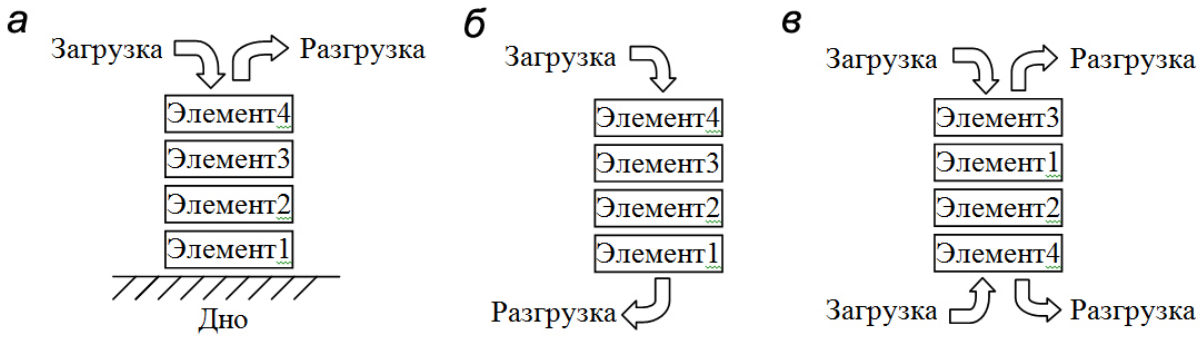


Рис. 46. Линейные очереди:
а – стек; *б* – прямая очередь; *в* – дек

Среди N-мерных списков особое место занимает структура *дерево* (рис. 47). Деревом называется множество, состоящее из $n \geq 0$ элементов:

- у которых имеется один особый элемент – корень дерева;
- остальные $(n - 1)$ элементов входят в $m \geq 1$ взаимно непересекающихся множеств, каждое из которых также является деревом.

Таким образом, дерево является структурой без поперечных связей между элементами. Примером дерева является файловая система. Данный принцип не выполняется в N-мерной структуре, называется сетью. Особый класс деревьев составляют *бинарные*: в них из каждого узла возможно разветвление не более чем на два направления.

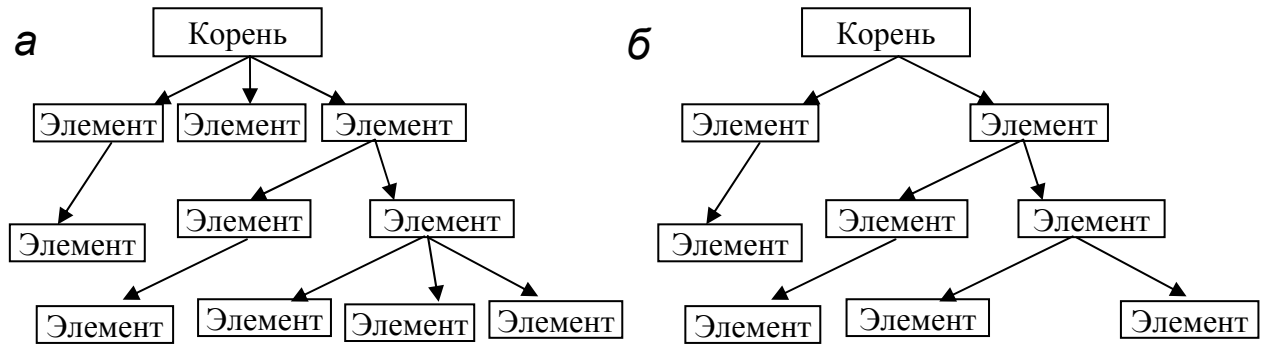


Рис. 47. Дерево (*а*) и бинарное дерево (*б*)

Линейные связные списки

Представим себе некоторую последовательность целых чисел, упорядоченную по возрастанию: 10, 20, 30, 40, 50, 60. Ставится задача создания системы, оперирующей с подобным набором данных. Простейшим вариантом является использование массива:

```
int[6] m= { 10, 20, 30, 40, 50, 60};
```

Когда функциональность разрабатываемой системы ограничивается хранением данных и их предоставлением по соответствующему запросу, такой вариант будет оправдан. При более широкой функциональности проблематично использование массива:

- добавление новых элементов связано с перераспределением всего набора;
- удаление существующих элементов приводит к *разреженности* и нерациональному использованию памяти;
- любая модификация состава элементов требует достаточно сложного протоколирования;
- нетривиальны любые операции по перегруппировке элементов
- и другое.

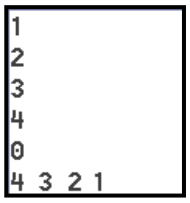
Более универсальной формой реализации информационных структур является связная форма. Элемент связанного списка обладает способностью *самоадресации*, поэтому в общем случае он состоит из двух частей: информационной (info) и адресной (link). Информационная и адресная части элемента связного списка могут, в свою очередь, состоять из произвольного количества полей. В частности, количество полей в адресной части зависит от *проходовости* очереди: элементы *однопроходовой* очереди имеют один адрес, двупроходовой – два.

Последовательность чисел в связной форме может быть организована в обратной очереди (рис. 49, *а*), модификация элементов которой показана на рис. 49, *б*.

Состав элемента очереди может быть определена специальным классом (в примере он будет называться `List`). При этом элементы очереди целесообразно размещать динамически, нулевой адрес считать признаком окончания, а для начального адреса всей очереди дополнительно объявить статическую ссылку*:*

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
namespace ConsoleApplication1
{
    class Program
    {
        class List
        {
            static List top = null;
            int info;
            List link;
            public List(int i)
            {
                info = i;
                if (top == null) link = null;
                else link = top;
                top = this;
            }
            static public void Type()
            {
                List p;
                for ( p=top ; p != null; p = p.link)
                    Console.Write(" {0}",p.info);
            }
        }
        static void Main()// испытательный метод
        {
            string s;
            int i;
            for (; ; )
            {
                s = Console.ReadLine();
                i = Convert.ToInt32(s);
                if (i != 0) new List(i);
                else break;
            }
            List.Type();
        }
    }
}
```



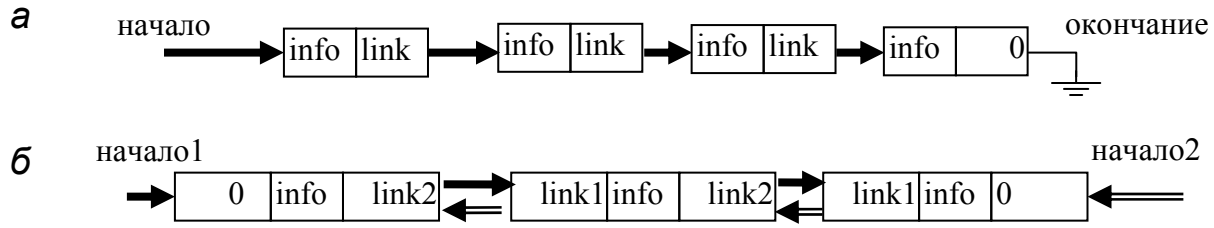


Рис. 48. Однопроходовый (*а*) и двупроходовый (*б*) связные списки

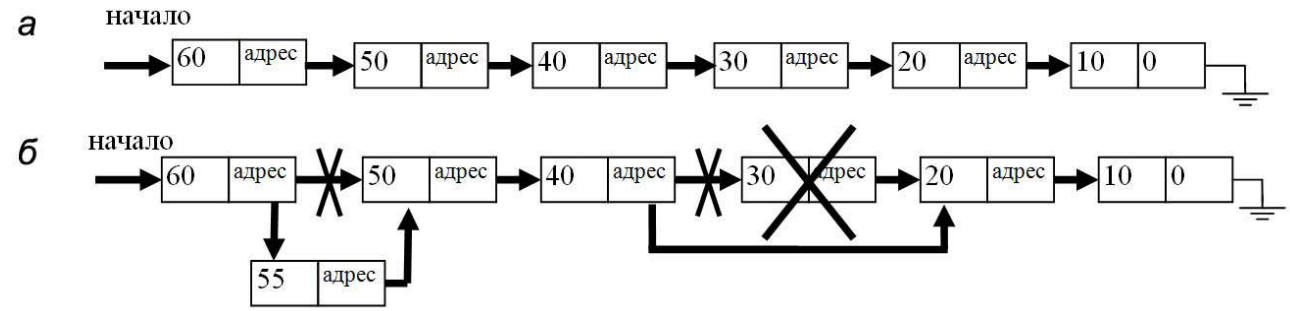


Рис. 49. Обратная очередь:
а – в связной форме организации; *б* – после удаления элемента 30
и добавления элемента 55

ГРАФИКА

Визуализация двумерной графики на платформе .NET выполняется с помощью типов из пространств имён, объединённых в GDI+ (*Graphics Device Interface* – интерфейс графических устройств). GDI+ является более удобной для программирования альтернативой GDI API (*Application Programming Interface* – программный интерфейс приложений). Ниже перечислены базовые пространства имён GDI+ (все они, разумеется, входят в корневое пространство имён `System`):

- `Drawing` – основные типы для основных операций визуализации (перья, кисти, шрифты). В этом пространстве объявлен основной тип графики – `Graphics`;
- `Drawing.Drawing2` – типы для более сложной двумерной векторной графики (градиентные кисти, стили концов стрелок, геометрические трансформации);
- `Drawing.Imaging` – типы для обработки графических изображений;
- `Drawing.Printing` – типы для отображения графики на странице предварительного просмотра и взаимодействия с принтером;
- `Drawing.Text` – типы для управления наборами шрифтов.

Основные типы пространства имён System.Drawing следующие:

Bitmap – битовая карта – расширяет класс Image дополнительными методами создания и манипулирования растровыми изображениями;

Brush – кисть (стандартные кисти содержатся в классе `Brushes`, который содержит свойства `readonly`, возвращающие объекты `Brush`: `Brushes.Red`, `Brushes.Orange`, `Brushes.Yellow`, `Brushes.Green`, ..., `Brushes.Transparent`) – абстрактный класс, который определяет объекты, используемые для заливки внутренних областей графических фигур. Не может быть реализован непосредственно. Для создания объекта «кисть» используются такие производные от него классы, как `SolidBrush` (кисть одного цвета), `Texture-Brush` (кисть из графического изображения), `LinearGradientBrush` (кисть с линейным или многоцветным градиентом);

Color – цвет. Стандартные цвета представляют многочисленные (141) константы, например:

Color.White=0xFFFFFFFF,
Color.WhiteSmoke=0xFFF5F5F5,
Color.Black=0xFF000000,
Color.BlanchedAlmond=0xFFFFEBCD,
Color.Blue=0xFF0000FF,
Color.BlueViolet=0xFF8A2BE2,
Color.Brown=0xFFA52A2A,
Color.Yellow=0xFFFFFF00,
Color.YellowGreen=0xFF9ACD32.

Объект класса `Color` представляет собой 32-разрядное значение, состоящее из четырёх байтовых компонент, соответственно для управления альфа-каналом, красным, зелёным и синим цветом (система ARGB). Альфа-компонент задаёт прозрачность (0 – прозрачен, 255 – непрозрачен) и определяет, в каких пропорциях данный цвет будет смешан с фоновым;

Font – шрифт – инкапсулирует набор характеристик шрифта: название, высоту, размер, стиль. Используется при прорисовке строк на графической поверхности;

Graphics – поверхность для рисования – дополнительно класс содержит методы и свойства, позволяющие эффективно манипулировать геометрическими объектами на поверхности рисования;

Icon – пиктограмма – инициализирует экземпляр класса `Icon` из указанного файла;

Image – базовый класс для различных форматов изображений – обеспечивает методы для загрузки и сохранения растровых и векторных изображений. Могут быть использованы такие форматы графических файлов, как bmp, icon, gif, jpeg, exif, png, tiff, wmf и emf;

Pen – карандаш – определяет объект, используемый для рисования прямых и кривых линий заданной ширины и указанного стиля. Нарисованную линию можно заполнить, используя различные стили заливки, включая сплошные цвета и текстуры. Стиль заливки зависит от кисти или текстуры, выбранной в качестве объекта заполнения;

Point – точка – представляет упорядоченную пару целых чисел – координат *X* и *Y*, определяющую точку на двумерной плоскости (структура);

Rectangle – прямоугольник – содержит набор из четырех целых чисел, определяющих расположение и размер прямоугольника;

Size – размер прямоугольной области – структура, содержащая пару чисел: ширину и высоту прямоугольника.

Особенности создания экземпляров графических типов

В силу особенностей работы графики создать экземпляр класса `Graphics` нельзя. Тем не менее существует три варианта обходного решения этой задачи:

1) объект для визуализации графических изображений возвращает метод `CreateGraphics()`, который имеется у всех объектов управления, производных от класса `Control`,

```
Graphics area = this.CreateGraphics();
```

2) объект экземпляра класса `Graphics` можно создать, взяв за основу файл с точечным изображением:

```
Bitmap bimg = new Bitmap("d:\\Catalog1\\pict1.bmp");
Graphics area = Graphics.FromImage(bimg);
```

3) ссылка на объект для рисования может быть получена из параметра `PaintEventArgs`, который передаётся в обработчик события `Paint`. Последнее генерируется всегда, когда какое-либо окно становится недействительным:

- переопределены размеры окна;
- окно или его часть открывается из-под другого окна;
- окно восстанавливается из значка панели задач
- и тому подобное.

Третий вариант является основным. После надлежащим образом выполненной регистрации метода `Form1_Paint` в событии `Paint` объект `e.Graphics` может быть использован следующим образом (рис. 50).

```
private void Form1_Paint(object sender, PaintEventArgs e)
     {Pen p = new Pen(Color.Green, 2);
      Rectangle rect = new Rectangle(10, 10, 200, 200);
      e.Graphics.DrawEllipse(p, rect);
      p.Dispose();
     }
```

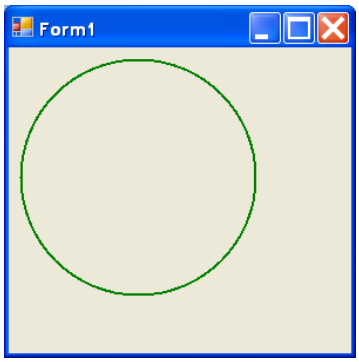


Рис. 50. Форма с окружностью

Место для рисования, полученное с помощью Graphics, совпадает с клиентской областью элемента управления (поверхность, которая остаётся после прорисовки заголовка, рамки и меню).

Графические объекты потребляют достаточно заметные системные ресурсы (например, память). Для того чтобы по возможности сократить продолжительность удержания ресурса (после того как он стал ненужным), *явным образом* (то есть с помощью new) созданные экземпляры рекомендуется освобождать с помощью метода Dispose. Правда, сделать это можно только для тех типов, которые поддерживают интерфейс IDisposable**.** В частности, тип Rectangle в приведённом на с. 200 примере этот интерфейс не поддерживает, и поэтому ускорить его уничтожение с помощью метода Dispose невозможно. То же самое освобождение памяти можно осуществить с помощью конструкции using:

```
private void Form1_Paint(object sender, PaintEventArgs e)
    {
     using (Pen p = new Pen(Color.Green, 2) )
       {
        Rectangle rect = new Rectangle(10, 10, 200, 200);
        e.Graphics.DrawEllipse(p, rect);
       }
    }
```

В данном случае при выходе потока выполнения за границу *блока* using метод Dispose вызывается автоматически.

В прил. 2 приведены примеры тестового задания.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Основы Windows-программирования на языке C# : методические указания / сост. О. М. Котов. Екатеринбург : УГТУ-УПИ, 2009. 40 с.

Павловская Т. А. C#. Программирование на языке высокого уровня : учебник для вузов / Т. А. Павловская. СПб. : Питер, 2009. 432 с.

Прайс Джейсон. Visual C# /NET. Полное руководство : пер. с англ. / Джейсон Прайс, Майк Гандерлой. Киев : ВЕК+ ; СПб. : КОРОНА принт ; Киев : НТИ ; М. : Энтроп, 2008. 960 с.

Троелсен Эндрю. Язык программирования C# 2008 и платформа .NET 3.5 Framework : пер. с англ. / Энрдю Троелсен. 4-е изд. М. : ВИЛЬЯМС, 2009. 1168 с.

ПРИЛОЖЕНИЯ

Приложение 1

Пример 1 тестового задания

А. Укажите неправильное утверждение.
1. Идентификатор может содержать не более одной цифры.
2. Идентификатор может содержать неограниченное количество символов.
3. Идентификатор не может совпадать с ключевым словом.
4. Идентификатор может содержать неограниченное количество цифр.

Б. Какого типа на языке C# не существует?
1. `ubyte`.
2. `int`.
3. `uint`.
4. `byte`.

В. Укажите правильное утверждение.
1. Размер типа bool составляет 1 байт.
2. Размер типа bool составляет 1 бит.
3. Размер типа bool составляет 2 байта.
4. Размер типа bool составляет 4 байт.

Г. Какое значение принимает переменная `a` в результате выполнения выражений

```
int a = 4 << 2 >> 1;
a *= 10 % 6;
```

1. 32.
2. 9.
3. 1.
4. 0.

Д. Что появится в консольном окне в результате выполнения фрагмента

```
for (short j = 1; j <= 4; Console.WriteLine(), j++)
    for (short I = 4; i >= j; i--) Console.Write ( j%i);
```

Номер ответа	1	2	3	4
Содержимое окна	1110 220 30 0	11 222 3333 44444	1111 222 11 0	10 120 1230 12340

E. Что появится в консольном окне в результате выполнения фрагмента

```
int [,] m = {{1,2,3},{4,5,6},{7,8,9}};
int c=0;
for (int i = 0; i < 3; i++)
for (int j = 0; j < i; j++) c += m[i,j];
Console.WriteLine(c);
```

1. 19	2. 17	3. 15	4. 12

Приложение 2

Пример 2 тестового задания

На перечисленные ниже теоретические и практические вопросы входного тестирования необходимо выбрать правильный ответ из четырёх предложенных. Практические вопросы необходимо дополнительно прокомментировать схемами объектов с содержимым, изменяющимся по ходу выполнения программы.

Если не установлен атрибут доступа, все элементы типа, объявленного с помощью оператора `class`, по умолчанию становятся:

1) public
2) protected
3) static
4) private

Система виртуальных методов реализуется в C# с помощью

5) операторов virtual и catch
6) операторов virtual и new
7) операторов virtual и override
8) операторов virtual и event

Наследование – это специальный механизм, который

9) позволяет производному классу иметь в своем составе все элементы базового класса
10) позволяет произвольному классу иметь в своем составе все элементы-данные и особые методы базового класса
11) позволяет производному классу получить доступ к общей области памяти
12) позволяет произвольному классу иметь в своем составе все элементы-данные и неособые методы базового класса

Статические поля класса

13) предназначены для хранения данных, которые остаются неизменными на протяжении всей программы
14) не могут быть унаследованы производным классом
15) всегда присутствуют в экземпляре класса скрытым образом
16) существуют в единственном числе, независимо от количества экземпляров класса

Исключение представляет собой

17) способ управления последовательностью вызовов методов в экземпляре-делегате
18) способ выборочного использования элементов базового класса
19) способ защиты данных от несанкционированного доступа
20) способ альтернативного возврата из функции

Что появится на экране:

```
class Program
 { class A
   {int Fa;
    static int Fb=0;
    public A(int a) { Fa = a; Fb+=1; }
    public A(int a, int b) { Fa = 2*a - b; Fb+=2;}
        public override string ToString()
    { return String.Format("{0}",Fa + Fb); }
    }
   static void Main()
{A[] m ={new A(1),new A(1,1),new A(2),new A(2,1)};
  foreach (A i in m) Console.Write(i);
 }
}
```

Ответы:

21) 7789
22) 1234
23) 9877
24) 5931

Что появится на экране:

```
class Program
{ class A
  {protected int Fa=1;
   protected A(int a) { Fa = a; }
   protected A() { Fa++  ; }
   protected int F{ get{return Fa;}}
  }
 class B : A
{ int Fb = 2;
public B() {Fb += 2; }
public B(int a): base(a) {Fb=2*a;}
public new int F
     { get{ return Fb + base.F;}}
public override string ToString()
 {returnString.Format("{0}",Fa+Fb);}
}
  static void Main()
  { B obj1 = new B(), obj2= new B(1);
       Console.Write(obj1);
       Console.Write(obj2);
     }
}
```

Ответы:

25) 63
26) 53
27) 43
28) 33

Правильные ответы 4, 7, 12, 16, 20, 21, 25.

ОГЛАВЛЕНИЕ