O'REILLY®

C# 8.0

Карманный справочник

Скорая помощь для программистов на C# 8.0

Джозеф Албахари
и Бен Албахари

C# 8.0

Карманный справочник

Скорая помощь для программистов на C# 8.0

C# 8.0 Pocket Reference

Instant Help for C# 8.0 Programmers

Joseph Albahari
and Ben Albahari

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY

C# 8.0

Карманный справочник

Скорая помощь для программистов на C# 8.0

Джозеф Албахари
и Бен Албахари

Диалектика

Москва · Санкт-Петербург
2020

ББК 32.973.26-018.2.75
А45
УДК 004.432

Компьютерное издательство "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *Ю.Н. Артеменко*

По общим вопросам обращайтесь в издательство "Диалектика"
по адресу: info@dialektika.com, http://www.dialektika.com

Албахари, Джозеф, **Албахари**, Бен.

А45 C# 8.0. Карманный справочник. : Пер. с англ. — СпБ. : ООО "Диалектика", 2020. — 240 с. : ил. — Парал. тит. англ.

ISBN 978-5-907203-14-3 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Джозеф Албахари, Бен Албахари

C# 8.0. Карманный справочник

Подписано в печать 16.12.2019. Формат 84×108/32.
Гарнитура Times.
Усл. печ. л. 9,68. Уч.-изд. л. 8,85.
Тираж 400 экз. Заказ № 362.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907203-14-3 (рус.)

ISBN 978-1-492-05121-3 (англ.)

Содержание

Об авторах

Джозеф Албахари — автор предыдущих изданий книг *C# in a Nutshell* (*C#. Справочник. Полное описание языка*) и *C# Pocket Reference* (*C#. Карманный справочник*), а также книги *LINQ Pocket Reference*. Он разработал LINQPad — популярную утилиту для подготовки кода и проверки запросов LINQ.

Бен Албахари — соучредитель веб-сайта Auditionist, предназначенного для кастинга актеров в Соединенном Королевстве. На протяжении пяти лет он был руководителем проектов в Microsoft, работая помимо прочего над проектами .NET Compact Framework и ADO.NET. Кроме того, он являлся соучредителем фирмы Genamics, занимающейся поставкой инструментов для программистов на C# и J++, а также программного обеспечения для анализа ДНК и протеиновых цепочек. Бен выступал соавтором книги *C# Essentials*, первой книги по языку C# от издательства O'Reilly, и предыдущих изданий *C# in a Nutshell*.

Карманный справочник по языку C# 8.0

C# является универсальным, безопасным в отношении типов, объектно-ориентированным языком программирования, целью которого является обеспечение продуктивности работы программистов. Для этого в языке соблюдается баланс между простотой, выразительностью и производительностью. Версия C# 8 рассчитана на работу с исполняемой средой Microsoft *.NET Core 3* и платформой .NET Standard 2.1 (тогда как версия *C# 7* была ориентирована на Microsoft *.NET Framework* 4.6/4.7/4.8, а также .NET Core 2.x и .NET Standard 2.0).

НА ЗАМЕТКУ!

Программы и фрагменты кода в этой книге соответствуют примерам, которые рассматриваются в главах 2–4 книги *C# 8.0. Справочник. Полное описание языка*, и доступны в виде интерактивных примеров для LINQPad (`http://www.linqpad.net/`). Проработка примеров в сочетании с чтением настоящей книги ускоряет процесс изучения, т.к. вы можете редактировать код и немедленно видеть результаты без необходимости в настройке проектов и решений в средеVisual Studio. Для загрузки примеров перейдите на вкладку Samples (Примеры) в окне LINQPad и щелкните на ссылке Download/import more samples (Загрузить/импортировать дополнительные примеры). Утилита LINQPad бесплатна и доступна для загрузки на веб-сайте `www.linqpad.net`.

Соглашения, используемые в этой книге

В книге приняты следующие типографские соглашения.

Курсив

Используется для новых терминов.

Моноширинный

Применяется для представления листингов программ, URL, адресов электронной почты, имен файлов и файловых расширений, а также внутри текста для ссылки на элементы программ, такие как имена переменных или функций, базы данных, типы данных, переменные среды, операторы и ключевые слова.

НА ЗАМЕТКУ!

Здесь приводится общее замечание.

Использование примеров кода

Программы и фрагменты кода, рассмотренные в этой книге, доступны в виде интерактивных примеров для LINQPad.

Данная книга призвана помочь вам делать свою работу. В общем случае вы можете применять приведенный здесь код примеров в своих программах и документации. Вы не обязаны спрашивать у нас разрешения, если только не воспроизводите значительную часть кода. Например, для написания программы, в которой встречаются многие фрагменты кода из этой книги, разрешение не требуется. Однако для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение обязательно. Ответ на вопрос путем ссылки на эту книгу и цитирования кода из примера разрешения не требует. Но для внедрения существенного объема кода примеров, предлагаемых в этой книге, в документацию по вашему продукту разрешение обязательно.

Мы высоко ценим указание авторства, хотя и не требуем этого. Установление авторства обычно включает название книги, фамилии и имена авторов, издательство и номер ISBN. Например: "*C# 8.0 Pocket Reference* by Joseph Albahari and Ben Albahari (O'Reilly). Copyright 2020 Joseph Albahari, Ben Albahari, 978-1-492-05121-3".

Если вам кажется, что способ использования вами примеров кода выходит за законные рамки или упомянутые выше разрешения, тогда свяжитесь с нами по следующему адресу электронной почты: `permissions@oreilly.com`.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`
WWW: `http://www.dialektika.com`

Первая программа на C#

Ниже показана программа, которая умножает 12 на 30 и выводит на экран результат 360. Двойная косая черта указывает на то, что остаток строки является *комментарием*:

```
using System;                       // Импортирование
                                    //   пространства имен
class Test                          // Объявление класса
{
  static void Main()                // Объявление метода
  {
    int x = 12 * 30;                // Оператор 1
    Console.WriteLine (x);          // Оператор 2
  }                                 // Конец метода
}                                   // Конец класса
```

Основным компонентом программы являются два *оператора*. Операторы в C# выполняются последовательно и завершаются точкой с запятой. Первый оператор вычисляет *выражение* `12 * 30` и сохраняет результат в *локальной переменной* по имени `x`, которая имеет целочисленный тип. Второй оператор вызывает *метод* `WriteLine()` класса `Console` для вывода значения переменной `x` в текстовое окно на экране.

Метод выполняет действие в виде последовательности операторов, которая называется *блоком операторов* и представляет собой пару фигурных скобок, содержащих ноль или большее количество операторов. Мы определили единственный метод по имени `Main()`.

Написание высокоуровневых функций, которые вызывают низкоуровневые функции, упрощает программу. Мы можем провести *рефакторинг* программы, выделив многократно используемый метод, который умножает целое число на 12, следующим образом:

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));   // 360
    Console.WriteLine (FeetToInches (100)); // 1200
  }
```

```
  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Метод может получать *входные* данные из вызывающего кода за счет указания параметров и передавать *выходные* данные обратно вызывающему коду путем указания *возвращаемого типа*. В предыдущем примере мы определили метод по имени `FeetToInches()`, который имеет параметр для входного значения в футах и возвращаемый тип для выходного значения в дюймах; оба они принадлежат к (целочисленному) типу `int`.

Аргументами, передаваемыми методу `FeetToInches()`, являются *литералы* `30` и `100`. Метод `Main()` в этом примере содержит пустые круглые скобки, поскольку он не имеет параметров, и является `void`, потому что не возвращает какого-либо значения вызывающему коду. Метод по имени `Main()` распознается в C# как признак стандартной точки входа в поток выполнения. Метод `Main()` может необязательно возвращать целочисленное значение (вместо `void`) с целью его передачи исполняющей среде. Кроме того, метод `Main()` способен дополнительно принимать в качестве параметра массив строк (который будет заполняться любыми аргументами, передаваемыми исполняемому файлу). Например:

```
static int Main (string[] args) {...}
```

НА ЗАМЕТКУ!

Массив (такой как `string[]`) представляет фиксированное количество элементов определенного типа (см. раздел “Массивы” на стр. 39).

Методы являются одним из нескольких видов функций в C#. Другим видом функции, который мы применяли, была операция `*`, выполняющая умножение. Существуют также *конструкторы*, *свойства*, *события*, *индексаторы* и *финализаторы*.

В рассматриваемом примере два метода сгруппированы в класс. *Класс* объединяет функции-члены и данные-члены для

формирования объектно-ориентированного строительного блока. Класс `Console` группирует члены, которые поддерживают функциональность ввода-вывода в командной строке, такие как метод `WriteLine()`. Наш класс `Test` объединяет два метода — `Main()` и `FeetToInches()`. Класс представляет собой разновидность *типа* (типы мы обсудим в разделе "Основы типов" на стр. 17).

На самом внешнем уровне программы типы организованы в *пространства имен*. Директива `using` делает пространство имен `System` доступным нашему приложению для использования класса `Console`. Мы могли бы определить все свои классы внутри пространства имен `TestPrograms`, как показано ниже:

```
using System;

namespace TestPrograms
{
  class Test {...}
  class Test2 {...}
}
```

Библиотеки .NET Core организованы в виде вложенных пространств имен. Например, следующее пространство имен содержит типы для обработки текста:

```
using System.Text;
```

Директива `using` присутствует здесь ради удобства; на тип можно также ссылаться с помощью полностью заданного имени, которое представляет собой имя типа, предваренное его пространством имен, наподобие `System.Text.StringBuilder`.

Компиляция

Компилятор C# транслирует исходный код, указываемый в виде набора файлов с расширением `.cs`, в *сборку*, которая представляет собой единицу упаковки и развертывания в .NET. Сборка может быть либо *приложением*, либо *библиотекой*; разница между ними заключается в том, что приложение имеет точку входа (метод `Main()`), тогда как библиотека — нет. Библиотека предназначена для вызова (*ссылки*) приложением или другими библиотеками. Исполняющая среда .NET Core и инфраструктура .NET Framework состоят из набора библиотек.

Для вызова компилятора можно либо применять интегрированную среду разработки (integrated development environment — IDE), такую как Visual Studio или Visual Studio Code, либо запускать его вручную в командной строке. Чтобы вручную скомпилировать консольное приложение с .NET Core, сначала понадобится загрузить комплект .NET Core SDK, после чего создать новый проект:

```
dotnet new console -o MyFirstProgram
cd MyFirstProgram
```

В результате создается папка по имени `MyFirstProgram`, содержащая файл с кодом на C# под названием `Program.cs`, который затем вы можете редактировать. Для вызова компилятора введите команду `dotnet build` (или команду `dotnet run`, которая скомпилирует и запустит программу). Вывод записывается в подкаталог под `bin\debug` и включает файл `MyFirstProgram.dll` (выходная сборка), а также файл `MyFirstProgram.exe` (напрямую запускающий скомпилированную программу).

Синтаксис

На синтаксис C# оказал влияние синтаксис языков C и C++. В этом разделе мы опишем элементы синтаксиса C#, применяя в качестве примера следующую программу:

```
using System;
class Test
{
  static void Main()
  {
    int x = 12 * 30;
    Console.WriteLine (x);
  }
}
```

Идентификаторы и ключевые слова

Идентификаторы — это имена, которые программисты выбирают для своих классов, методов, переменных и т.д. Ниже перечислены идентификаторы в примере программы в порядке их появления:

```
System   Test   Main   x   Console   WriteLine
```

Идентификатор должен быть целостным словом, которое по существу состоит из символов Unicode и начинается с буквы или символа подчеркивания. Идентификаторы C# чувствительны к регистру символов. По соглашению для параметров, локальных переменных и закрытых полей должен применяться "верблюжий" стиль (например, `myVariable`), а для всех остальных идентификаторов — стиль Pascal (скажем, `MyMethod`).

Ключевые слова представляют собой имена, которые имеют для компилятора особый смысл. Ниже перечислены ключевые слова в примере программы:

```
using  class  static  void  int
```

Большинство ключевых слов являются *зарезервированными*, т.е. их нельзя использовать в качестве идентификаторов. Вот полный список зарезервированных ключевых слов C#:

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

Избегание конфликтов

Если вы действительно хотите применять идентификатор с именем, которое конфликтует с ключевым словом, то к нему необходимо добавить префикс `@`. Например:

```
class class  {...}  // Не допускается
class @class {...}  // Разрешено
```

Символ @ не является частью самого идентификатора. Таким образом, `@myVariable` — то же самое, что и `myVariable`.

Контекстные ключевые слова

Некоторые ключевые слова являются *контекстными*, а это значит, что их также можно использовать в качестве идентификаторов — без символа @. Такие ключевые слова перечислены ниже:

add	dynamic	join	select
alias	equals	let	set
ascending	from	nameof	value
async	get	on	var
await	global	orderby	when
by	group	partial	where
descending	into	remove	yield

Неоднозначность с контекстными ключевыми словами не может возникать внутри контекста, в котором они применяются.

Литералы, знаки пунктуации и операции

Литералы — это элементарные порции данных, лексически встраиваемые в программу. В рассматриваемом примере программы используются литералы `12` и `30`. *Знаки пунктуации* помогают размечать структуру программы. В примере программы присутствуют знаки пунктуации `{`, `}` и `;`.

Фигурные скобки группируют множество операторов в *блок операторов*. Точка с запятой завершает оператор (но не блок операторов). Операторы могут охватывать несколько строк:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Операция преобразует и объединяет выражения. Большинство операций в C# обозначаются с помощью некоторого символа, например, операция умножения выглядит как *. Вот какие операции задействованы в примере программы:

```
.   ()   *   =
```

Точкой обозначается членство (или десятичная точка в числовых литералах). Круглые скобки в примере присутствуют там, где объявляется или вызывается метод; пустые круглые скоб-

ки означают, что метод не принимает аргументов. Знак "равно" выполняет *присваивание* (двойной знак "равно", ==, производит сравнение эквивалентности).

Комментарии

В C# поддерживаются два разных стиля документирования исходного кода: *однострочные комментарии* и *многострочные комментарии*. Однострочный комментарий начинается с двойной косой черты и продолжается до конца строки. Например:

```
int x = 3;  // Комментарий относительно
            //  присваивания переменной x значения 3
```

Многострочный комментарий начинается с символов `/*` и заканчивается символами `*/`. Например:

```
int x = 3;  /* Это комментарий,
               который занимает две строки */
```

В комментарии можно встраивать XML-дескрипторы документации (см. раздел "XML-документация" на стр. 231).

Основы типов

Тип определяет шаблон для значения. В рассматриваемом примере мы применяем два литерала типа `int` со значениями `12` и `30`. Мы также объявляем *переменную* типа `int` по имени `x`.

Переменная обозначает ячейку в памяти, которая с течением времени может содержать разные значения. Напротив, *константа* всегда представляет одно и то же значение (подробнее об этом — позже).

Все значения в C# являются *экземплярами* специфического типа. Смысл значения и набор возможных значений, которые может иметь переменная, определяются ее типом.

Примеры предопределенных типов

Предопределенные типы (также называемые *встроенными* типами) — это типы, которые специально поддерживаются компилятором. Тип `int` является предопределенным типом для представления набора целых чисел, которые умещаются в 32 бита памяти, от -2^{31} до $2^{31}-1$.

С экземплярами типа int можно выполнять операции, например, арифметические:

```
int x = 12 * 30;
```

Еще одним предопределенным типом в C# является string. Тип string представляет последовательность символов, такую как ".NET" или "http://oreilly.com". Со строками можно работать, вызывая на них функции следующим образом:

```
string message = "Добро пожаловать";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);  // ДОБРО ПОЖАЛОВАТЬ
int x = 2020;
message = message + x.ToString();
Console.WriteLine (message);   // Добро пожаловать2020
```

Предопределенный тип bool поддерживает в точности два возможных значения: true и false. Тип bool обычно используется для условного разветвления потока выполнения с помощью оператора if. Например:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("Это не будет выведено");
int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("Это будет выведено");
```

НА ЗАМЕТКУ!

Пространство имен System в .NET Core содержит много важных типов, которые не являются предопределенными в языке C# (скажем, DateTime).

Примеры специальных типов

Точно так же, как из простых функций можно строить сложные функции, из элементарных типов допускается создавать сложные типы. В следующем примере мы определим специальный тип по имени UnitConverter — класс, который служит шаблоном для преобразования единиц:

```
using System;
public class UnitConverter
{
  int ratio;                                  // Поле
  public UnitConverter (int unitRatio)  // Конструктор
  {
    ratio = unitRatio;
  }
  public int Convert (int unit)           // Метод
  {
    return unit * ratio;
  }
}
class Test
{
  static void Main()
  {
    UnitConverter feetToInches = new UnitConverter(12);
    UnitConverter milesToFeet = new UnitConverter(5280);
    Console.Write (feetToInches.Convert(30));  // 360
    Console.Write (feetToInches.Convert(100)); // 1200
    Console.Write (feetToInches.Convert
                  (milesToFeet.Convert(1)));     // 63360
  }
}
```

Члены типа

Тип содержит *данные-члены* и *функции-члены*. Данными-членами типа `UnitConverter` является *поле* по имени `ratio`. Функции-члены типа `UnitConverter` — это метод `Convert()` и *конструктор* класса `UnitConverter`.

Гармония предопределенных и специальных типов

Привлекательный аспект языка C# заключается в том, что между предопределенными и специальными типами имеется совсем мало отличий. Предопределенный тип `int` служит шаблоном для целых чисел. Он содержит данные — 32 бита — и предоставляет функции-члены, использующие эти данные, такие как `ToString()`. Аналогичным образом наш специальный тип `UnitConverter` действует в качестве шаблона для преобразования единиц. Он хранит данные — коэффициент (`ratio`) — и предлагает функции-члены для работы с этими данными.

Конструкторы и создание экземпляров

Данные создаются путем *создания экземпляров* типа. Мы можем создавать экземпляры предопределенных типов просто за счет применения литерала, такого как `12` или `"Добро пожаловать"`.

Операция `new` создает экземпляры специального типа. Наш метод `Main()` начинается с создания двух экземпляров типа `UnitConverter`. Непосредственно после создания операцией `new` объекта вызывается его *конструктор* для выполнения инициализации. Конструктор определяется подобно методу за исключением того, что вместо имени метода и возвращаемого типа указывается имя типа, которому конструктор принадлежит:

```
public UnitConverter (int unitRatio)  // Конструктор
{
  ratio = unitRatio;
}
```

Члены экземпляра и статические члены

Данные-члены и функции-члены, которые оперируют на *экземпляре* типа, называются членами экземпляра. Примерами членов экземпляра могут служить метод `Convert()` типа `UnitConverter` и метод `ToString()` типа `int`. По умолчанию члены являются членами экземпляра.

Данные-члены и функции-члены, которые не оперируют на экземпляре типа, а взамен имеют дело с самим типом, должны помечаться как статические (`static`). Статическими методами являются `Test.Main()` и `Console.WriteLine()`. Класс `Console` в действительности представляет собой *статический класс*, т.е. *все* его члены статические. Создавать экземпляры класса `Console` на самом деле никогда не придется — одна консоль используется целым приложением.

Давайте сравним члены экземпляра и статические члены. В показанном ниже коде поле экземпляра `Name` относится к конкретному экземпляру `Panda`, тогда как поле `Population` принадлежит набору всех экземпляров класса `Panda`:

```
public class Panda
{
  public string Name;           // Поле экземпляра
  public static int Population; // Статическое поле

  public Panda (string n)       // Конструктор
```

```
    {
      Name = n;     // Присвоить значение полю экземпляра
      Population = Population+1;
          // Инкрементировать значение статического поля
    }
}
```

В следующем коде создаются два экземпляра `Panda`, выводятся их имена (`Name`) и затем общее количество (`Population`):

```
Panda p1 = new Panda ("Панда Ди");
Panda p2 = new Panda ("Панда Да");

Console.WriteLine (p1.Name);            // Панда Ди
Console.WriteLine (p2.Name);            // Панда Да

Console.WriteLine (Panda.Population);   // 2
```

Ключевое слово `public`

Ключевое слово `public` открывает доступ к членам со стороны других классов. Если бы в рассматриваемом примере поле `Name` класса `Panda` не было помечено как `public`, то оно оказалось бы закрытым и класс `Test` не смог бы получить к нему доступ. Маркировка члена как открытого (`public`) означает, что тип разрешает его видеть другим типам, а все остальное будет относиться к закрытым деталям реализации. В рамках объектно-ориентированной терминологии мы говорим, что открытые члены *инкапсулируют* закрытые члены класса.

Преобразования

В языке C# возможны преобразования между экземплярами совместимых типов. Преобразование всегда приводит к созданию нового значения из существующего. Преобразования могут быть либо *неявными*, либо *явными*: неявные преобразования происходят автоматически, в то время как явные преобразования требуют *приведения*. В следующем примере мы *неявно* преобразовываем `int` в тип `long` (который имеет в два раза больше битов, чем `int`) и *явно* приводим `int` к типу `short` (имеющий вполовину меньше битов, чем `int`):

```
int x = 12345;     // int - это 32-битное целое
long y = x;  // Неявное преобразование в 64-битное целое
short z = (short)x; // Явное приведение к 16-битному целому
```

В общем случае неявные преобразования разрешены, когда компилятор в состоянии гарантировать, что они всегда будут проходить успешно без потери информации. В других обстоятельствах для преобразования между совместимыми типами должно выполняться явное приведение.

Типы значений и ссылочные типы

Типы в C# можно разделить на *типы значений* и *ссылочные типы*.

Типы значений включают большинство встроенных типов (а именно — все числовые типы, тип `char` и тип `bool`), а также специальные типы `struct` и `enum`. *Ссылочные типы* включают все классы, массивы, делегаты и интерфейсы.

Фундаментальное отличие между типами значений и ссылочными типами связано с тем, как они поддерживаются в памяти.

Типы значений

Содержимым переменной или константы, относящейся к *типу значения*, является просто значение. Например, содержимое встроенного типа значения `int` — это 32 бита данных.

С помощью ключевого слова `struct` можно определить специальный тип значения (рис. 1):

```
public struct Point { public int X, Y; }
```

Рис. 1. Экземпляр типа значения в памяти

Присваивание экземпляра типа значения всегда приводит к *копированию* этого экземпляра. Например:

```
Point p1 = new Point();
p1.X = 7;

Point p2 = p1;  // Присваивание приводит к копированию

Console.WriteLine (p1.X);  // 7
Console.WriteLine (p2.X);  // 7
```

```
p1.X = 9;              // Изменить p1.X
Console.WriteLine (p1.X);  // 9
Console.WriteLine (p2.X);  // 7
```

На рис. 2 видно, что экземпляры p1 и p2 хранятся независимо друг от друга.

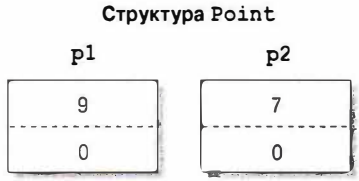


Рис. 2. Присваивание копирует экземпляр типа значения

Ссылочные типы

Ссылочный тип сложнее типа значения из-за наличия двух частей: *объекта* и *ссылки* на этот объект. Содержимым переменной или константы ссылочного типа является ссылка на объект, который содержит значение. Ниже приведен тип Point из предыдущего примера, переписанный в виде класса (рис. 3):

```
public class Point { public int X, Y; }
```

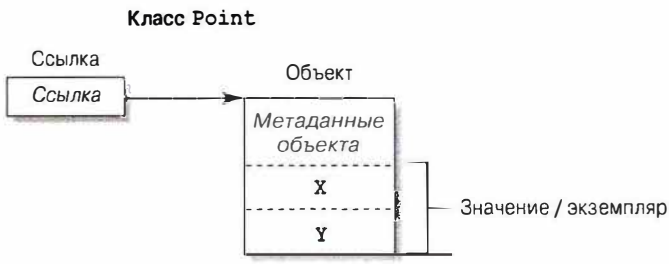


Рис. 3. Экземпляр ссылочного типа в памяти

Присваивание переменной ссылочного типа вызывает копирование ссылки, но не экземпляра объекта. Это позволяет множеству переменных ссылаться на один и тот же объект — то, что с типами значений обычно невозможно. Если повторить предыдущий пример при условии, что Point теперь представляет собой класс, тогда операция над p1 будет воздействовать на p2:

```
Point p1 = new Point();
p1.X = 7;
Point p2 = p1;              // Копирует ссылку на p1
Console.WriteLine (p1.X);  // 7
Console.WriteLine (p2.X);  // 7

p1.X = 9;                   // Изменить p1.X
Console.WriteLine (p1.X);  // 9
Console.WriteLine (p2.X);  // 9
```

На рис. 4 видно, что p1 и p2 — это две ссылки, которые указывают на один и тот же объект.

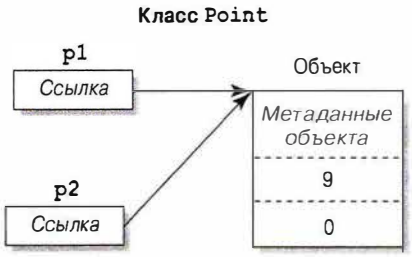


Рис. 4. *Присваивание копирует ссылку*

Значение null

Ссылке может быть присвоен литерал null, который отражает тот факт, что ссылка не указывает на какой-либо объект. Предположим, что Point является классом:

```
Point p = null;
Console.WriteLine (p == null);  // True
```

Обращение к члену ссылки null приводит к возникновению ошибки времени выполнения:

```
Console.WriteLine (p.X);  // Генерация исключения
                          //   NullReferenceException
```

НА ЗАМЕТКУ!

В C# 8 появилось новое средство, призванное уменьшить случайные ошибки NullReferenceException. Дополнительные сведения ищите в разделе “Ссылочные типы, допускающие null (C# 8)” на стр. 163.

В противоположность этому тип значения обычно не может иметь значение null:

```
struct Point {...}
...
Point p = null;  // Ошибка на этапе компиляции
int x = null;    // Ошибка на этапе компиляции
```

НА ЗАМЕТКУ!

В C# имеется специальная конструкция под названием *типы, допускающие значение* `null`, которая предназначена для представления `null` в типах значений. Дополнительные сведения ищите в разделе "Типы (значений), допускающие null" на стр. 158.

Классификация предопределенных типов

Предопределенные типы в C# классифицируются следующим образом.

Типы значений

- Числовой
  - Целочисленный со знаком (`sbyte`, `short`, `int`, `long`)
  - Целочисленный без знака (`byte`, `ushort`, `uint`, `ulong`)
  - Вещественный (`float`, `double`, `decimal`)
- Логический (`bool`)
- Символьный (`char`)

Ссылочные типы

- Строковый (`string`)
- Объектный (`object`)

Предопределенные типы в C# являются псевдонимами типов .NET Core в пространстве имен `System`. Показанные ниже два оператора отличаются только синтаксисом:

```
int i = 5;
System.Int32 i = 5;
```

Набор предопределенных типов *значений*, исключая `decimal`, в общеязыковой исполняющей среде (Common Language Runtime — CLR) известен как *примитивные типы*. Примитивные типы называются так потому, что они поддерживаются непосредственно через инструкции в скомпилированном коде, которые обычно транслируются в прямую поддержку внутри имеющегося процессора.

Числовые типы

Ниже показаны предопределенные числовые типы в C#.

Тип C#	Тип в `System`	Суффикс	Размер в битах	Диапазон
Целочисленный со знаком				
`sbyte`	`SByte`		8	$-2^7 – 2^7-1$
`short`	`Int16`		16	$-2^{15} – 2^{15}-1$
`int`	`Int32`		32	$-2^{31} – 2^{31}-1$
`long`	`Int64`	`L`	64	$-2^{63} – 2^{63}-1$
Целочисленный без знака				
`byte`	`Byte`		8	$0 – 2^8-1$
`ushort`	`UInt16`		16	$0 – 2^{16}-1$
`uint`	`UInt32`	`U`	32	$0 – 2^{32}-1$
`ulong`	`UInt64`	`UL`	64	$0 – 2^{64}-1$
Вещественный				
`float`	`Single`	`F`	32	$\pm(\sim10^{-45} – 10^{38})$
`double`	`Double`	`D`	64	$\pm(\sim10^{-324} – 10^{308})$
`decimal`	`Decimal`	`M`	128	$\pm(\sim10^{-28} – 10^{28})$

Из всех *целочисленных* типов `int` и `long` являются первоклассными типами, которым обеспечивается поддержка как в языке C#, так и в исполняющей среде. Другие целочисленные типы обычно применяются для реализации взаимодействия или когда главная задача связана с эффективностью хранения.

В рамках *вещественных* числовых типов `float` и `double` называются *типами с плавающей точкой* и обычно используются в научных и графических вычислениях. Тип `decimal` применяется, как правило, в финансовых вычислениях, когда требуется десятичная арифметика и высокая точность. (Формально `decimal` также является типом с плавающей точкой, хотя обычно на него так не ссылаются.)

Числовые литералы

Целочисленные литералы могут использовать десятичную или шестнадцатеричную форму записи; шестнадцатеричная форма записи предусматривает применение префикса `0x` (например, `0x7f` эквивалентно `127`). Начиная с версии C# 7, можно также использовать префикс `0b` для двоичных литералов. В *вещественных литералах* может применяться десятичная и/или экспоненциальная форма записи, такая как `1E06`.

Начиная с версии C# 7, для улучшения читабельности внутрь числового литерала могут вставляться символы подчеркивания (например, `1_000_000`).

Выведение типа числового литерала

По умолчанию компилятор *выводит* тип числового литерала, относя его либо к `double`, либо к какому-то целочисленному типу.

- Если литерал содержит десятичную точку или символ экспоненты (`E`), то он получает тип `double`.
- В противном случае типом литерала будет первый тип, в который может уместиться значение литерала, из следующего списка: `int`, `uint`, `long` и `ulong`.

Например:

```
Console.Write (    1.0.GetType()); // Double (double)
Console.Write (   1E06.GetType()); // Double (double)
Console.Write (      1.GetType()); // Int32  (int)
Console.Write ( 0xF0000000.GetType()); // UInt32 (uint)
Console.Write (0x100000000.GetType()); // Int64  (long)
```

Числовые суффиксы

Числовые суффиксы, указанные в предыдущей таблице, явно определяют тип литерала:

```
decimal d = 3.5M; //M = decimal (не чувствителен к регистру)
```

Необходимость в суффиксах U и L возникает редко, поскольку типы uint, long и ulong почти всегда могут либо *выводиться*, либо *неявно преобразовываться* из int:

```
long i = 5;  // Неявное преобразование из int в long
```

Суффикс D формально избыточен из-за того, что все литералы с десятичной точкой выводятся в double (и к числовому литералу всегда можно добавить десятичную точку). Суффиксы F и M наиболее полезны и обязательны при указании дробных литералов float или decimal. Без суффиксов следующий код не скомпилируется, т.к. литерал 4.5 был бы выведен в тип double, для которого не предусмотрено неявное преобразование в float или decimal:

```
float f = 4.5F;  // Не будет компилироваться без суффикса

decimal d = -1.23M;   // Не будет компилироваться
                      //   без суффикса
```

Числовые преобразования

Преобразования целых чисел в целые числа

Целочисленные преобразования являются *неявными*, когда целевой тип в состоянии представить каждое возможное значение исходного типа. Иначе требуется *явное* преобразование. Например:

```
int x = 12345;        // int - 32-битное целое

long y = x;           // Неявное преобразование
                      //   в 64-битное целое

short z = (short)x;   // Явное преобразование
                      //   в 16-битное целое
```

Преобразования чисел с плавающей точкой в числа с плавающей точкой

Тип `float` может быть неявно преобразован в тип `double`, т.к. `double` позволяет представить любое возможное значение `float`. Обратное преобразование должно быть явным.

Преобразования между `decimal` и другими вещественными типами должны быть явными.

Преобразования чисел с плавающей точкой в целые числа

Преобразования целочисленных типов в вещественные типы являются неявными, тогда как обратные преобразования должны быть явными. Преобразование числа с плавающей точкой в целое число усекает дробную часть; для выполнения преобразований с округлением следует применять статический класс `System.Convert`.

Важно знать, что неявное преобразование большого целочисленного типа в тип с плавающей точкой сохраняет *величину*, но иногда может приводить к потере *точности*:

```
int i1 = 100000001;
float f = i1; // Величина сохраняется, точность теряется
int i2 = (int)f;  // 100000000
```

Арифметические операции

Арифметические операции (`+`, `-`, `*`, `/`, `%`) определены для всех числовых типов кроме 8- и 16-битных целочисленных типов. Операция `%` дает остаток от деления.

Операции инкремента и декремента

Операции инкремента и декремента (соответственно `++`, `--`) увеличивают и уменьшают значения числовых типов на 1. Такая операция может находиться перед или после переменной в зависимости от того, когда требуется обновить значение переменной — до или после вычисления выражения. Например:

```
int x = 0;
Console.WriteLine (x++); // Выводит 0; x теперь содержит 1
Console.WriteLine (++x); // Выводит 2; x теперь содержит 2
Console.WriteLine (--x); // Выводит 1; x теперь содержит 1
```

Специальные целочисленные операции

Деление

Операции деления на целочисленных типах всегда усекают остатки (округляют в направлении нуля). Деление на переменную, значение которой равно нулю, генерирует ошибку во время выполнения (исключение `DivideByZeroException`). Деление на *литерал* или *константу*, равную нулю, вызывает ошибку на этапе компиляции.

Переполнение

Во время выполнения арифметические операции на целочисленных типах могут приводить к переполнению. По умолчанию это происходит молча — никакие исключения не генерируются, а результат демонстрирует поведение с циклическим возвратом, как если бы вычисление производилось над большим целочисленным типом с отбрасыванием дополнительных значащих битов. Например, декрементирование минимально возможного значения типа `int` дает в результате максимально возможное значение `int`:

```
int a = int.MinValue; a--;
Console.WriteLine (a == int.MaxValue);  // True
```

Операции checked и unchecked

Операция `checked` сообщает исполняющей среде о том, что вместо молчаливого переполнения она должна генерировать исключение `OverflowException`, когда целочисленное выражение или оператор выходит за арифметические пределы данного типа. Операция `checked` воздействует на выражения с операциями `++`, `--`, `-` (унарная), `+`, `-`, `*`, `/` и явными преобразованиями между целочисленными типами. С проверкой переполнения связаны небольшие накладные расходы в плане производительности.

Операцию `checked` можно использовать либо с выражением, либо с блоком операторов. Например:

```
int a = 1000000, b = 1000000;
int c = checked (a * b); // Проверяет только это выражение.
```

```
checked                    // Проверяет все выражения
{                          // в блоке операторов.
  c = a * b;
  ...
}
```

Проверку на арифметическое переполнение можно сделать обязательной для всех выражений в программе, скомпилировав ее с переключателем командной строки /checked+ (в Visual Studio это делается на вкладке Advanced Build Settings (Дополнительные параметры сборки)). Если позже понадобится отключить проверку переполнения для конкретных выражений или операторов, то можно применить операцию unchecked.

Побитовые операции

В C# поддерживаются следующие побитовые операции.

Операция	Описание	Пример выражения	Результат
~	Дополнение	~0xfU	0xfffffff0U
&	И	0xf0 & 0x33	0x30
|	ИЛИ	0xf0 | 0x33	0xf3
^	Исключающее ИЛИ	0xff00 ^ 0x0ff0	0xf0f0
<<	Сдвиг влево	0x20 << 2	0x80
>>	Сдвиг вправо	0x20 >> 1	0x10

8- и 16-битные целочисленные типы

К 8- и 16-битным целочисленным типам относятся byte, sbyte, short и ushort. У таких типов отсутствуют собственные арифметические операции, поэтому в C# они при необходимости неявно преобразуются в более крупные типы. Попытка присваивания результата переменной меньшего целочисленного типа может вызвать ошибку на этапе компиляции:

```
short x = 1, y = 1;
short z = x + y;  // Ошибка на этапе компиляции
```

В данном случае переменные x и y неявно преобразуются в тип int, так что сложение может быть выполнено. Это означает, что результат также будет иметь тип int, который не может

быть неявно приведен к типу short (из-за возможной потери информации). Чтобы показанный код скомпилировался, потребуется добавить явное приведение:

```
short z = (short) (x + y);  // Компилируется
```

Специальные значения float и double

В отличие от целочисленных типов типы с плавающей точкой имеют значения, которые определенные операции трактуют особым образом. Такими специальными значениями являются NaN (Not a Number — не число), +∞, −∞ и −0. В классах float и double предусмотрены константы для NaN, +∞ и −∞ (а также для других значений, включая MaxValue, MinValue и Epsilon). Например:

```
Console.Write (double.NegativeInfinity); // -бесконечность
```

Деление ненулевого числа на ноль дает в результате бесконечную величину. Например:

```
Console.WriteLine ( 1.0 / 0.0);     // бесконечность
Console.WriteLine (-1.0 / 0.0);     // -бесконечность
Console.WriteLine ( 1.0 / -0.0);    // -бесконечность
Console.WriteLine (-1.0 / -0.0);    // бесконечность
```

Деление нуля на ноль или вычитание бесконечности из бесконечности дает в результате NaN. Например:

```
Console.Write ( 0.0 / 0.0);                 // NaN
Console.Write ((1.0 / 0.0) - (1.0 / 0.0));  // NaN
```

Когда применяется операция ==, значение NaN никогда не будет равно другому значению, даже NaN. Для проверки, равно ли значение специальному значению NaN, должен использоваться метод float.IsNaN() или double.IsNaN():

```
Console.WriteLine (0.0 / 0.0 == double.NaN);   // False
Console.WriteLine (double.IsNaN (0.0 / 0.0));  // True
```

Однако в случае применения метода object.Equals() два значения NaN равны:

```
bool isTrue = object.Equals (0.0/0.0, double.NaN);
```

Выбор между double и decimal

Тип double удобен в научных вычислениях (таких как вычисление пространственных координат), а тип decimal — в финансовых вычислениях и для представления значений, которые являются "искусственными", а не полученными в результате реальных измерений. Ниже представлена сводка по отличиям между типами double и decimal.

Характеристика	double	decimal
Внутреннее представление	Двоичное	Десятичное
Точность	15–16 значащих цифр	28–29 значащих цифр
Диапазон	$\pm(\sim10^{-324}$ — $\sim10^{308})$	$\pm(\sim10^{-28}$ — $\sim10^{28})$
Специальные значения	+0, –0, +∞, –∞ и NaN	Отсутствуют
Скорость обработки	Присущая процессору	Не присущая процессору (примерно в 10 раз медленнее, чем double)

Ошибки округления вещественных чисел

Типы float и double внутренне представляют числа в двоичной форме. По этой причине большинство литералов с дробной частью (которые являются десятичными) не будут представлены точно:

```
float tenth = 0.1f;                    // Не точно 0.1
float one = 1f;
Console.WriteLine (one - tenth * 10f); // -1.490116E-08
```

Именно потому типы float и double не подходят для финансовых вычислений. В противоположность им тип decimal работает в десятичной системе счисления и способен точно представлять дробные числа вроде 0.1 (десятичное представление которого является непериодическим).

Булевские типы и операции

Тип bool в C# (псевдоним типа System.Boolean) представляет логическое значение, которому может быть присвоен литерал true или false.

Хотя для хранения булевского значения достаточно только одного бита, исполняющая среда будет использовать один байт памяти, поскольку это минимальная порция, с которой исполняющая среда и процессор могут эффективно работать. Во избежание непродуктивных затрат пространства в случае массивов инфраструктура .NET предлагает в пространстве имен `System.Collections` класс `BitArray`, который позволяет задействовать по одному биту для каждого булевского значения в массиве.

Операции эквивалентности и сравнения

Операции `==` и `!=` проверяют на предмет эквивалентности и неэквивалентности значения любого типа и всегда возвращают значение `bool`. Типы значений обычно поддерживают очень простое понятие эквивалентности:

```
int x = 1, y = 2, z = 1;
Console.WriteLine (x == y);  // False
Console.WriteLine (x == z);  // True
```

Для ссылочных типов эквивалентность по умолчанию базируется на *ссылке*, а не на действительном *значении* лежащего в основе объекта. Следовательно, два экземпляра объекта с идентичными данными не будут считаться равными, если только операция `==` специально не была перегружена для достижения такого эффекта (см. разделы "Тип `object`" на стр. 100 и "Перегрузка операций" на стр. 204).

Операции эквивалентности и сравнения, `==`, `!=`, `<`, `>`, `>=` и `<=`, работают со всеми числовыми типами, но должны осмотрительно использоваться с вещественными числами (см. раздел "Ошибки округления вещественных чисел" на стр. 33). Операции сравнения также работают с членами типа `enum`, сравнивая лежащие в их основе целочисленные значения.

Условные операции

Операции `&&` и `||` реализуют условия *И* и *ИЛИ*. Они часто применяются в сочетании с операцией `!`, которая выражает условие *НЕ*. В показанном ниже примере метод `UseUmbrella()` (брать ли зонт) возвращает `true`, если дождливо (`rainy`) или солнечно (`sunny`) при условии, что не дует ветер (`windy`):

```
static bool UseUmbrella (bool rainy, bool sunny,
                         bool windy)
{
  return !windy && (rainy || sunny);
}
```

Когда возможно, операции `&&` и `||` *сокращают* вычисление. В предыдущем примере, если дует ветер (`windy`), то выражение `(rainy || sunny)` даже не оценивается. Сокращение вычислений играет важную роль в обеспечении выполнения выражений, таких как показанное ниже, без генерации исключения `NullReferenceException`:

```
if (sb != null && sb.Length > 0) ...
```

Операции `&` и `|` также реализуют проверки условий *И* и *ИЛИ*:

```
return !windy & (rainy | sunny);
```

Их отличие связано с тем, что они *не сокращают вычисления*. По этой причине `&` и `|` редко используются в качестве операций сравнения.

Тернарная условная операция (называемая просто *условной операцией*) имеет форму `q ? a : b`, где результатом является `a`, если условие `q` равно `true`, и `b` — в противном случае. Например:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

Условная операция особенно удобна в запросах LINQ (Language Integrated Query — язык интегрированных запросов).

Строки и символы

Тип `char` в C# (псевдоним типа `System.Char`) представляет символ Unicode и занимает 2 байта (UTF-16). Литерал `char` указывается в одинарных кавычках:

```
char c = 'A';  // Простой символ
```

Управляющие последовательности выражают символы, которые не могут быть представлены или интерпретированы буквально. Управляющая последовательность состоит из символа обратной косой черты, за которым следует символ со специальным смыслом.

Например:

```
char newLine = '\n';
char backSlash = '\\';
```

Символы управляющих последовательностей перечислены ниже.

Символ	Смысл	Значение
\'	Одинарная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\	Обратная косая черта	0x005C
\0	Пусто	0x0000
\a	Сигнал тревоги	0x0007
\b	Забой	0x0008
\f	Перевод страницы	0x000C
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальная табуляция	0x0009
\v	Вертикальная табуляция	0x000B

Управляющая последовательность \u (или \x) позволяет указывать любой символ Unicode в виде его шестнадцатеричного кода, состоящего из четырех цифр:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Неявное преобразование из char в числовой тип работает для числовых типов, которые могут вместить тип short без знака. Для других числовых типов требуется явное преобразование.

Строковый тип

Тип string в C# (псевдоним типа System.String) представляет неизменяемую последовательность символов Unicode. Строковый литерал указывается в двойных кавычках:

```
string a = "Heat";
```

НА ЗАМЕТКУ!

`string` — это ссылочный тип, а не тип значения. Тем не менее, его операции эквивалентности следуют семантике типов значений:

```
string a = "test", b = "test";
Console.Write (a == b);  // True
```

Управляющие последовательности, допустимые для литералов `char`, также работают внутри строк:

```
string a = "Пример табуляции:\t";
```

Платой за это является необходимость дублирования символа обратной косой черты, когда он нужен буквально:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

Чтобы избежать такой проблемы, в C# разрешены *дословные* строковые литералы. Дословный строковый литерал снабжается префиксом `@` и не поддерживает управляющие последовательности. Следующая дословная строка идентична предыдущей строке:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

Дословный строковый литерал может также занимать несколько строк. Чтобы включить в дословный строковый литерал символ двойной кавычки, его понадобится записать дважды.

Конкатенация строк

Операция + выполняет конкатенацию двух строк:

```
string s = "a" + "b";
```

Один из операндов может быть нестроковым значением; в этом случае для него будет вызван метод `ToString()`. Например:

```
string s = "a" + 5;  // a5
```

Многократное применение операции + для построения строки может оказаться неэффективным: более удачное решение предусматривает использование типа `System.Text.StringBuilder`, который позволят представлять изменяемую (редактируемую) строку и располагает методами для эффективного добавления, вставки, удаления и замены подстрок.

Интерполяция строк

Строка, предваренная символом $, называется *интерполированной строкой*. Интерполированные строки могут содержать выражения, заключенные в фигурные скобки:

```
int x = 4;
Console.Write ($"Квадрат имеет {x} стороны");
// Выводит: Квадрат имеет 4 стороны
```

Внутри фигурных скобок может быть указано любое допустимое выражение C# произвольного типа, и компилятор C# преобразует это выражение в строку, вызывая `ToString()` или эквивалентный метод типа выражения. Форматирование можно изменять путем добавления к выражению двоеточия и *форматной строки* (форматные строки рассматриваются в главе 6 книги *C# 8.0. Справочник. Полное описание языка*):

```
string s =
  $"15 в шестнадцатеричной форме выглядит как {15:X2}";
// s получает значение
//   "15 в шестнадцатеричной форме выглядит как 0F"
```

Интерполированные строки должны находиться в одной строке кода, если только вы также не укажете операцию дословной строки. Обратите внимание, что операция $ должна располагаться перед @:

```
int x = 2;
string s = $@"это охватывает {
x} строк";
```

Для включения в интерполированную строку литеральной фигурной скобки символ фигурной скобки должен быть продублирован.

Сравнения строк

Тип `string` не поддерживает операции < и > для сравнений. Взамен должен применяться метод `CompareTo()` типа `string`, который возвращает положительное число, отрицательное число или ноль в зависимости от того, находится первое значение после, перед или рядом со вторым значением:

```
Console.Write ("Bbb".CompareTo ("Aaa"));  // 1
Console.Write ("Bbb".CompareTo ("Bbb"));  // 0
Console.Write ("Bbb".CompareTo ("Ccc"));  // -1
```

Поиск внутри строк

Индексатор типа `string` возвращает символ в указанной позиции:

```
Console.Write ("слово"[3]);  // в
```

Методы `IndexOf()` и `LastIndexOf()` осуществляют поиск символа внутри строки. Методы `Contains()`, `StartsWith()` и `EndsWith()` ищут подстроку внутри строки.

Манипулирование строками

Поскольку тип `string` является неизменяемым, все методы, которые "манипулируют" строкой, возвращают новую строку, оставляя исходную незатронутой:

- метод `Substring()` извлекает часть строки;
- методы `Insert()` и `Remove()` вставляют и удаляют символы в указанной позиции;
- методы `PadLeft()` и `PadRight()` добавляют пробельные символы;
- методы `TrimStart()`, `TrimEnd()` и `Trim()` удаляют пробельные символы.

В классе `string` также определены методы `ToUpper()` и `ToLower()` для изменения регистра символов, метод `Split()` для разбиения строки на подстроки (на основе предоставленных разделителей) и статический метод `Join()` для объединения подстрок в строку.

Массивы

Массив представляет фиксированное количество элементов конкретного типа. Элементы в массиве всегда хранятся в непрерывном блоке памяти, обеспечивая высокоэффективный доступ.

Массив обозначается квадратными скобками после типа элементов. Например, ниже объявлен массив из 10 символов:

```
char[] vowels = new char[10];
```

С помощью квадратных скобок также указывается *индекс* в массиве, что позволяет получать доступ к элементам по их позициям:

```
vowels[0] = 'а'; vowels[1] = 'о'; vowels[2] = 'и';
vowels[3] = 'е'; vowels[4] = 'ё'; vowels[5] = 'э';
vowels[6] = 'ы'; vowels[7] = 'у'; vowels[8] = 'ю';
vowels[9] = 'я';
Console.WriteLine (vowels [1]);  // о
```

Этот код приведет к выводу буквы "о", поскольку массив индексируется, начиная с 0. Оператор цикла `for` можно использовать для прохода по всем элементам в массиве. Цикл `for` в следующем примере выполняется для целочисленных значений `i` от 0 до 4:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels [i]);          // аоиеёэыуюя
```

Массивы также реализуют интерфейс `IEnumerable<T>` (см. раздел "Перечисление и итераторы" на стр. 152), так что по элементам массива можно проходить посредством оператора `foreach`:

```
foreach (char c in vowels) Console.Write (c);
                                        // аоиеёэыуюя
```

Во время выполнения все обращения к индексам массивов проверяются на предмет выхода за границы. В случае применения некорректного значения индекса генерируется исключение `IndexOutOfRangeException`:

```
vowels[10] = 'а';  // Ошибка времени выполнения
```

Свойство `Length` массива возвращает количество элементов в массиве. После создания массива изменить его длину невозможно. Пространство имен `System.Collection` и вложенные в него пространства имен предоставляют такие высокоуровневые структуры данных, как массивы с динамически изменяемыми размерами и словари.

Выражение инициализации массива позволяет объявлять и заполнять массив в единственном операторе:

```
char[] vowels =
  new char[] {'а','о','и','е','ё','э','ы','у','ю','я'};
```

или проще:

```
char[] vowels =
  {'а','о','и','е','ё','э','ы','у','ю','я'};
```

Все массивы унаследованы от класса `System.Array`, в котором определены общие методы и свойства для всех массивов. Сюда входят свойства экземпляра вроде `Length` и `Rank` и статические методы для выполнения следующих действий:

- динамическое создание массива (`CreateInstance()`);
- извлечение и установка элементов независимо от типа массива (`GetValue()`/`SetValue()`);
- поиск в сортированном (`BinarySearch()`) или несортированном (`IndexOf()`, `LastIndexOf()`, `Find()`, `FindIndex()`, `FindLastIndex()`) массиве;
- сортировка массива (`Sort()`);
- копирование массива (`Copy()`).

Стандартная инициализация элементов

При создании массива всегда происходит инициализация его элементов стандартными значениями. Стандартное значение для типа представляет собой результат побитового обнуления памяти. Например, предположим, что создается массив целых чисел. Поскольку `int` — тип значения, выделится пространство под 1000 целочисленных значений в непрерывном блоке памяти. Стандартным значением для каждого элемента будет 0:

```
int[] a = new int[1000];
Console.Write (a[123]);  // 0
```

Для элементов ссылочного типа стандартным значением будет `null`.

Независимо от типа элементов массив *сам по себе* всегда является объектом ссылочного типа. Например, следующий оператор допустим:

```
int[] a = null;
```

Индексы и диапазоны (C# 8)

Для упрощения работы с элементами и или порциями массива в C# 8 были введены *индексы* и *диапазоны*.

НА ЗАМЕТКУ!

Индексы и диапазоны также работают с типами Span<T> и ReadOnlySpan<T> из CLR, которые предоставляют эффективный низкоуровневый доступ к управляемой или неуправляемой памяти.

Вы также можете создавать собственные типы, работающие с индексами и диапазонами, за счет определения индексатора типа Index или Range (см. раздел "Индексаторы" на стр. 86).

Индексы

Индексы позволяют ссылаться на элементы относительно *конца* массива посредством операции ^. Конструкция ^1 обращается к последнему элементу, ^2 — ко второму с конца и т.д.:

```
char[] vowels =
  new char[] {'а','о','и','е','ё','э','ы','у','ю','я'};
char lastElement = vowels[^1];      // 'я'
char secondToLast = vowels[^2];     // 'ю'
```

(^0 соответствует длине массива, поэтому vowels[^0] приводит к возникновению ошибки.)

Индексы в C# реализованы с помощью типа Index, а потому можно также поступать следующим образом:

```
Index first = 0;
Index last = ^1;
char firstElement = vowels [first]; // 'а'
char lastElement = vowels [last];   // 'ю'
```

Диапазоны

Диапазоны дают возможность "нарезать" массив посредством операции ..:

```
char[] firstTwo = vowels [..2];     // 'а', 'о'
char[] lastEight = vowels [2..];    // 'и','е','ё',
                                    // 'э','ы','у','ю','я'
char[] middleOne = vowels [2..3];   // 'и'
```

Второе число в диапазоне является *исключающим*, так что ..2 возвращает элементы *перед* vowels[2].

В диапазонах можно также использовать символ ^. Показанный далее код возвращает последние два элемента:

```
char[] lastTwo = vowels [^2..^0];  // 'ю', 'я'
```

(Конструкция ^0 здесь допустима, поскольку второе число в диапазоне *исключающее*.)

Диапазоны реализованы в C# с помощью типа Range, поэтому можно также делать следующее:

```
Range firstTwoRange = 0..2;
char[] firstTwo = vowels [firstTwoRange];  // 'a', 'o'
```

Многомерные массивы

Многомерные массивы имеют две разновидности: *прямоугольные* и *зубчатые*. Прямоугольный массив представляет *n*-мерный блок памяти, а зубчатый массив — это массив, содержащий массивы.

Прямоугольные массивы

Прямоугольные массивы объявляются с использованием запятых для отделения каждого измерения друг от друга. Ниже приведено объявление прямоугольного двумерного массива размерностью 3×3:

```
int[,] matrix = new int [3, 3];
```

Метод GetLength() массива возвращает длину для заданного измерения (начиная с 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix [i, j] = i * 3 + j;
```

Прямоугольный массив может быть инициализирован следующим образом (для создания массива, идентичного предыдущему примеру):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

(В операторах объявления подобного рода код, выделенный полужирным, может быть опущен.)

Зубчатые массивы

Зубчатые массивы объявляются с применением последовательно идущих пар квадратных скобок для каждого измерения. Ниже показан пример объявления зубчатого двумерного массива, в котором самое внешнее измерение составляет 3:

```
int[][] matrix = new int[3][];
```

Внутренние измерения в объявлении не указываются, т.к. в отличие от прямоугольного массива каждый внутренний массив может иметь произвольную длину. Каждый внутренний массив неявно инициализируется `null`, а не пустым массивом. Каждый внутренний массив должен создаваться вручную:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int [3];  // Создать внутренний массив
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

Зубчатый массив можно инициализировать следующим образом (для создания массива, идентичного предыдущему примеру, но с дополнительным элементом в конце):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

(В операторах объявления подобного рода код, выделенный полужирным, может быть опущен.)

Упрощенные выражения инициализации массивов

Ранее уже было показано, как упростить выражения инициализации массивов, опуская ключевое слово `new` и объявление типа:

```
char[] vowels =
  new char[] {'а','о','и','е','ё','э','ы','у','ю','я'};
char[] vowels =
  {'а','о','и','е','ё','э','ы','у','ю','я'};
```

При другом подходе после ключевого слова new имя типа не указывается и компилятор должен будет самостоятельно *вывести* тип массива. Это удобное сокращение при передаче массивов в качестве аргументов. Например, рассмотрим следующий метод:

```
void Foo (char[] data) { ... }
```

Мы можем вызывать метод Foo() с массивом, создаваемым на лету:

```
// Обычный синтаксис
Foo (new char[]{'a','о','и','е','ё','э','ы','у','ю','я'});

// Сокращенный синтаксис
Foo (new[]{'a','о','и','е','ё','э','ы','у','ю','я'});
```

Как вы увидите позже, такое сокращение жизненно важно для создания массивов *анонимных типов*.

Переменные и параметры

Переменная представляет ячейку в памяти, которая содержит изменяемое значение. Переменная может быть *локальной переменной*, *параметром* (передаваемым по *значению*, `ref`, `out` либо `in`), *полем* (*экземпляра* либо *статическим*) или *элементом массива*.

Стек и куча

Стек и куча — это места, где располагаются переменные. Стек и куча имеют существенно отличающуюся семантику времени жизни.

Стек

Стек представляет собой блок памяти для хранения локальных переменных и параметров. Стек логически увеличивается при входе в метод или функцию, а после выхода уменьшается. Взгляните на следующий метод (чтобы не отвлекать внимания, проверка входного аргумента не делается):

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

Метод `Factorial()` является рекурсивным, т.е. он вызывает сам себя. Каждый раз, когда происходит вход в метод, в стеке размещается новый экземпляр `int`, а каждый раз, когда метод завершается, экземпляр `int` освобождается.

Куча

Куча — это память, в которой располагаются *объекты* (экземпляры ссылочного типа). Всякий раз, когда создается новый объект, он размещается в куче, и на него возвращается ссылка. Во время выполнения программы куча начинает заполняться по мере создания новых объектов. В исполняющей среде предусмотрен сборщик мусора, который периодически освобождает объекты из кучи, чтобы программа не столкнулась с нехваткой памяти. Объект становится пригодным для освобождения, если отсутствуют ссылки на него.

Экземпляры типов значений (и ссылки на объекты) хранятся там, где были объявлены соответствующие переменные. Если экземпляр был объявлен как поле внутри типа класса или как элемент массива, то такой экземпляр располагается в куче.

НА ЗАМЕТКУ!

В языке C# нельзя явно удалять объекты, как это можно делать в C++. Объект без ссылок со временем уничтожается сборщиком мусора.

В куче также хранятся статические поля и константы. В отличие от объектов, распределенных в куче (которые могут быть обработаны сборщиком мусора), они существуют до тех пор, пока домен приложения не прекратит свое существование.

Определенное присваивание

В C# принудительно применяется политика определенного присваивания. На практике это означает, что за пределами контекста `unsafe` получать доступ к неинициализированной памяти невозможно. Определенное присваивание приводит к трем последствиям.

- Локальным переменным должны быть присвоены значения перед тем, как их можно будет читать.
- При вызове метода должны быть предоставлены аргументы функции (если только они не помечены как необязательные — см. раздел "Необязательные параметры" на стр. 51).
- Все остальные переменные (такие как поля и элементы массивов) автоматически инициализируются исполняющей средой.

Например, следующий код вызывает ошибку на этапе компиляции:

```
static void Main()
{
  int x;
  Console.WriteLine (x);  // Ошибка на этапе компиляции
}
```

Тем не менее, если взамен переменная `x` будет *полем* включающего класса, тогда код успешно скомпилируется и во время выполнения выведет на экран значение `0`.

Стандартные значения

Экземпляры всех типов имеют стандартные значения. Стандартные значения для предопределенных типов являются результатом побитового обнуления памяти и представляют собой `null` для ссылочных типов, `0` для числовых и перечислимых типов, `'\0'` для типа `char` и `false` для типа `bool`.

Получить стандартное значение для любого типа можно с использованием ключевого слова `default` (как вы увидите позже, на практике поступать так удобно при работе с обобщениями). Стандартное значение в специальном типе значения (т.е. `struct`) — это то же самое, что и стандартные значения для всех полей, определенных данным специальным типом.

Параметры

Метод может принимать последовательность параметров. Параметры определяют набор аргументов, которые должны быть предоставлены этому методу.

В следующем примере метод Foo() имеет единственный параметр по имени p типа int:

```
static void Foo (int p)          // p является параметром
{
  ...
}
static void Main() { Foo (8); } // 8 является аргументом
```

Управлять способом передачи параметров можно с помощью модификаторов ref, out и in:

Модификатор параметра	Способ передачи	Когда требуется определенное присваивание значения переменной
Отсутствует	По значению	При *входе*
ref	По ссылке	При *входе*
out	По ссылке	При *выходе*
in	По ссылке (только чтение)	При *входе*

Передача аргументов по значению

По умолчанию аргументы в C# *передаются по значению*, что безоговорочно является самым распространенным случаем. Это означает, что при передаче методу создается копия значения:

```
static void Foo (int p)
{
  p = p + 1;                     // Увеличить p на 1
  Console.WriteLine (p);         // Вывести p на экран
}
static void Main()
{
  int x = 8;
  Foo (x);                       // Создается копия x
  Console.WriteLine (x);         // x по-прежнему имеет
                                 //   значение 8
}
```

Присваивание p нового значения не изменяет содержимое x, потому что p и x находятся в разных ячейках памяти.

Передача по значению аргумента ссылочного типа приводит к копированию *ссылки*, но не объекта. В следующем примере метод Foo() видит тот же объект StringBuilder, который был

создан в Main(), однако имеет независимую *ссылку* на него. Другими словами, sb и fooSB являются отдельными друг от друга переменными, которые ссылаются на один и тот же объект StringBuilder:

```
static void Foo (StringBuilder fooSB)
{
  fooSB.Append ("тест");
  fooSB = null;
}
static void Main()
{
  StringBuilder sb = new StringBuilder();
  Foo (sb);
  Console.WriteLine (sb.ToString()); // тест
}
```

Поскольку fooSB — *копия* ссылки, установка ее в null не приводит к установке в null переменной sb. (Тем не менее, если параметр fooSB объявить и вызвать с модификатором ref, то sb *станет* null.)

Модификатор `ref`

Для *передачи по ссылке* в C# предусмотрен модификатор параметра ref. В приведенном ниже примере p и x ссылаются на одну и ту же ячейку памяти:

```
static void Foo (ref int p)
{
  p = p + 1;
  Console.WriteLine (p);
}
static void Main()
{
  int x = 8;
  Foo (ref x);              // Передать x по ссылке
  Console.WriteLine (x);  // x теперь имеет значение 9
}
```

Теперь присваивание p нового значения изменяет содержимое x. Обратите внимание, что модификатор ref должен быть указан как при определении, так и при вызове метода. Это делает происходящее совершенно ясным.

НА ЗАМЕТКУ!

Параметр может быть передан по ссылке или по значению независимо от того, относится он к ссылочному типу или к типу значения.

Модификатор `out`

Аргумент `out` похож на аргумент `ref` за исключением следующих аспектов:

- он не нуждается в присваивании значения перед входом в функцию;
- ему должно быть присвоено значение перед *выходом* из функции.

Модификатор `out` чаще всего применяется для получения из метода нескольких возвращаемых значений.

Переменные `out` и отбрасывание

Начиная с версии C# 7, переменные можно объявлять на лету при вызове методов с параметрами `out`:

```
int.TryParse ("123", out int x);
Console.WriteLine (x);
```

Приведенный выше код эквивалентен такому коду:

```
int x;
int.TryParse ("123", out x);
Console.WriteLine (x);
```

Когда вызываются методы с множеством параметров `out`, посредством символа подчеркивания можно "отбрасывать" любые параметры, которые не интересны для кода. Предполагая, что метод `SomeBigMethod()` был определен с пятью параметрами `out`, вот как проигнорировать все параметры кроме третьего:

```
SomeBigMethod (out _, out _, out int x, out _, out _);
Console.WriteLine (x);
```

Модификатор `in`

Начиная с версии C# 7.2, параметр можно снабжать префиксом в форме модификатора `in`, чтобы предотвращать изменение

параметра внутри метода. В результате у компилятора появляется возможность избежать накладных расходов по созданию копии аргумента до его передачи, которые могут оказаться существенными в случае крупных специальных типов значений (см. раздел "Структуры" на стр. 105).

Модификатор `params`

Модификатор `params`, когда он применяется к последнему параметру метода, позволяет методу принимать любое количество аргументов определенного типа. Тип параметра должен быть объявлен как массив. Например:

```
static int Sum (params int[] ints)
{
  int sum = 0;
  for (int i = 0; i < ints.Length; i++) sum += ints[i];
  return sum;
}
```

Вызвать метод `Sum()` можно так:

```
Console.WriteLine (Sum (1, 2, 3, 4));  // 10
```

Аргумент `params` может быть также предоставлен как обычный массив. Предыдущий вызов семантически эквивалентен следующему коду:

```
Console.WriteLine (Sum (new int[] { 1, 2, 3, 4 } ));
```

Необязательные параметры

В методах, конструкторах и индексаторах можно объявлять *необязательные параметры*. Параметр является необязательным, если в его объявлении указано *стандартное значение*:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

При вызове метода необязательные параметры могут быть опущены:

```
Foo();  // 23
```

В действительности необязательному параметру `x` *передается стандартный аргумент* со значением `23` — компилятор встраивает это значение в скомпилированный код на *вызывающей* стороне.

Показанный выше вызов `Foo()` семантически эквивалентен такому вызову:

```
Foo (23);
```

потому что компилятор просто подставляет стандартное значение необязательного параметра там, где он используется.

ВНИМАНИЕ!

Добавление необязательного параметра к открытому методу, который вызывается из другой сборки, требует перекомпиляции обеих сборок — точно как в случае, если бы параметр был обязательным.

Стандартное значение необязательного параметра должно быть указано в виде константного выражения или конструктора без параметров для типа значения. Необязательные параметры не могут быть помечены посредством `ref` или `out`.

Обязательные параметры должны находиться *перед* необязательными параметрами в объявлении метода и при его вызове (исключением являются аргументы `params`, которые всегда располагаются в конце). В следующем примере для `x` передается явное значение `1`, а для `y` — стандартное значение `0`:

```
void Foo (int x = 0, int y = 0)
{
  Console.WriteLine (x + ", " + y);
}
void Test()
{
  Foo(1);  // 1, 0
}
```

Чтобы сделать обратное (передать стандартное значение для `x` и указанное явно значение для `y`), потребуется скомбинировать необязательные параметры с *именованными аргументами*.

Именованные аргументы

Вместо распознавания аргумента по позиции его можно идентифицировать по имени. Например:

```
void Foo (int x, int y)
{
  Console.WriteLine (x + ", " + y);
}
void Test()
{
  Foo (x:1, y:2);  // 1, 2
}
```

Именованные аргументы могут встречаться в любом порядке. Следующие вызовы `Foo()` семантически идентичны:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

Именованные и позиционные аргументы можно смешивать при условии, что именованные аргументы указаны последними:

```
Foo (1, y:2);
```

Именованные аргументы особенно удобны в сочетании с необязательными параметрами. Например, взгляните на такой метод:

```
void Bar (int a=0, int b=0, int c=0, int d=0) { ... }
```

Его можно вызвать, предоставив только значение для `d`:

```
Bar (d:3);
```

Это чрезвычайно удобно при работе с API-интерфейсами COM.

Объявление неявно типизированных локальных переменных с помощью `var`

Часто случается так, что переменная объявляется и инициализируется за один шаг. Если компилятор способен вывести тип из инициализирующего выражения, то на месте объявления типа можно применять ключевое слово `var`. Например:

```
var x = "строка";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Это в точности эквивалентно следующему коду:

```
string x = "строка";
System.Text.StringBuilder y =
  new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Из-за такой прямой эквивалентности неявно типизированные переменные являются статически типизированными. Скажем, приведенный ниже код вызовет ошибку на этапе компиляции:

```
var x = 5;
x = "строка";     // Ошибка на этапе компиляции;
                  // x имеет тип int
```

В разделе "Анонимные типы" на стр. 167 мы опишем сценарий, когда использовать ключевое слово `var` обязательно.

Выражения и операции

Выражение по существу указывает значение. Простейшими разновидностями выражений являются константы (наподобие `123`) и переменные (вроде `x`). Выражения могут видоизменяться и комбинироваться с помощью операций. *Операция* принимает один или более входных *операндов* и дает на выходе новое выражение:

```
12 * 30  // * - операция, а 12 и 30 - операнды
```

Допускается строить сложные выражения, поскольку операнд сам по себе может быть выражением, как операнд `(12 * 30)` в следующем примере:

```
1 + (12 * 30)
```

Операции в C# могут быть классифицированы как *унарные*, *бинарные* и *тернарные* в зависимости от количества операндов, с которыми они работают (один, два или три). Бинарные операции всегда применяют *инфиксную* форму записи, при которой операция помещается *между* двумя операндами.

Операции, которые являются неотъемлемой частью самого языка, называются *первичными*; примером может служить операция вызова метода. Выражение, не имеющее значения, называется *пустым выражением*:

```
Console.WriteLine (1)
```

Из-за того, что пустое выражение не имеет значения, оно не может использоваться в качестве операнда при построении более сложных выражений:

```
1 + Console.WriteLine (1) // Ошибка на этапе компиляции
```

Выражения присваивания

Выражение присваивания применяет операцию = для присваивания переменной результата вычисления другого выражения. Например:

```
x = x * 5
```

Выражение присваивания — не пустое выражение. На самом деле оно заключает в себе присваиваемое значение и потому может встраиваться в другое выражение. В следующем примере выражение присваивает 2 переменной x и 10 переменной y:

```
y = 5 * (x = 2)
```

Такой стиль выражения может использоваться для инициализации нескольких значений:

```
a = b = c = d = 0
```

Составные операции присваивания являются синтаксическим сокращением, которое комбинирует присваивание с другой операцией. Например:

```
x *= 2   // эквивалентно x = x * 2
x <<= 1  // эквивалентно x = x << 1
```

(Тонкое исключение из этого правила касается *событий*, которые рассматриваются позже: операции += и -= в них трактуются специальным образом и отображаются на средства доступа add и remove события.)

Приоритеты и ассоциативность операций

Когда выражение содержит несколько операций, порядок их вычисления определяется *приоритетами* и *ассоциативностью*. Операции с более высокими приоритетами выполняются перед операциями, приоритеты которых ниже. Если операции имеют одинаковые приоритеты, то порядок их выполнения определяется ассоциативностью.

Приоритеты операций

Выражение 1 + 2 * 3 вычисляется как 1 + (2 * 3), потому что операция * имеет более высокий приоритет, чем +.

Левоассоциативные операции

Бинарные операции (кроме операции присваивания, лямбда-операции и операции объединения с `null`) являются *левоассоциативными*; другими словами, они вычисляются слева направо. Например, выражение `8/4/2` вычисляется как `(8/4)/2` по причине левой ассоциативности. Разумеется, порядок вычисления можно изменить, расставив скобки.

Правоассоциативные операции

Операция присваивания, *лямбда-операция*, операция объединения с `null` и условная операция являются *правоассоциативными*; другими словами, они вычисляются справа налево. Правая ассоциативность позволяет компилировать множественное присваивание, такое как `x=y=3`: сначала значение `3` присваивается `y`, а затем результат этого выражения (`3`) присваивается `x`.

Таблица операций

В следующей таблице перечислены операции C# в порядке их приоритетов. Операции в одной и той же категории имеют одинаковые приоритеты. Операции, которые могут быть перегружены пользователем, объясняются в разделе "Перегрузка операций" на стр. 204.

Символ операции	Название операции	Пример	Возможность перегрузки
Первичные			
`.`	Доступ к члену	`x.y`	Нет
`?.`	`null`-условная	`x?.y`	Нет
`->`	Указатель на структуру (небезопасная)	`x->y`	Нет
`()`	Вызов функции	`x()`	Нет
`[]`	Массив/индекс	`a[x]`	Через индексатор
`++`	Постфиксная форма инкремента	`x++`	Да

Продолжение табл.

Символ операции	Название операции	Пример	Возможность перегрузки
`--`	Постфиксная форма декремента	`x--`	Да
`new`	Создание экземпляра	`new Foo()`	Нет
`stackalloc`	Небезопасное выделение памяти в стеке	`stackalloc(10)`	Нет
`typeof`	Получение типа по идентификатору	`typeof(int)`	Нет
`nameof`	Получение имени идентификатора	`nameof(x)`	Нет
`checked`	Включение проверки целочисленного переполнения	`checked(x)`	Нет
`unchecked`	Отключение проверки целочисленного переполнения	`unchecked(x)`	Нет
`default`	Стандартное значение	`default(char)`	Нет
`sizeof`	Получение размера структуры	`sizeof(int)`	Нет
Унарные			
`await`	Ожидание	`await myTask`	Нет
`+`	Положительное значение	`+x`	Да
`-`	Отрицательное значение	`-x`	Да
`!`	НЕ	`!x`	Да
`~`	Побитовое дополнение	`~x`	Да
`++`	Префиксная форма инкремента	`++x`	Да
`--`	Префиксная форма декремента	`--x`	Да
`()`	Приведение	`(int)x`	Нет
`*`	Значение по адресу (небезопасная)	`*x`	Нет
`&`	Адрес значения (небезопасная)	`&x`	Нет

Продолжение табл.

Символ операции	Название операции	Пример	Возможность перегрузки
Мультипликативные			
*	Умножение	x * y	Да
/	Деление	x / y	Да
%	Остаток от деления	x % y	Да
Аддитивные			
+	Сложение	x + y	Да
-	Вычитание	x - y	Да
Сдвига			
<<	Сдвиг влево	x << 1	Да
>>	Сдвиг вправо	x >> 1	Да
Отношения			
<	Меньше	x < y	Да
>	Больше	x > y	Да
<=	Меньше или равно	x <= y	Да
>=	Больше или равно	x >= y	Да
is	Принадлежность к типу или его подклассу	x is y	Нет
as	Преобразование типа	x as y	Нет
Эквивалентности			
==	Равно	x == y	Да
!=	Не равно	x != y	Да
Логическое И			
&	И	x & y	Да
Логическое исключающее ИЛИ			
^	Исключающее ИЛИ	x ^ y	Да

Окончание табл.

Символ операции	Название операции	Пример	Возможность перегрузки
Логическое ИЛИ			
`|`	ИЛИ	`x | y`	Да
Условное И			
`&&`	И	`x && y`	Через `&`
Условное ИЛИ			
`||`	ИЛИ	`x || y`	Через `|`
Объединение с `null`			
`??`	Объединение с `null`	`x??y`	Нет
Условная (тернарная)			
`?:`	Условная	`isTrue ? thenThis : elseThis`	Нет
Присваивания и лямбда (самый низкий приоритет)			
`=`	Присваивание	`x = y`	Нет
`*=`	Умножение с присваиванием	`x *= 2`	Через `*`
`/=`	Деление с присваиванием	`x /= 2`	Через `/`
`+=`	Сложение с присваиванием	`x += 2`	Через `+`
`-=`	Вычитание с присваиванием	`x -= 2`	Через `-`
`<<=`	Сдвиг влево с присваиванием	`x <<= 2`	Через `<<`
`>>=`	Сдвиг вправо с присваиванием	`x >>= 2`	Через `>>`
`&=`	Операция И с присваиванием	`x &= 2`	Через `&`
`^=`	Операция исключающего ИЛИ с присваиванием	`x ^= 2`	Через `^`
`|=`	Операция ИЛИ с присваиванием	`x |= 2`	Через `|`
`=>`	Лямбда-операция	`x => x + 1`	Нет

Операции для работы со значениями null

В языке C# определены три операции, предназначенные для упрощения работы со значениями null: *операция объединения с null* (null coalescing), *null-условная операция* (null-conditional) и *операция присваивания с объединением с null*.

Операция объединения с null

Операция объединения с null обозначается как ??. Она говорит о следующем: если операнд слева не равен null, тогда возвратить его значение, иначе возвратить другое значение. Например:

```
string s1 = null;
string s2 = s1 ?? "пусто"; // s2 получает значение "пусто"
```

Если левостороннее выражение не равно null, то правостороннее выражение никогда не вычисляется. Операция объединения с null также работает с типами, допускающими null (см. раздел "Типы (значений), допускающие null" на стр. 158).

null-условная операция

null-условная операция (или *элвис-операция*), обозначаемая как ?., появилась в версии C# 6. Она позволяет вызывать методы и получать доступ к членам подобно стандартной операции точки, но с той разницей, что если находящийся слева операнд равен null, то результатом выражения будет null, а не генерация исключения NullReferenceException:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();      // Ошибка не возникает;
                                // s равно null
```

Последняя строка кода эквивалентна такой строке:

```
string s = (sb == null ? null : sb.ToString());
```

Столкнувшись со значением null, элвис-операция сокращает вычисление остатка выражения. В следующем примере переменная s получает значение null, несмотря на наличие стандартной операции точки между ToString() и ToUpper():

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper();  // Ошибка
                                      // не возникает
```

Многократное использование элвис-операции необходимо, только если находящийся непосредственно слева операнд может быть равен null. Приведенное ниже выражение надежно работает в ситуациях, когда x и x.y равны null:

```
x?.y?.z
```

Оно эквивалентно следующему выражению (за исключением того, что x.y оценивается только раз):

```
x == null ? null
          : (x.y == null ? null : x.y.z)
```

Окончательное выражение должно быть способным принимать значение null. Показанный далее код не является допустимым, т.к. тип int не может принимать null:

```
System.Text.StringBuilder sb = null;
int length = sb?.ToString().Length;  // Не допускается
```

Исправить положение можно за счет применения типа значения, допускающего null (см. раздел "Типы (значений), допускающие null" на стр. 158):

```
int? length = sb?.ToString().Length; // Допустимо;
                                     // int? может принимать null
```

null-условную операцию можно также использовать для вызова метода void:

```
someObject?.SomeVoidMethod();
```

Если переменная someObject равна null, то вместо генерации исключения NullReferenceException этот вызов превращается в "отсутствие операции".

null-условная операция может применяться с часто используемыми членами типов, которые описаны в разделе "Классы" на стр. 76, в том числе с *методами*, *полями*, *свойствами* и *индексаторами*. Она также хорошо сочетается с операцией объединения с null:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "пусто";    // s получает
                                         // значение "пусто"
```

Операция присваивания с объединением с null (C# 8)

Операция ??= выполняет присваивание переменной, только если ее значением не является null. Вместо кода:

```
if (s != null) s = "Добро пожаловать";
```

теперь можно записывать так:

```
s ??= "Добро пожаловать";
```

Операторы

Функции состоят из операторов, которые выполняются последовательно в порядке их появления внутри программы. *Блок операторов* — это группа операторов, находящихся между фигурными скобками ({ }).

Операторы объявления

Оператор объявления объявляет новую переменную с возможностью ее дополнительной инициализации посредством выражения. Оператор объявления завершается точкой с запятой. Можно объявлять несколько переменных одного и того же типа, указывая их в списке с запятой в качестве разделителя. Например:

```
bool rich = true, famous = false;
```

Объявление константы похоже на объявление переменной за исключением того, что после объявления константа не может быть изменена и объявление обязательно должно сопровождаться инициализацией (см. раздел "Константы" на стр. 77):

```
const double c = 2.99792458E08;
```

Область видимости локальной переменной

Областью видимости локальной переменной или локальной константы является текущий блок. Объявлять еще одну локальную переменную с тем же самым именем в текущем блоке или в любых вложенных блоках не разрешено.

Операторы выражений

Операторы выражений — это выражения, которые также представляют собой допустимые операторы. На практике такие выражения что-то "делают"; другими словами, выражения:

- создают экземпляр объекта;
- вызывают метод.

Выражения, которые не делают ничего из перечисленного выше, не являются допустимыми операторами:

```
string s = "foo";
s.Length;  // Недопустимый оператор: ничего не делает!
```

При вызове конструктора или метода, который возвращает значение, вы не обязаны использовать результат. Тем не менее, если только данный конструктор или метод не изменяет состояние, то такой оператор бесполезен:

```
new StringBuilder();       // Допустим, но бесполезен
x.Equals (y);              // Допустим, но бесполезен
```

Операторы выбора

Операторы выбора предназначены для условного управления потоком выполнения программы.

Оператор `if`

Оператор `if` выполняет некоторый оператор, если результатом вычисления выражения `bool` оказывается `true`. Например:

```
if (5 < 2 * 3)
  Console.WriteLine ("true");        // true
```

Оператором может быть блок кода:

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");        // true
  Console.WriteLine ("...")
}
```

Конструкция `else`

Оператор `if` может дополнительно содержать конструкцию `else`:

```
if (2 + 2 == 5)
  Console.WriteLine ("Не вычисляется");
else
  Console.WriteLine ("False");  // False
```

Внутрь конструкции `else` можно помещать еще один оператор `if`:

```
if (2 + 2 == 5)
  Console.WriteLine ("Не вычисляется");
else
  if (2 + 2 == 4)
   Console.WriteLine ("Вычисляется"); // Вычисляется
```

Изменение потока выполнения с помощью фигурных скобок

Конструкция `else` всегда применяется к непосредственно предшествующему оператору `if` в блоке операторов. Например:

```
if (true)
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("выполняется");
```

Это семантически идентично следующему коду:

```
if (true)
{
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("выполняется");
}
```

Переместив фигурные скобки, поток выполнения можно изменить:

```
if (true)
{
  if (false)
    Console.WriteLine();
}
else
  Console.WriteLine ("не выполняется");
```

В языке C# отсутствует аналог ключевого слова "elseif "; однако приведенный ниже шаблон позволяет достичь того же результата:

```
static void TellMeWhatICanDo (int age)
{
  if (age >= 35)
    Console.WriteLine ("Ты можешь стать президентом!");
  else if (age >= 21)
    Console.WriteLine ("Ты можешь пить!");
  else if (age >= 18)
    Console.WriteLine ("Ты можешь голосовать!");
  else
    Console.WriteLine ("Ты должен ждать!");
}
```

Оператор switch

Операторы `switch` позволяют организовать ветвление потока выполнения программы на основе выбора из возможных значений, которые переменная может принимать. Операторы `switch` могут дать в результате более ясный код, чем множество операторов `if`, потому что они требуют только однократной оценки выражения. Например:

```
static void ShowCard (int cardNumber)
{
  switch (cardNumber)
  {
    case 13:
      Console.WriteLine ("Король");
      break;
    case 12:
      Console.WriteLine ("Дама");
      break;
    case 11:
      Console.WriteLine ("Валет");
      break;
    default:  // Любое другое значение cardNumber
      Console.WriteLine (cardNumber);
      break;
  }
}
```

Значения в каждом выражении `case` должны быть константами, что ограничивает разрешенные типы встроенными целочисленными типами, типами `bool`, `char` и `enum`, а также типом `string`.

В конце каждой конструкции case необходимо явно указывать, куда выполнение должно передаваться дальше, с помощью одного из операторов перехода. Ниже перечислены варианты:

- break (переход в конец оператора switch);
- goto case *x* (переход на другую конструкцию case);
- goto default (переход на конструкцию default);
- любой другой оператор перехода, в частности, return, throw, continue или goto *метка*.

Если для нескольких значений должен выполняться тот же самый код, то конструкции case можно записывать последовательно:

```
switch (cardNumber)
{
  case 13:
  case 12:
  case 11:
    Console.WriteLine ("Фигурная карта");
    break;
  default:
    Console.WriteLine ("Нефигурная карта");
    break;
}
```

Такая особенность оператора switch может иметь решающее значение в плане получения более ясного кода, чем в случае множества операторов if-else.

Оператор switch с шаблонами

Начиная с версии C# 7, можно переключаться на основе *типа*:

```
static void TellMeTheType (object x)
{
  switch (x)
  {
    case int i:
      Console.WriteLine ("Это целочисленное значение!");
      break;
    case string s:
      Console.WriteLine (s.Length);  // Можно
                                     // использовать s
      break;
```

```
    case bool b when b == true  // Выполняется,
                                //   когда b равно true
      Console.WriteLine ("True");
      break;
    case null:  // Можно также переключаться по null
      Console.WriteLine ("null");
      break;
  }
}
```

(Тип `object` допускает переменную любого типа — см. разделы "Наследование" на стр. 91 и "Тип `object`" на стр. 100.)

В каждой конструкции `case` указывается тип, с которым следует сопоставлять, и переменная, которой необходимо присвоить типизированное значение в случае успешного совпадения. В отличие от констант ограничения на применяемые типы отсутствуют. В необязательной конструкции `when` указывается условие, которое должно быть удовлетворено, чтобы произошло совпадение для `case`.

Порядок следования конструкций `case` важен, когда производится переключение по типу (что отличается от случая переключения по константам). Исключением из этого правила является конструкция `default`, которая выполняется последней независимо от того, где она находится.

Можно указывать несколько конструкций `case` подряд. Вызов `Console.WriteLine()` в приведенном ниже коде будет выполняться для значения любого типа с плавающей точкой, которое больше 1000:

```
switch (x)
{
  case float f when f > 1000:
  case double d when d > 1000:
  case decimal m when m > 1000:
    Console.WriteLine ("Переменные f, d и m находятся
вне области видимости");
    break;
```

В данном примере компилятор позволяет задействовать переменные `f`, `d` и `m` *только* в конструкциях `when`. Во время вызова метода `Console.WriteLine()` неизвестно, какой из трех переменных будет присвоено значение, поэтому компилятор помещает их все за пределы области видимости.

Выражения switch (C# 8)

Начиная с версии C# 8, переключение можно использовать также в контексте *выражения*. Ниже представлен пример применения, в котором предполагается, что cardName имеет тип int:

```
string cardName = cardNumber switch
{
  13 => "Король",
  12 => "Дама",
  11 => "Валет",
  _  => "Нефигурная карта"  // Эквивалент default
};
```

Обратите внимание на то, что ключевое слово switch находится *после* имени переменной и конструкции case являются выражениями (заканчивающими запятыми), а не операторами. Переключение можно также организовать по множественным значениям (*кортежам*):

```
int cardNumber = 12; string suite = "spades";
string cardName = (cardNumber, suite) switch
{
  (13, "spades") => "Пиковый король",
  (13, "clubs")  => "Трефовый король",
  ...
};
```

Операторы итераций

Язык C# позволяет многократно выполнять последовательность операторов с помощью операторов while, do-while, for и foreach.

Циклы while и do-while

Циклы while многократно выполняют код в своем теле до тех пор, пока результатом вычисления выражения bool является true. Выражение проверяется *перед* выполнением тела цикла. Например, следующий код выведет 012:

```
int i = 0;
while (i < 3)
{           // Фигурные скобки не обязательны
  Console.Write (i++);
}
```

Циклы do-while отличаются по функциональности от циклов while только тем, что выражение в них проверяется *после* выполнения блока операторов (гарантируя, что блок выполняется, по крайней мере, один раз). Ниже приведен предыдущий пример, переписанный для использования цикла do-while:

```
int i = 0;
do
{
  Console.WriteLine (i++);
}
while (i < 3);
```

Циклы for

Циклы for похожи на циклы while, но имеют специальные конструкции для *инициализации* и *итерирования* переменной цикла. Цикл for содержит три конструкции:

```
for (конструкция-инициализации; конструкция-условия;
     конструкция-итерации)
  оператор-или-блок-операторов
```

Часть *конструкция-инициализации* выполняется перед началом цикла и обычно инициализирует одну или большее количество переменных *итерации*.

Часть *конструкция-условия* представляет собой выражение типа bool, которое проверяется *перед* каждой итерацией цикла. Тело цикла выполняется до тех пор, пока условие дает true.

Часть *конструкция-итерации* выполняется *после* каждой итерации цикла. Эта часть обычно применяется для обновления переменной итерации.

Например, следующий код выводит числа от 0 до 2:

```
for (int i = 0; i < 3; i++)
  Console.WriteLine (i);
```

Показанный далее код выводит первые 10 чисел Фибоначчи (где каждое число является суммой двух предшествующих):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine (prevFib);
  int newFib = prevFib + curFib;
  prevFib = curFib; curFib = newFib;
}
```

Любая из трех частей оператора for может быть опущена. Бесконечный цикл можно реализовать так (взамен допускается использовать while(true)):

```
for (;;) Console.WriteLine ("прервите меня");
```

Циклы foreach

Оператор foreach обеспечивает проход по всем элементам в перечислимом объекте. Большинство типов в C# и .NET Core, которые представляют набор или список элементов, являются перечислимыми, например, массив и строка. Ниже демонстрируется перечисление символов в строке, от первого до последнего:

```
foreach (char c in "горы")
  Console.WriteLine (c + " ");  // г о р ы
```

Перечислимые объекты определены в разделе "Перечисление и итераторы" на стр. 152.

Операторы перехода

К операторам перехода в C# относятся break, continue, goto, return и throw. Ключевое слово throw описано в разделе "Операторы try и исключения" на стр. 143.

Оператор break

Оператор break завершает выполнение тела итерации или оператора switch:

```
int x = 0;
while (true)
{
  if (x++ > 5) break;  // прекратить цикл
}
// После break выполнение продолжится здесь
...
```

Оператор continue

Оператор continue пропускает оставшиеся операторы в цикле и начинает следующую итерацию. Показанный далее цикл *пропускает* четные числа:

```
for (int i = 0; i < 10; i++)
{
```

```
  if ((i % 2) == 0) continue;
    Console.Write (i + " "); // 1 3 5 7 9
}
```

Оператор goto

Оператор `goto` переносит выполнение на метку (обозначаемую с помощью суффикса в виде двоеточия) внутри блока операторов. Следующий код выполняет итерацию по числам от 1 до 5, имитируя поведение цикла `for`:

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " "); // 1 2 3 4 5
  i++;
  goto startLoop;
}
```

Оператор return

Оператор `return` завершает метод и должен возвращать выражение, имеющее возвращаемый тип метода, если метод не является `void`:

```
static decimal AsPercentage (decimal d)
{
  decimal p = d * 100m;
  return p; //Возвратиться в вызывающий метод со значением
}
```

Оператор `return` может находиться в любом месте метода (кроме блока `finally`) и использоваться более одного раза.

Пространства имен

Пространство имен — это область, внутри которой имена типов должны быть уникальными. Типы обычно организуются в иерархические пространства имен, чтобы устранять конфликты имен и упрощать поиск имен типов. Например, тип `RSA`, который поддерживает шифрование открытым ключом, определен в пространстве имен `System.Security.Cryptography`.

Пространство имен является неотъемлемой частью имени типа. В показанном далее коде производится вызов метода `Create()` класса RSA:

```
System.Security.Cryptography.RSA rsa =
  System.Security.Cryptography.RSA.Create();
```

НА ЗАМЕТКУ!

Пространства имен не зависят от сборок, которые представляют собой единицы развертывания, такие как `.exe` или `.dll`.

Пространства имен также не влияют на доступность членов — `public`, `internal`, `private` и т.д.

Ключевое слово `namespace` определяет пространство имен для типов внутри данного блока. Например:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

Иерархия вложенных пространств имен отражается посредством точек. Следующий код семантически идентичен предыдущему примеру:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

Ссылаться на тип можно с помощью его *полностью заданного имени*, которое включает все пространства имен, от самого внешнего до самого внутреннего. Например, вот как можно было бы сослаться на `Class1` из предыдущего примера: `Outer.Middle.Inner.Class1`.

Говорят, что типы, которые не определены в каком-либо пространстве имен, находятся в *глобальном пространстве имен*. Глобальное пространство имен также включает пространства имен верхнего уровня, такие как `Outer` в приведенном выше примере.

Директива `using`

Директива `using` *импортирует* пространство имен и является удобным способом ссылки на типы без указания их полностью заданных имен. Мы можем ссылаться на `Class1` из предыдущего примера следующим образом:

```
using Outer.Middle.Inner;
class Test        // Test находится в глобальном
                  //   пространстве имен
{
  static void Main()
  {
    Class1 c;     // Полностью заданное имя указывать
                  //   не обязательно
    ...
  }
}
```

Для ограничения области видимости директива `using` может быть вложена в само пространство имен.

Директива `using static`

Начиная с версии C# 6, можно импортировать не только пространство имен, но и специфический тип с применением директивы `using static`. Затем все статические члены данного типа можно использовать без их снабжения именем типа. В приведенном далее примере вызывается статический метод `WriteLine()` класса `Console`:

```
using static System.Console;
class Test
{
  static void Main() { WriteLine ("Добро пожаловать"); }
}
```

Директива `using static` импортирует все доступные статические члены типа, включая поля, свойства и вложенные типы. Эту директиву можно также применять к перечислимым типам (см. раздел "Перечисления" на стр. 113), в случае чего импортируются их члены. Если между несколькими директивами статического импорта возникнет неоднозначность, тогда компилятор C# не сможет вывести корректный тип из контекста и сообщит об ошибке.

Правила внутри пространств имен

Область видимости имен

Имена, объявленные во внешних пространствах имен, могут использоваться во внутренних пространствах имен без дополнительного указания пространства. В следующем примере `Class1` не нуждается в указании пространства имен внутри `Inner`:

```
namespace Outer
{
  class Class1 {}
  namespace Inner
  {
    class Class2 : Class1 {}
  }
}
```

Если на тип необходимо ссылаться из другой ветви иерархии пространств имен, то можно применять частично заданное имя. В приведенном ниже примере класс `SalesReport` основан на `Common.ReportBase`:

```
namespace MyTradingCompany
{
  namespace Common
  {
    class ReportBase {}
  }
  namespace ManagementReporting
  {
    class SalesReport : Common.ReportBase {}
  }
}
```

Сокрытие имен

Если одно и то же имя типа встречается во внутреннем и во внешнем пространстве имен, то преимущество получает тип из внутреннего пространства имен. Чтобы сослаться на тип во внешнем пространстве имен, имя потребуется уточнить.

НА ЗАМЕТКУ!

Все имена типов во время компиляции преобразуются в полностью заданные имена. В коде на промежуточном языке (Intermediate Language — IL) неполные или частично заданные имена отсутствуют.

Повторяющиеся пространства имен

Объявление пространства имен можно повторять, если имена типов в этих пространствах имен не конфликтуют друг с другом:

```
namespace Outer.Middle.Inner { class Class1 {} }
namespace Outer.Middle.Inner { class Class2 {} }
```

Классы могут даже охватывать файлы исходного кода и сборки.

Квалификатор `global::`

Иногда полностью заданное имя типа может конфликтовать с каким-то внутренним именем. Чтобы заставить компилятор C# использовать полностью заданное имя типа, его понадобится снабдить префиксом `global::`, как показано ниже:

```
global::System.Text.StringBuilder sb;
```

Назначение псевдонимов типам и пространствам имен

Импортирование пространства имен может привести к конфликту имен типов. Вместо полного пространства имен можно импортировать только специфические типы, которые нужны, и назначать каждому такому типу псевдоним. Например:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

Псевдоним можно назначить целому пространству имен:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

Классы

Класс является наиболее распространенной разновидностью ссылочного типа. Вот как выглядит объявление простейшего класса из всех возможных:

```
class Foo
{
}
```

Более сложный класс может дополнительно иметь перечисленные ниже компоненты.

Перед ключевым словом `class`	*Атрибуты* и *модификаторы класса.* Модификаторами невложенных классов являются `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe` и `partial`
После имени класса	*Параметры обобщенных типов*, *базовый класс* и *интерфейсы*
Внутри фигурных скобок	*Члены класса* (к ним относятся методы, свойства, индексаторы, события, поля, конструкторы, перегруженные операции, вложенные типы и финализатор)

Поля

Поле — это переменная, которая является членом класса или структуры. Например:

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

Поле может иметь модификатор `readonly`, который предотвращает его изменение после конструирования. Присваивать значение полю, допускающему только чтение, можно лишь в его объявлении или внутри конструктора типа, в котором оно определено.

Инициализация полей является необязательной. Неинициализированное поле получает свое стандартное значение (`0`, `\0`, `null`, `false`). Инициализаторы полей выполняются перед конструкторами в порядке, в котором они указаны.

Для удобства множество полей одного типа можно объявлять в списке, разделяя запятыми. Это подходящий способ обеспечить совместное использование всеми полями одних и тех же атрибутов и модификаторов полей. Например:

```
static readonly int legs = 8, eyes = 2;
```

Константы

Константа вычисляется статически на этапе компиляции и компилятор буквально подставляет ее значение везде, где оно используется (что довольно похоже на макрос в C++). Константа может иметь любой встроенный тип: `bool`, `char`, `string` или перечисление.

Константа объявляется посредством ключевого слова `const` и должна быть инициализирована каким-то значением. Например:

```
public class Test
{
  public const string Message = "Добро пожаловать";
}
```

Константа является гораздо более ограничивающей, чем поле `static readonly` — как в плане типов, которые можно применять, так и в смысле семантики инициализации полей. Кроме того, константа отличается от поля `static readonly` тем, что ее вычисление происходит на этапе компиляции. Константы также могут объявляться локально в методе:

```
static void Main()
{
  const double twoPI = 2 * System.Math.PI;
  ...
}
```

Методы

Метод выполняет действие в форме последовательности операторов. Метод может получать *входные* данные из вызывающего кода посредством указания *параметров* и возвращать *выходные*

данные обратно вызывающему коду за счет указания *возвращаемого типа*. Для метода может быть определен возвращаемый тип `void`, который указывает на то, что метод никакого значения не возвращает. Метод также может возвращать выходные данные вызывающему коду через параметры `ref` и `out`.

Сигнатура метода должна быть уникальной в рамках типа. Сигнатура метода включает в себя имя метода и типы параметров (но не содержит *имена* параметров и возвращаемый тип).

Методы, сжатые до выражений

Метод следующего вида, который состоит из единственного выражения:

```
int Foo (int x) { return x * 2; }
```

начиная с версии C# 6, можно более кратко записать как *метод, сжатый до выражения* (expression-bodied method). Фигурные скобки и ключевое слово `return` заменяются комбинацией =>:

```
int Foo (int x) => x * 2;
```

Функции, сжатые до выражений, могут также иметь возвращаемый тип `void`:

```
void Foo (int x) => Console.WriteLine (x);
```

Перегрузка методов

Тип может перегружать методы (иметь несколько методов с одним и тем же именем) при условии, что типы параметров отличаются. Например, все перечисленные ниже методы могут сосуществовать внутри одного типа:

```
void Foo (int x);
void Foo (double x);
void Foo (int x, float y);
void Foo (float x, int y);
```

Локальные методы

Начиная с версии C# 7, метод разрешено определять внутри другого метода:

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  int Cube (int value) => value * value * value;
}
```

Локальный метод (`Cube()` в данном случае) будет видимым только для объемлющего метода (`WriteCubes()`). Это упрощает содержащий тип и немедленно подает сигнал любому просматривающему код, что `Cube()` больше нигде не применяется. Локальные методы могут обращаться к локальным переменным и параметрам объемлющего метода, что имеет несколько последствий, которые описаны в разделе "Захватывание внешних переменных" на стр. 139.

Локальные методы могут появляться внутри функций других видов, таких как средства доступа к свойствам, конструкторы и т.д., и даже внутри других локальных методов. Локальные методы могут быть итераторными или асинхронными.

Статические локальные методы (C# 8)

Добавление модификатора `static` к локальному методу не позволяет ему видеть локальные переменные и параметры объемлющего метода, что помогает ослабить связность, а также разрешить локальному методу объявлять переменные по своему усмотрению без риска возникновения конфликта с переменными в объемлющем методе.

Конструкторы экземпляров

Конструкторы выполняют код инициализации класса или структуры. Конструктор определяется подобно методу за исключением того, что вместо имени метода и возвращаемого типа указывается имя типа, к которому относится этот конструктор:

```
public class Panda
{
  string name;              // Определение поля
  public Panda (string n) // Определение конструктора
  {
    name = n;               // Код инициализации
  }
}
...
Panda p = new Panda ("Petey");  // Вызов конструктора
```

Начиная с версии C# 7, конструкторы с единственным оператором могут записываться как члены, сжатые до выражений:

```
public Panda (string n) => name = n;
```

Класс или структура может перегружать конструкторы. Один перегруженный конструктор способен вызывать другой, используя ключевое слово `this`:

```
public class Wine
{
  public Wine (decimal price) {...}
  public Wine (decimal price, int year)
              : this (price) {...}
}
```

Когда один конструктор вызывает другой, то первым выполняется *вызванный конструктор*.

Другому конструктору можно передавать *выражение*:

```
public Wine (decimal price, DateTime year)
            : this (price, year.Year) {...}
```

В самом выражении не допускается применять ссылку `this`, скажем, для вызова метода экземпляра. Тем не менее, вызывать статические методы разрешено.

Неявные конструкторы без параметров

Компилятор C# автоматически генерирует для класса открытый конструктор без параметров тогда и только тогда, когда в нем не было определено ни одного конструктора. Однако после определения хотя бы одного конструктора конструктор без параметров больше автоматически не генерируется.

Неоткрытые конструкторы

Конструкторы не обязательно должны быть открытыми. Распространенной причиной наличия неоткрытого конструктора является управление созданием экземпляров через вызов статического метода. Статический метод может использоваться для возвращения объекта из пула вместо создания нового объекта или для возвращения специализированного подкласса, выбираемого на основе входных аргументов.

Деконструкторы

В то время как конструктор обычно принимает набор значений (в виде параметров) и присваивает их полям, деконструктор (C# 7+) делает противоположное и присваивает поля набо-

ру переменных. Метод деконструирования должен называться `Deconstruct()` и иметь один или большее количество параметров `out`:

```
class Rectangle
{
  public readonly float Width, Height;
  public Rectangle (float width, float height)
  {
    Width = width; Height = height;
  }
  public void Deconstruct (out float width,
                           out float height)
  {
    width = Width; height = Height;
  }
}
```

Для вызова деконструктора применяется следующий специальный синтаксис:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;
Console.WriteLine (width + " " + height);  // 3 4
```

Деконструирующий вызов содержится во второй строке. Он создает две локальных переменных и затем обращается к методу `Deconstruct()`. Вот чему эквивалентен такой деконструирующий вызов:

```
rect.Deconstruct (out var width, out var height);
```

Деконструирующие вызовы допускают неявную типизацию, так что наш вызов можно было бы сократить следующим образом:

```
(var width, var height) = rect;
```

Или просто:

```
var (width, height) = rect;
```

Если переменные, в которые производится деконструирование, уже определены, то типы вообще не указываются; это называется *деконструирующим присваиванием*:

```
(width, height) = rect;
```

Перегружая метод `Deconstruct()`, вызывающему коду можно предложить целый диапазон вариантов деконструирования.

НА ЗАМЕТКУ!

Метод `Deconstruct()` может быть расширяющим методом (см. раздел "Расширяющие методы" на стр. 165). Прием удобен, если вы хотите деконструировать типы, автором которых не являетесь.

Инициализаторы объектов

Для упрощения инициализации объекта любые его доступные поля и свойства могут быть установлены с помощью *инициализатора объекта* непосредственно после создания. Например, рассмотрим следующий класс:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots, LikesHumans;

  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```

Используя инициализаторы объектов, создавать объекты `Bunny` можно так:

```
Bunny b1 = new Bunny {
                       Name="Bo",
                       LikesCarrots = true,
                       LikesHumans = false
                     };
Bunny b2 = new Bunny ("Bo") {
                              LikesCarrots = true,
                              LikesHumans = false
                            };
```

Ссылка `this`

Ссылка `this` указывает на сам экземпляр. В следующем примере метод `Marry()` использует ссылку `this` для установки поля `Mate` экземпляра `partner`:

```
public class Panda
{
```

```
  public Panda Mate;
  public void Marry (Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

Ссылка `this` также устраняет неоднозначность между локальной переменной или параметром и полем. Например:

```
public class Test
{
  string name;
  public Test (string name) { this.name = name; }
}
```

Ссылка `this` допустима только внутри нестатических членов класса или структуры.

Свойства

Снаружи свойства выглядят похожими на поля, но подобно методам внутренне они содержат логику. Например, взглянув на следующий код, невозможно выяснить, чем является `CurrentPrice` — полем или свойством:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Свойство объявляется подобно полю, но с добавлением блока `get/set`. Ниже показано, как реализовать `CurrentPrice` в виде свойства:

```
public class Stock
{
  decimal currentPrice;    // Закрытое "поддерживающее"
                           //   поле
  public decimal CurrentPrice  // Открытое свойство
  {
    get { return currentPrice; }
    set { currentPrice = value; }
  }
}
```

С помощью get и set обозначаются *средства доступа* к свойству. Средство доступа get запускается при чтении свойства. Оно должно возвращать значение, имеющее тип самого свойства. Средство доступа set выполняется во время присваивания свойству значения. Оно принимает неявный параметр по имени value с типом свойства, который обычно присваивается закрытому полю (в данном случае currentPrice).

Хотя доступ к свойствам осуществляется таким же способом, как к полям, свойства отличаются тем, что предоставляют программисту полный контроль над получением и установкой их значений. Такой контроль позволяет программисту выбирать любое необходимое внутреннее представление, не демонстрируя детали пользователю свойства. В приведенном примере метод set мог бы генерировать исключение, если значение value выходит за пределы допустимого диапазона.

НА ЗАМЕТКУ!

В книге повсеместно применяются открытые поля, чтобы излишне не усложнять примеры и не отвлекать от сути. В реальном приложении для содействия инкапсуляции предпочтение обычно отдается открытым свойствам, а не открытым полям.

Свойство будет предназначено только для чтения, если для него указано одно лишь средство доступа get, и только для записи, если определено одно лишь средство доступа set. Свойства только для записи используются редко.

Свойство обычно имеет отдельное поддерживающее поле, предназначенное для хранения лежащих в основе данных. Тем не менее, это не обязательно — свойство может возвращать значение, вычисленное на базе других данных. Например:

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
  get { return currentPrice * sharesOwned; }
}
```

Свойства, сжатые до выражений

Начиная с версии C# 6, свойство только для чтения вроде показанного в предыдущем разделе можно объявлять более кратко как *свойство, сжатое до выражения* (expression-bodied property). Фигурные скобки, а также ключевые слова `get` и `return` заменяются комбинацией =>:

```
public decimal Worth => currentPrice * sharesOwned;
```

Начиная с версии C# 7, дополнительно допускается объявлять сжатыми до выражения также и средства доступа `set`:

```
public decimal Worth
{
  get => currentPrice * sharesOwned;
  set => sharesOwned = value / currentPrice;
}
```

Автоматические свойства

Наиболее распространенная реализация свойства предусматривает наличие средств доступа `get` и/или `set`, которые просто читают и записывают в закрытое поле того же типа, что и свойство. Объявление *автоматического свойства* указывает компилятору на необходимость предоставления такой реализации. Первый пример в этом разделе можно усовершенствовать, объявив `CurrentPrice` как автоматическое свойство:

```
public class Stock
{
  public decimal CurrentPrice { get; set; }
}
```

Компилятор автоматически создает закрытое поддерживающее поле со специальным сгенерированным именем, ссылаться на которое невозможно. Средство доступа `set` может быть помечено как `private` или `protected`, если свойство должно быть доступно другим типам только для чтения.

Инициализаторы свойств

Начиная с версии C# 6, к автоматическим свойствам можно добавлять инициализаторы свойств в точности как к полям:

```
public decimal CurrentPrice { get; set; } = 123;
```

В результате свойство CurrentPrice получает начальное значение 123. Свойства с инициализаторами могут допускать только чтение:

```
public int Maximum { get; } = 999;
```

Как и поля, предназначенные только для чтения, автоматические свойства, допускающие только чтение, могут устанавливаться также в конструкторе типа. Это удобно при создании *неизменяемых* (только для чтения) типов.

Доступность get и set

Средства доступа get и set могут иметь разные уровни доступа. В типичном сценарии применения есть свойство public с модификатором доступа internal или private, указанным для средства доступа set:

```
private decimal x;
public decimal X
{
  get { return x; }
  private set { x = Math.Round (value, 2); }
}
```

Обратите внимание, что само свойство объявлено с более либеральным уровнем доступа (public в данном случае), а к средству доступа, которое должно быть *менее* доступным, добавлен модификатор.

Индексаторы

Индексаторы предлагают естественный синтаксис для доступа к элементам в классе или структуре, которая инкапсулирует список либо словарь значений. Индексаторы подобны свойствам, но предусматривают доступ через аргумент индекса, а не имя свойства. Класс string имеет индексатор, который позволяет получать доступ к каждому его значению char посредством индекса int:

```
string s = "строка";
Console.WriteLine (s[0]);  // 'с'
Console.WriteLine (s[3]);  // 'о'
```

Синтаксис использования индексов похож на синтаксис работы с массивами за исключением того, что аргумент (аргументы) индекса может быть любого типа (типов). Индексаторы могут вызываться null-условным образом за счет помещения вопросительного знака перед открывающей квадратной скобкой (см. раздел "Операции для работы со значениями null" на стр. 60):

```
string s = null;
Console.WriteLine (s?[0]);      // Ничего не выводится;
                                //   ошибка не возникает
```

Реализация индексатора

Для реализации индексатора понадобится определить свойство по имени this, указав аргументы в квадратных скобках. Например:

```
class Sentence
{
  string[] words = "Большой хитрый рыжий лис".Split();
  public string this [int wordNum]  // индексатор
  {
    get { return words [wordNum]; }
    set { words [wordNum] = value; }
  }
}
```

Ниже показано, как можно было бы применять индексатор:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);     // лис
s[3] = "пес";
Console.WriteLine (s[3]);     // пес
```

Для типа можно объявлять несколько индексаторов, каждый с параметрами разных типов. Индексатор также может принимать более одного параметра:

```
public string this [int arg1, string arg2]
{
  get { ... } set { ... }
}
```

Если опустить средство доступа set, то индексатор станет предназначенным только для чтения, и его определение можно сократить с использованием синтаксиса, сжатого до выражения (начиная с версии C# 6):

```
public string this [int wordNum] => words [wordNum];
```

Использование индексов и диапазонов с помощью индексаторов (C# 8)

Поддерживать в своих классах индексы и диапазоны (см. раздел "Индексы и диапазоны (C# 8)" на стр. 41) можно за счет определения индексатора с типом параметра `Index` или `Range`. Мы можем расширить предыдущий пример, добавив в класс `Sentence` следующие индексаторы:

```
public string this [Index index] => words [index];
public string[] this [Range range] => words [range];
```

Затем можно будет поступать так:

```
Sentence s = new Sentence();
Console.WriteLine (s [^1]);        // лис
string[] firstTwoWords = s [..2];  // (Большой, хитрый)
```

Статические конструкторы

Статический конструктор выполняется однократно для *типа*, а не однократно для *экземпляра*. В типе может быть определен только один статический конструктор, он не должен принимать параметры, и обязан иметь то же имя, что и тип:

```
class Test
{
  static Test() { Console.Write ("Тип инициализирован"); }
}
```

Исполняющая среда автоматически вызывает статический конструктор прямо перед тем, как тип начинает применяться. Этот вызов инициируется двумя действиями: создание экземпляра типа и доступ к статическому члену типа.

ВНИМАНИЕ!

Если статический конструктор генерирует необработанное исключение, тогда тип, к которому он относится, становится *непригодным* в жизненном цикле приложения.

Инициализаторы статических полей запускаются непосредственно *перед* вызовом статического конструктора. Если тип не

имеет статического конструктора, то инициализаторы статических полей будут выполняться перед тем, как тип начнет использоваться — или *в любой момент раньше* по прихоти исполняющей среды.

Статические классы

Класс может быть помечен как `static`, указывая на то, что он должен состоять исключительно из статических членов и не допускать создание подклассов на своей основе. Хорошими примерами статических классов могут служить `System.Console` и `System.Math`.

Финализаторы

Финализаторы — это методы, предназначенные только для классов, которые выполняются до того, как сборщик мусора освободит память, занятую объектом с отсутствующими ссылками на него. Синтаксически финализатор записывается как имя класса, предваренное символом ~:

```
class Class1
{
  ~Class1() { ... }
}
```

Компилятор C# транслирует финализатор в метод, который переопределяет метод `Finalize()` класса `object`. Сборка мусора и финализаторы подробно обсуждаются в главе 12 книги *C# 8.0. Справочник. Полное описание языка*.

Начиная с версии C# 7, финализаторы, состоящие из единственного оператора, могут быть записаны с помощью синтаксиса сжатия до выражения.

Частичные типы и методы

Частичные типы позволяют расщеплять определение типа, обычно разнося его по нескольким файлам. Распространенный сценарий предполагает автоматическую генерацию частичного класса из какого-то другого источника (например, шаблона Visual Studio) и последующее его дополнение вручную написанными методами. Например:

```
// PaymentFormGen.cs - сгенерирован автоматически
partial class PaymentForm { ... }

// PaymentForm.cs - написан вручную
partial class PaymentForm { ... }
```

Каждый участник должен иметь объявление `partial`.

Участники не могут содержать конфликтующие члены. Скажем, конструктор с теми же самыми параметрами повторять нельзя. Частичные типы распознаются полностью компилятором, а это значит, что каждый участник должен быть доступным на этапе компиляции и располагаться в той же самой сборке.

Базовый класс может быть указан для единственного участника или для множества участников (при условии, что для каждого из них базовый класс будет тем же). Кроме того, для каждого участника можно независимо указывать интерфейсы, подлежащие реализации. Базовые классы и интерфейсы рассматриваются в разделах "Наследование" на стр. 91 и "Интерфейсы" на стр. 109.

Частичные методы

Частичный тип может содержать *частичные методы*. Они позволяют автоматически сгенерированному частичному типу предоставлять настраиваемые точки привязки для ручного написания кода. Например:

```
partial class PaymentForm // В файле автоматически
                          //   сгенерированного кода
{
  partial void ValidatePayment (decimal amount);
}
partial class PaymentForm // В файле написанного
                          //   вручную кода
{
  partial void ValidatePayment (decimal amount)
  {
    if (amount > 100) Console.Write ("Дорого!");
  }
}
```

Частичный метод состоит из двух частей: *определения* и *реализации*. Определение обычно записывается генератором кода, а реализация — вручную. Если реализация не предоставлена, то оп-

ределение частичного метода при компиляции удаляется (вместе с кодом, в котором он вызывается). Это дает автоматически сгенерированному коду большую свободу в предоставлении точек привязки, не заставляя беспокоиться по поводу эффекта разбухания кода. Частичные методы должны быть void, и они неявно являются private.

Операция nameof

Операция nameof (появившаяся в версии C# 6) возвращает имя любого символа (типа, члена, переменной и т.д.) в виде строки:

```
int count = 123;
string name = nameof (count);  // name получает
                               //   значение "count"
```

Преимущество применения данной операции по сравнению с простым указанием строки связано со статической проверкой типов. Инструменты, подобные Visual Studio, способны воспринимать символические ссылки, поэтому переименование любого символа приводит к переименованию также его ссылок.

Для указания имени члена типа, такого как поле или свойство, необходимо включать тип члена, что работает со статическими членами и членами экземпляра:

```
string name = nameof (StringBuilder.Length);
```

Результатом будет "Length". Чтобы возвратить "StringBuilder.Length", понадобится следующее выражение:

```
nameof(StringBuilder)+"."+nameof(StringBuilder.Length);
```

Наследование

Класс может быть *унаследован* от другого класса с целью расширения или настройки исходного класса. Наследование от класса позволяет повторно использовать функциональность данного класса вместо ее построения с нуля. Класс может наследоваться только от одного класса, но сам может быть унаследован множеством классов, формируя иерархию классов. В этом примере мы начнем с определения класса по имени Asset:

```
public class Asset { public string Name; }
```

Далее мы определим классы Stock и House, которые будут унаследованы от Asset. Классы Stock и House получат все, что имеет Asset, плюс любые дополнительные члены, которые в них будут определены:

```
public class Stock : Asset      // унаследован от Asset
{
  public long SharesOwned;
}
public class House : Asset      // унаследован от Asset
{
  public decimal Mortgage;
}
```

Вот как можно работать с данными классами:

```
Stock msft = new Stock { Name="MSFT",
                         SharesOwned=1000 };
Console.WriteLine (msft.Name);          // MSFT
Console.WriteLine (msft.SharesOwned);   // 1000

House mansion = new House { Name="Mansion",
                            Mortgage=250000 };
Console.WriteLine (mansion.Name);       // Mansion
Console.WriteLine (mansion.Mortgage);   // 250000
```

Подклассы Stock и House наследуют свойство Name от *базового класса* Asset.

Подклассы также называются *производными классами*.

Полиморфизм

Ссылки являются полиморфными. Это значит, что переменная типа *x* может ссылаться на объект, относящийся к подклассу *x*. Например, рассмотрим следующий метод:

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

Метод Display() способен отображать значение свойства Name объектов Stock и House, т.к. они оба являются Asset. В основе работы полиморфизма лежит тот факт, что подклассы (Stock и House) обладают всеми характеристиками своего базового класса (Asset). Однако обратное утверждение не будет верным. Если метод Display() переписать так, чтобы он принимал House, то передача ему Asset станет невозможной.

Приведение и ссылочные преобразования

Ссылка на объект может быть:

- неявно *приведена вверх* к ссылке на базовый класс;
- явно *приведена вниз* к ссылке на подкласс.

Приведение вверх и вниз между совместимыми ссылочными типами выполняет *ссылочные преобразования*: создается новая ссылка, которая указывает на *тот же самый* объект. Приведение вверх всегда успешно; приведение вниз успешно только в случае, когда объект подходящим образом типизирован.

Приведение вверх

Операция приведения вверх создает ссылку на базовый класс из ссылки на подкласс. Например:

```
Stock msft = new Stock();  // Из предыдущего примера
Asset a = msft;            // Приведение вверх
```

После приведения вверх переменная `a` по-прежнему ссылается на тот же самый объект `Stock`, что и переменная `msft`. Сам объект, на который имеются ссылки, не изменяется и не преобразуется:

```
Console.WriteLine (a == msft);  // True
```

Хотя переменные `a` и `msft` ссылаются на один и тот же объект, `a` обеспечивает более ограниченное представление этого объекта:

```
Console.WriteLine (a.Name);         // Нормально
Console.WriteLine (a.SharesOwned); // Ошибка на этапе
                                    //   компиляции
```

Последняя строка кода вызывает ошибку на этапе компиляции, поскольку переменная `a` имеет тип `Asset` несмотря на то, что она ссылается на объект типа `Stock`. Чтобы получить доступ к полю `SharesOwned`, экземпляр `Asset` потребуется *привести вниз* к `Stock`.

Приведение вниз

Операция приведения вниз создает ссылку на подкласс из ссылки на базовый класс. Например:

```
Stock msft = new Stock();
Asset a = msft;                  // Приведение вверх
Stock s = (Stock)a;              // Приведение вниз
Console.WriteLine (s.SharesOwned); // Ошибка
                                   //   не возникает
Console.WriteLine (s == a);      // True
Console.WriteLine (s == msft);   // True
```

Как и в случае приведения вверх, затрагиваются только ссылки, но не лежащий в основе объект. Приведение вниз требует явного указания, потому что потенциально оно может не достигнуть успеха во время выполнения:

```
House h = new House();
Asset a = h;          // Приведение вверх всегда успешно
Stock s = (Stock)a;   // Приведение вниз не достигает
                      //   успеха: a не является Stock
```

Когда приведение вниз терпит неудачу, генерируется исключение `InvalidCastException`. Это пример *проверки типов во время выполнения* (которая более подробно рассматривается в разделе "Статическая проверка типов и проверка типов во время выполнения" на стр. 102).

Операция `as`

Операция `as` выполняет приведение вниз, которое в случае неудачи вычисляется как `null` (вместо генерации исключения):

```
Asset a = new Asset();
Stock s = a as Stock; // s равно null; исключение
                      // не генерируется
```

Эта операция удобна, когда нужно организовать последующую проверку результата на предмет `null`:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```

Операция `as` не может выполнять *специальные преобразования* (см. раздел "Перегрузка операций" на стр. 204), равно как и числовые преобразования.

Операция `is`

Операция `is` проверяет, будет ли преобразование ссылки успешным; другими словами, является ли объект производным от

указанного класса (или реализует ли он какой-то интерфейс). Она часто применяется при проверке перед приведением вниз:

```
if (a is Stock) Console.Write (((Stock)a).SharesOwned);
```

Операция `is` также дает в результате `true`, если может успешно выполниться *распаковывающее преобразование* (см. раздел "Тип `object`" на стр. 100). Однако она не принимает во внимание специальные или числовые преобразования.

Начиная с версии C# 7, появилась возможность вводить в действие переменную при использовании операции `is`:

```
if (a is Stock s)
  Console.WriteLine (s.SharesOwned);
```

Введенная переменная доступна для "немедленного" потребления и остается в области видимости за пределами выражения:

```
if (a is Stock s && s.SharesOwned > 100000)
  Console.WriteLine ("Wealthy");
else
  s = new Stock(); // s в области видимости
Console.WriteLine (s.SharesOwned); // По-прежнему
                                   // в области видимости
```

Виртуальные функции-члены

Функция, помеченная как *виртуальная* (`virtual`), может быть *переопределена* в подклассах, где требуется предоставить ее специализированную реализацию. Объявлять виртуальными можно методы, индексаторы и события:

```
public class Asset
{
  public string Name;
  public virtual decimal Liability => 0;
}
```

(Конструкция `Liability => 0` является сокращенной записью для `{ get { return 0; } }`. За дополнительной информацией по такому синтаксису обращайтесь в раздел "Свойства, сжатые до выражений" на стр. 85.) Виртуальный метод переопределяется в подклассе с применением модификатора `override`:

```
public class House : Asset
{
  public decimal Mortgage;
```

```
  public override decimal Liability => Mortgage;
}
```

По умолчанию свойство `Liability` класса `Asset` возвращает 0. Класс `Stock` не нуждается в специализации этого поведения. Тем не менее, класс `House` специализирует свойство `Liability`, чтобы возвращать значение `Mortgage`:

```
House mansion = new House { Name="Mansion",
                            Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability);   // 250000
Console.WriteLine (a.Liability);         // 250000
```

Сигнатуры, возвращаемые типы и доступность виртуального и переопределенного методов должны быть идентичными. Внутри переопределенного метода можно вызвать его реализацию из базового класса с помощью ключевого слова `base` (см. раздел "Ключевое слово `base`" на стр. 98).

Абстрактные классы и абстрактные члены

Класс, объявленный как *абстрактный* (`abstract`), не разрешает создавать свои экземпляры. Взамен можно создавать только экземпляры его конкретных *подклассов*.

В абстрактных классах есть возможность определять *абстрактные члены*. Абстрактные члены похожи на виртуальные члены за исключением того, что они не предоставляют стандартную реализацию. Реализация должна обеспечиваться подклассом, если только подкласс тоже не объявлен как `abstract`:

```
public abstract class Asset
{
  // Обратите внимание на пустую реализацию.
  public abstract decimal NetValue { get; }
}
```

В подклассах абстрактные члены переопределяются так, как если бы они были виртуальными.

Сокрытие унаследованных членов

В базовом классе и подклассах могут быть определены идентичные члены.

Например:

```
public class A     { public int Counter = 1; }
public class B : A { public int Counter = 2; }
```

Говорят, что поле Counter в классе B *скрывает* поле Counter в классе A. Обычно это происходит случайно, когда член добавляется к базовому типу *после* того, как идентичный член был добавлен к подтипу. В таком случае компилятор генерирует предупреждение и затем разрешает неоднозначность следующим образом:

- ссылки на A (на этапе компиляции) привязываются к A.Counter;
- ссылки на B (на этапе компиляции) привязываются к B.Counter.

Иногда необходимо преднамеренно сокрыть какой-то член; тогда к члену в подклассе можно применить ключевое слово new. Модификатор new *не делает ничего сверх того, что просто подавляет выдачу компилятором соответствующего предупреждения*:

```
public class A     { public     int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

Модификатор new сообщает компилятору — и другим программистам — о том, что дублирование члена произошло не случайно.

Запечатывание функций и классов

С помощью ключевого слова sealed переопределенная функция может *запечатывать* свою реализацию, предотвращая ее переопределение другими подклассами. В ранее показанном примере виртуальной функции-члена можно было бы запечатать реализацию Liability в классе House, чтобы запретить переопределение Liability в классе, производном от House:

```
public sealed override decimal Liability { get { ... } }
```

Можно также запечатать весь класс, неявно запечатав все его виртуальные функции, за счет применения модификатора sealed к самому классу.

Ключевое слово base

Ключевое слово base похоже на ключевое слово this. Оно служит двум важным целям: доступ к переопределенной функции-члену из подкласса и вызов конструктора базового класса (см. следующий раздел).

В приведенном ниже примере класса House ключевое слово base используется для доступа к реализации Liability из Asset:

```
public class House : Asset
{
  ...
  public override decimal Liability
    => base.Liability + Mortgage;
}
```

С помощью ключевого слова base мы получаем доступ к свойству Liability класса Asset *невиртуальным* образом. Это значит, что мы всегда обращаемся к версии Asset данного свойства независимо от фактического типа экземпляра во время выполнения.

Тот же самый подход работает в ситуации, когда Liability *скрывается*, а не *переопределяется*. (Получить доступ к скрытым членам можно также путем приведения к базовому классу перед обращением к члену.)

Конструкторы и наследование

В подклассе должны быть объявлены собственные конструкторы. Например, если классы Baseclass и Subclass определены следующим образом:

```
public class Baseclass
{
  public int X;
  public Baseclass () { }
  public Baseclass (int x) { this.X = x; }
}
public class Subclass : Baseclass { }
```

то показанный ниже код будет недопустимым:

```
Subclass s = new Subclass (123);
```

В классе Subclass должны быть "повторно определены" любые конструкторы, которые необходимо открыть. При этом можно вызывать любой конструктор базового класса с применением ключевого слова base:

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { ... }
}
```

Ключевое слово base работает подобно ключевому слову this, но вызывает конструктор базового класса. Конструкторы базовых классов всегда выполняются первыми, что гарантирует выполнение *базовой* инициализации перед *специализированной* инициализацией.

Если в конструкторе подкласса опустить ключевое слово base, тогда будет неявно вызываться конструктор *без параметров* базового класса (если базовый класс не имеет доступного конструктора без параметров, то компилятор сообщит об ошибке).

Конструктор и порядок инициализации полей

Когда объект создан, инициализация происходит в указанном ниже порядке.

1. От подкласса к базовому классу:
   а) инициализируются поля;
   б) вычисляются аргументы для вызова конструкторов базового класса.
2. От базового класса к подклассу:
   а) выполняются тела конструкторов.

Перегрузка и распознавание

Наследование оказывает интересное влияние на перегрузку методов. Предположим, что есть следующие две перегруженных версии:

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

При вызове перегруженной версии приоритет получает наиболее специфичный тип:

```
House h = new House (...);
Foo(h);  // Вызывается Foo(House)
```

Конкретная перегруженная версия, подлежащая вызову, определяется статически (на этапе компиляции), а не во время выполнения. В показанном ниже коде вызывается `Foo(Asset)` несмотря на то, что типом времени выполнения переменной `a` является `House`:

```
Asset a = new House (...);
Foo(a);  // Вызывается Foo(Asset)
```

НА ЗАМЕТКУ!

Если привести `Asset` к `dynamic` (см. раздел "Динамическое связывание" на стр. 196), тогда решение о том, какая перегруженная версия должна вызываться, откладывается до стадии выполнения, и выбор будет основан на фактическом типе объекта.

Тип `object`

Тип `object` (`System.Object`) представляет собой первоначальный базовый класс для всех типов. Любой тип может быть неявно приведен вверх к `object`.

Чтобы проиллюстрировать, насколько это полезно, рассмотрим универсальный *стек*. Стек является структурой данных, работа которой основана на принципе LIFO ("Last-In First-Out" — "последним пришел — первым обслужен"). Стек поддерживает две операции: *заталкивание* объекта в стек и *выталкивание* объекта из стека. Ниже показана простая реализация, которая может хранить до 10 объектов:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object o) { data[position++] = o; }
  public object Pop() { return data[--position]; }
}
```

Поскольку Stack работает с типом object, методы Push() и Pop() класса Stack можно использовать с экземплярами *любого типа*:

```
Stack stack = new Stack();
stack.Push ("элемент");
string s = (string) stack.Pop();     // Приведение вниз
Console.WriteLine (s);               //   элемент
```

object относится к ссылочным типам в силу того, что представляет собой класс. Несмотря на это, типы значений, такие как int, также можно приводить к object, а object приводить к ним. Чтобы сделать это возможным, среда CLR должна выполнить специальную работу по преодолению внутренних отличий между типами значений и ссылочными типами. Данный процесс называется *упаковкой* (boxing) и *распаковкой* (unboxing).

НА ЗАМЕТКУ!

В разделе "Обобщения" на стр. 116 будет показано, как усовершенствовать класс Stack, чтобы улучшить поддержку стеков однотипных элементов.

Упаковка и распаковка

Упаковка представляет собой действие по приведению экземпляра типа значения к экземпляру ссылочного типа. Ссылочным типом может быть либо класс object, либо интерфейс (см. раздел "Интерфейсы" на стр. 109). В следующем примере мы упаковываем int в объект:

```
int x = 9;
object obj = x;        // Упаковать int
```

Распаковка является обратной операцией, которая предусматривает приведение объекта к исходному типу значения:

```
int y = (int)obj;      // Распаковать int
```

Распаковка требует явного приведения. Исполняющая среда проверяет, соответствует ли указанный тип значения фактическому объектному типу, и генерирует исключение InvalidCastException, если это не так. Например, показан-

ный далее код приведет к генерации исключения, поскольку long не соответствует int:

```
object obj = 9;        // Для значения 9 выводится тип int
long x = (long) obj;   // Генерируется исключение
                       //   InvalidCastException
```

Тем не менее, следующий код выполняется успешно:

```
object obj = 9;
long x = (int) obj;
```

Представленный ниже код также не вызывает ошибки:

```
object obj = 3.5;                 // Для значения 3.5
                                  // выводится тип double
int x = (int) (double) obj;       // x теперь равно 3
```

В последнем примере (double) осуществляет *распаковку*, после чего (int) выполняет *числовое преобразование*.

Упаковка *копирует* экземпляр типа значения в новый объект, а распаковка *копирует* содержимое данного объекта обратно в экземпляр типа значения:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed);  // 3
```

Статическая проверка типов и проверка типов во время выполнения

В языке C# проверка типов проводится как статически (на этапе компиляции), так и во время выполнения.

Статическая проверка типов позволяет компилятору контролировать корректность программы, не выполняя ее. Показанный ниже код не скомпилируется, т.к. компилятор принудительно применяет статическую проверку типов:

```
int x = "5";
```

Проверка типов во время выполнения осуществляется средой CLR, когда происходит приведение вниз через ссылочное преобразование или распаковку:

```
object y = "5";
int z = (int) y;       // Ошибка времени выполнения,
                       //   неудача приведения вниз
```

Проверка типов во время выполнения возможна из-за того, что каждый объект в куче внутренне хранит небольшой маркер типа. Такой маркер может быть извлечен посредством вызова метода `GetType()` класса `object`.

Метод `GetType()` и операция `typeof`

Все типы в C# во время выполнения представлены с помощью экземпляра `System.Type`. Получить объект `System.Type` можно двумя основными путями: вызвать метод `GetType()` на экземпляре или воспользоваться операцией `typeof` на имени типа. Результат `GetType()` оценивается во время выполнения, а `typeof` — статически на этапе компиляции.

В классе `System.Type` предусмотрены свойства для имени типа, сборки, базового типа и т.д. Например:

```
int x = 3;

Console.Write (x.GetType().Name);     // Int32
Console.Write (typeof(int).Name);     // Int32
Console.Write (x.GetType().FullName); // System.Int32
Console.Write (x.GetType() == typeof(int));  // True
```

В `System.Type` также имеются методы, которые действуют в качестве шлюза для модели рефлексии времени выполнения. За подробной информацией обращайтесь в главу 19 книги *C# 8.0. Справочник. Полное описание языка*.

Список членов `object`

Ниже приведен список всех членов `object`:

```
public extern Type GetType();
public virtual bool Equals (object obj);
public static bool Equals (object objA, object objB);
public static bool ReferenceEquals (object objA,
                                    object objB);
public virtual int GetHashCode();
public virtual string ToString();
protected virtual void Finalize();
protected extern object MemberwiseClone();
```

Методы Equals(), ReferenceEquals() и GetHashCode()

Метод Equals() в классе object похож на операцию == за исключением того, что Equals() является виртуальным, а операция == — статической. Разница демонстрируется в следующем примере:

```
object x = 3;
object y = 3;
Console.WriteLine (x == y);          // False
Console.WriteLine (x.Equals (y));    // True
```

Поскольку переменные x и y были приведены к типу object, компилятор производит статическую привязку к операции == класса object, которая применяет семантику *ссылочного типа* для сравнения двух экземпляров. (Из-за того, что x и y упакованы, они находятся в разных ячейках памяти, поэтому не равны.) Однако виртуальный метод Equals() полагается на метод Equals() типа Int32, который при сравнении двух значений использует семантику *типов значений*.

Статический метод object.Equals() просто вызывает виртуальный метод Equals() на первом аргументе — после проверки, не равны ли аргументы null:

```
object x = null, y = 3;
bool error = x.Equals (y); // Ошибка во время выполнения!
bool ok = object.Equals (x, y);  // Работает нормально
                        // (ok получает значение false)
```

Метод ReferenceEquals() принудительно применяет сравнение эквивалентности ссылочных типов (что иногда удобно для ссылочных типов, в которых операция == была перегружена для выполнения другого действия).

Метод GetHashCode() выдает хеш-код, который подходит для использования со словарями, основанными на хеш-таблицах, а именно — System.Collections.Generic.Dictionary и System.Collections.Hashtable.

Чтобы настроить семантику эквивалентности типа, потребуется, как минимум, переопределить методы Equals() и GetHashCode(). Обычно также перегружаются операции == и !=. Пример такой настройки приведен в разделе "Перегрузка операций" на стр. 204.

Метод ToString()

Метод `ToString()` возвращает стандартное текстовое представление экземпляра типа. Этот метод переопределен во всех встроенных типах:

```
string s1 = 1.ToString();       // s1 равно "1"
string s2 = true.ToString();    // s2 равно "True"
```

Переопределить метод `ToString()` в специальных типах можно следующим образом:

```
public override string ToString() => "Foo";
```

Структуры

Структура похожа на класс, но обладает следующими ключевыми отличиями.

- Структура является типом значения, тогда как класс — ссылочным типом.
- Структура не поддерживает наследование (за исключением того, что она неявно порождена от `object`, или точнее — от `System.ValueType`).

Структура может иметь все те же члены, что и класс, кроме конструктора без параметров, инициализаторов полей, финализатора и виртуальных или защищенных членов.

Структура подходит там, где желательно иметь семантику типа значения. Хорошими примерами могут служить числовые типы, для которых более естественным способом присваивания является копирование значения, а не ссылки. Поскольку структура представляет собой тип значения, каждый экземпляр не требует создания объекта в куче (и последующую сборку мусора); это дает ощутимую экономию при создании большого количества экземпляров типа.

Как и любой тип значения, структура может косвенно оказаться в куче, либо из-за упаковки, либо оттого, что она выступает в качестве поля в классе. Если бы мы создали экземпляр класса `SomeClass` в показанном далее примере, то поле `Y` ссылалось бы на структуру в куче:

```
struct SomeStruct { public int X; }
class SomeClass { public SomeStruct Y; }
```

Подобным же образом, если бы мы объявили массив из SomeStruct, то экземпляр был бы сохранен в куче (т.к. массивы являются ссылочными типами), хотя целый массив потребовал бы только одного выделения памяти.

Начиная с версии C# 7.2, к структуре можно применять модификатор ref, чтобы гарантировать ее использование только способами, которые будут помещать стьруктуру в стек. Такой прием обеспечивает возможность дальнейшей оптимизации со стороны компилятора, а также учитывает тип Span<T> (см. https://bit.ly/2LR2ctm).

Семантика конструирования структуры

Семантика конструирования структуры выглядит следующим образом.

- Существует конструктор без параметров, который нельзя неявно переопределить. Он выполняет побитовое обнуление полей структуры.
- При определении конструктора (с параметрами) структуры каждому полю должно быть явно присвоено значение.
- Инициализаторы полей в структуре не предусмотрены.

Структуры и функции readonly

Начиная с версии C# 7.2, к структуре можно применять модификатор readonly, чтобы гарантировать, что все поля будут readonly; такой прием помогает заявить о намерении и предоставляет компилятору большую свободу в плане оптимизации:

```
readonly struct Point
{
  public readonly int X, Y; // X и Y должны быть readonly
}
```

Если модификатор readonly нужно применять с большей степенью детализации, то C# 8 содействует этому, предлагая новую возможность, посредством которой модификатор readonly можно применять к *функциям* структуры. Если такая функция попытается модифицировать любое поле, тогда на этапе компиляции возникнет ошибка:

```
struct Point
{
  public int X, Y;
  public readonly void ResetX() => X = 0;  // Ошибка!
}
```

Если функция `readonly` вызывает функцию не `readonly`, то компилятор выдает предупреждение (и защищенным образом копирует структуру во избежание возможности ее изменения).

Модификаторы доступа

Для содействия инкапсуляции тип или член типа может ограничивать свою *доступность* другим типам и сборкам за счет добавления к объявлению одного из шести *модификаторов доступа*, которые описаны ниже.

`public`

Полная доступность. Это неявная доступность для членов перечисления либо интерфейса.

`internal`

Доступность только внутри содержащей сборки или в дружественных сборках. Это стандартная доступность для невложенных типов.

`private`

Доступность только внутри содержащего типа. Это стандартная доступность для членов класса или структуры.

`protected`

Доступность только внутри содержащего типа или в его подклассах.

`protected internal`

Объединение доступностей `protected` и `internal` (более *либеральная* доступность, чем `protected` или `internal` по отдельности, т.к. она делает член доступнее двумя путями).

`private protected` (начиная с версии C# 7.2)

Пересечение доступностей `protected` и `internal` (более *ограничивающая* доступность, чем `protected` или `internal` по отдельности).

В следующем примере класс Class2 доступен извне его сборки, а класс Class1 — нет:

```
class Class1 {} // Class1 является internal (по умолчанию)
public class Class2 {}
```

Класс ClassB открывает поле x другим типам в той же самой сборке, а класс ClassA — нет:

```
class ClassA { int x; }  // x является private
class ClassB { internal int x; }
```

При переопределении функции базового класса доступность должна быть такой же, как у переопределяемой функции. Компилятор препятствует любому несогласованному использованию модификаторов доступа — например, подкласс может иметь меньшую доступность, чем базовый класс, но не большую.

Дружественные сборки

Члены internal можно открывать другим *дружественным* сборкам, добавляя атрибут сборки System.Runtime.CompilerServices.InternalsVisibleTo, в котором указано имя дружественной сборки:

```
[assembly: InternalsVisibleTo ("Friend")]
```

Если дружественная сборка имеет строгое имя, тогда потребуется указать ее *полный* 160-байтовый открытый ключ. Извлечь этот ключ можно с помощью запроса LINQ — в бесплатной библиотеке примеров LINQPad для главы 3 книги *C# 8.0. Справочник. Полное описание языка* предлагается интерактивный пример.

Установление верхнего предела доступности

Тип устанавливает верхний предел доступности объявленных в нем членов. Наиболее распространенным примером такого установления является ситуация, когда есть тип internal с членами public. Например:

```
class C { public void Foo() {} }
```

Стандартная доступность internal класса C устанавливает верхний предел доступности метода Foo(), по существу делая Foo() внутренним. Распространенная причина пометки Foo() как public связана с облегчением рефакторинга, если позже будет решено изменить доступность C на public.

Интерфейсы

Интерфейс похож на класс, но обеспечивает для своих членов только спецификацию, а не реализацию (хотя начиная с версии C# 8.0, интерфейс способен предоставлять *стандартную* реализацию; см. раздел "Стандартные методы интерфейсов (C# 8)" на стр. 112). Интерфейс обладает следующими особенностями.

- Все члены интерфейса являются *неявно абстрактными*. В противоположность этому класс может предоставлять как абстрактные члены, так и конкретные члены с реализациями.
- Класс (или структура) может реализовывать *множество* интерфейсов. В отличие от этого класс может быть унаследован только от *одного* класса, а структура вообще не поддерживает наследование (за исключением порождения от `System.ValueType`).

Объявление интерфейса похоже на объявление класса, но никакой реализации для его членов не предоставляется, потому что все члены интерфейса неявно абстрактные. Такие члены будут реализованы классами и структурами, реализующими данный интерфейс. Интерфейс может содержать только методы, свойства, события и индексаторы, что совершенно неслучайно в точности соответствует членам класса, которые могут быть абстрактными.

Ниже показана упрощенная версия интерфейса `IEnumerator`, определенного в пространстве имен `System.Collections`:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Члены интерфейса всегда неявно являются `public`, и для них нельзя объявлять какие-либо модификаторы доступа. Реализация интерфейса означает предоставление реализации `public` для всех его членов:

```
internal class Countdown : IEnumerator
{
  int count = 6;
```

```
  public bool MoveNext() => count-- > 0 ;
  public object Current  => count;
  public void Reset()    => count = 6;
}
```

Объект можно неявно приводить к любому интерфейсу, который он реализует:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current + " ");  // 5 4 3 2 1 0
```

Расширение интерфейса

Интерфейсы могут быть производными от других интерфейсов. Например:

```
public interface IUndoable            { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); }
```

Интерфейс `IRedoable` "наследует" все члены интерфейса `IUndoable`.

Явная реализация членов интерфейса

Реализация множества интерфейсов иногда может приводить к конфликту между сигнатурами членов. Устранить такие конфликты можно за счет *явной реализации* члена интерфейса. Например:

```
interface I1 { void Foo(); }
interface I2 { int Foo();  }
public class Widget : I1, I2
{
  public void Foo()   // Неявная реализация
  {
    Console.Write ("Реализация I1.Foo() из Widget");
  }
  int I2.Foo()        // Явная реализация метода I2.Foo
  {
    Console.Write ("Реализация I2.Foo() из Widget");
    return 42;
  }
}
```

Поскольку интерфейсы `I1` и `I2` имеют конфликтующие сигнатуры `Foo()`, класс `Widget` явно реализует метод `Foo()` интер-

фейса I2. Такой прием позволяет этим двум методам сосуществовать в одном классе. Единственный способ вызова явно реализованного метода предусматривает приведение к его интерфейсу:

```
Widget w = new Widget();
w.Foo();         // Реализация I1.Foo() из Widget
((I1)w).Foo();   // Реализация I1.Foo() из Widget
((I2)w).Foo();   // Реализация I2.Foo() из Widget
```

Еще одной причиной явной реализации членов интерфейса может быть необходимость сокрытия членов, которые являются узкоспециализированными и нарушающими нормальный сценарий использования типа. Например, тип, который реализует `ISerializable`, обычно будет избегать демонстрации членов `ISerializable`, если только не осуществляется явное приведение к упомянутому интерфейсу.

Реализация виртуальных членов интерфейса

Неявно реализованный член интерфейса по умолчанию является запечатанным. Чтобы его можно было переопределить, он должен быть помечен в базовом классе как `virtual` или `abstract`: обращение к этому члену интерфейса через базовый класс либо интерфейс приводит к вызову его реализации из подкласса.

Явно реализованный член интерфейса не может быть помечен как `virtual`, равно как и не может быть переопределен обычным образом. Тем не менее, он может быть *реализован повторно*.

Повторная реализация члена интерфейса в подклассе

Подкласс может *повторно реализовать* любой член интерфейса, который уже реализован базовым классом. Повторная реализация перехватывает реализацию члена (при вызове через интерфейс) и работает вне зависимости от того, является ли член виртуальным в базовом классе.

В показанном ниже примере `TextBox` явно реализует `IUndoable.Undo()`, поэтому данный метод не может быть помечен как `virtual`. Чтобы "переопределить" его, класс `RichTextBox` должен повторно реализовать метод `Undo()` интерфейса `IUndoable`:

```
public interface IUndoable { void Undo(); }
public class TextBox : IUndoable
{
  void IUndoable.Undo()
    => Console.WriteLine ("TextBox.Undo()");
}
public class RichTextBox : TextBox, IUndoable
{
  public new void Undo()
    => Console.WriteLine ("RichTextBox.Undo()");
}
```

Обращение к повторно реализованному методу через интерфейс приводит к вызову его реализации из подкласса:

```
RichTextBox r = new RichTextBox();
r.Undo();                  // RichTextBox.Undo()
((IUndoable)r).Undo();     // RichTextBox.Undo()
```

В данном случае метод `Undo()` реализован явно. Неявно реализованные члены также могут быть повторно реализованы, но эффект не является всепроникающим, т.к. обращение к члену через базовый класс приводит к вызову базовой реализации.

Стандартные методы интерфейсов (C# 8)

В версии C# 8 к члену интерфейса можно добавлять стандартную реализацию, делая его необязательным для реализации:

```
interface ILogger
{
  void Log (string text) => Console.WriteLine (text);
}
```

Такая возможность полезна, когда необходимо добавить член к интерфейсу, определенному в какой-то популярной библиотеке, но не нарушить работу (потенциально тысяч) реализаций.

Стандартные реализации всегда являются явными, так что если в классе, реализующем `ILogger`, отсутствует определение метода `Log()`, то вызвать его получится только одним способом — через интерфейс:

```
class Logger : ILogger { }
...
((ILogger)new Logger()).Log ("сообщение");
```

Это предотвращает проблему наследования множества реализаций: если тот же самый стандартный член был добавлен в два интерфейса, которые реализует класс, то неоднозначность относительно того, какой член вызывается, никогда не возникнет.

Теперь в интерфейсах можно определять также и статические члены (включая поля), доступ к которым осуществляется из кода внутри стандартных реализаций:

```
interface ILogger
{
  void Log (string text) =>
    Console.WriteLine (Prefix + text);
  static string Prefix = "";
}
```

Поскольку члены интерфейсов неявно открыты, получать доступ к статическим членам можно также снаружи:

```
ILogger.Prefix = "Журнальный файл: ";
```

Можно ввести ограничение, добавив к статическому члену интерфейса модификатор доступа (такой как `private`, `protected` или `internal`).

Поля экземпляров (по-прежнему) запрещены, что соответствует принципу интерфейсов, предусматривающему определение ими *поведения*, но не *состояния*.

Перечисления

Перечисление — это специальный тип значения, который позволяет указывать группу именованных числовых констант. Например:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Данное перечисление можно применять следующим образом:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);  // true
```

Каждый член перечисления имеет лежащее в его основе целочисленное значение. По умолчанию лежащие в основе значения относятся к типу `int`, а членам перечисления присваиваются константы 0, 1, 2... (в порядке их объявления). Можно указать другой целочисленный тип:

```
public enum BorderSide : byte { Left,Right,Top,Bottom }
```

Для каждого члена перечисления можно также указывать явные лежащие в основе значения:

```
public enum BorderSide : byte
  { Left=1, Right=2, Top=10, Bottom=11 }
```

Вдобавок компилятор позволяет явно присваивать значения *определенным* членам перечисления. Члены, значения которым не были присвоены, получают значения на основе инкрементирования последнего явно указанного значения. Предыдущий пример эквивалентен следующему коду:

```
public enum BorderSide : byte
  { Left=1, Right, Top=10, Bottom }
```

Преобразования перечислений

Экземпляр перечисления может быть преобразован в и из лежащего в основе целочисленного значения с помощью явного приведения:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

Один тип перечисления можно также явно приводить к другому; при трансляции между типами перечислений используются лежащие в их основе целочисленные значения.

Числовой литерал 0 трактуется особым образом в том смысле, что не требует явного приведения:

```
BorderSide b = 0;  // Приведение не требуется
if (b == 0) ...
```

В данном примере `BorderSide` не имеет членов с целочисленным значением 0. Это вовсе не приводит к ошибке: ограничение перечислений связано с тем, что компилятор и среда CLR не препятствуют присваиванию целых чисел, значения которых выходят за пределы диапазона членов:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);  // 12345
```

Перечисления флагов

Члены перечислений можно комбинировать. Чтобы предотвратить неоднозначности, члены комбинируемого перечисления

требуют явного присваивания значений, обычно являющихся степенью двойки. Например:

```
[Flags]
public enum BorderSides
  { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

По соглашению типу комбинируемого перечисления назначается имя во множественном, а не единственном числе. Для работы со значениями комбинируемого перечисления применяются побитовые операции, такие как | и &. Они действуют на лежащих в основе целочисленных значениях:

```
BorderSides leftRight =
  BorderSides.Left | BorderSides.Right;
if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Включает Left"); // Включает Left
string formatted = leftRight.ToString(); //"Left, Right"
BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight);      // True
```

К типам комбинируемых перечислений должен быть применен атрибут `Flags`, иначе вызов `ToString()` на экземпляре перечисления возвратит число, а не последовательность имен.

Для удобства члены комбинаций могут быть помещены в само объявление перечисления:

```
[Flags] public enum BorderSides
{
  None=0,
  Left=1, Right=2, Top=4, Bottom=8,
  LeftRight = Left | Right,
  TopBottom = Top  | Bottom,
  All       = LeftRight | TopBottom
}
```

Операции над перечислениями

Ниже указаны операции, которые могут работать с перечислениями:

```
=  ==  !=  <  >  <=  >=  +  -  ^  &  |  ~
+= -=  ++  -  sizeof
```

Побитовые, арифметические и операции сравнения возвращают результат обработки лежащих в основе целочисленных значений. Сложение разрешено для перечисления и целочисленного типа, но не для двух перечислений.

Вложенные типы

Вложенный тип объявляется внутри области видимости другого типа. Например:

```
public class TopLevel
{
  public class Nested { }              // Вложенный класс
  public enum Color { Red, Blue, Tan }  // Вложенное
                                        // перечисление
}
```

Вложенный тип обладает следующими характеристиками.

- Он может получать доступ к закрытым членам включающего типа и ко всему остальному, к чему имеет доступ включающий тип.
- Он может быть объявлен с полным диапазоном модификаторов доступа, а не только `public` и `internal`.
- Стандартной доступностью вложенного типа является `private`, а не `internal`.
- Доступ к вложенному типу извне требует указания имени включающего типа (как при обращении к статическим членам).

Например, для доступа к члену `Color.Red` извне класса `TopLevel` необходимо записать так:

```
TopLevel.Color color = TopLevel.Color.Red;
```

Все типы могут быть вложенными, но *содержать* вложенные типы могут только классы и структуры.

Обобщения

В C# имеются два отдельных механизма для написания кода, многократно используемого различными типами: *наследование*

и *обобщения*. В то время как наследование выражает повторное использование с помощью базового типа, обобщения делают это посредством "шаблона", который содержит "типы-заполнители". Обобщения в сравнении с наследованием могут *увеличивать безопасность типов*, а также *сокращать количество приведений и упаковок*.

Обобщенные типы

Обобщенный тип объявляет *параметры типа* — типы-заполнители, предназначенные для замещения потребителем обобщенного типа, который предоставляет *аргументы типа*. Ниже показан обобщенный тип `Stack<T>`, предназначенный для реализации стека экземпляров типа T. В `Stack<T>` объявлен единственный параметр типа T:

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj) => data[position++] = obj;
  public T Pop()           => data[--position];
}
```

Применять `Stack<T>` можно следующим образом:

```
var stack = new Stack<int>();
stack.Push (5);
stack.Push (10);
int x = stack.Pop();  // x имеет значение 10
int y = stack.Pop();  // y имеет значение 5
```

НА ЗАМЕТКУ!

Обратите внимание, что в последних двух строках кода приведение вниз не требуется. Это позволяет избежать возможной ошибки во время выполнения и устраняет непроизводительные затраты на упаковку/распаковку. В результате наш обобщенный стек получает преимущество над необобщенным стеком, в котором на месте T используется тип `object` (пример приведен в разделе "Тип `object`" на стр. 100).

Класс Stack<int> замещает параметр типа T аргументом типа int, неявно создавая тип на лету (синтез происходит во время выполнения). Фактически Stack<int> имеет показанное ниже определение (подстановки выделены полужирным, а во избежание путаницы вместо имени класса указано ###):

```
public class ###
{
  int position;
  int[] data = new int[100];
  public void Push (int obj) => data[position++] = obj;
  public int Pop() => data[--position];
}
```

Формально мы говорим, что Stack<T> — это *открытый (open) тип*, а Stack<int> — *закрытый (closed) тип*. Во время выполнения все экземпляры обобщенных типов закрываются — с замещением их типов-заполнителей.

Обобщенные методы

Обобщенный метод объявляет параметры типа внутри сигнатуры метода. С помощью обобщенных методов многие фундаментальные алгоритмы могут быть реализованы единственным универсальным способом. Ниже показан обобщенный метод, который меняет местами содержимое двух переменных любого типа T:

```
static void Swap<T> (ref T a, ref T b)
{
  T temp = a; a = b; b = temp;
}
```

Метод Swap<T> можно применять следующим образом:

```
int x = 5, y = 10;
Swap (ref x, ref y);
```

Как правило, предоставлять аргументы типа обобщенному методу нет нужды, поскольку компилятор способен неявно вывести тип. Если имеется неоднозначность, то обобщенные методы могут быть вызваны с аргументами типа:

```
Swap<int> (ref x, ref y);
```

Внутри обобщенного *типа* метод не рассматривается как обобщенный до тех пор, пока он сам не *введет* параметры типа

(посредством синтаксиса с угловыми скобками). Метод `Pop()` в нашем обобщенном стеке просто потребляет существующий параметр типа `T` и не классифицируется как обобщенный.

Методы и типы — единственные конструкции, в которых могут вводиться параметры типа. Свойства, индексаторы, события, поля, конструкторы, операции и т.д. не могут объявлять параметры типа, хотя способны пользоваться любыми параметрами типа, которые уже объявлены во включающем типе. В примере с обобщенным стеком можно было бы написать индексатор, который возвращает обобщенный элемент:

```
public T this [int index] { get { return data[index]; } }
```

Аналогично конструкторы также могут пользоваться существующими параметрами типа, но не *вводить* их.

Объявление параметров типа

Параметры типа могут быть введены в объявлениях классов, структур, интерфейсов, делегатов (см. раздел "Делегаты" на стр. 125) и методов. Можно указывать множество параметров типа, разделяя их запятыми:

```
class Dictionary<TKey, TValue> {...}
```

Вот как создать его экземпляр:

```
var myDict = new Dictionary<int,string>();
```

Имена обобщенных типов и методов могут быть перегружены при условии, что количество параметров типа у них отличается. Например, показанные ниже три имени типов не конфликтуют друг с другом:

```
class A {}
class A<T> {}
class A<T1,T2> {}
```

НА ЗАМЕТКУ!

По соглашению обобщенные типы и методы с *единственным* параметром типа обычно именуют его как `T`, если назначение параметра очевидно. В случае *нескольких* параметров типа каждый такой параметр имеет более описательное имя (с префиксом `T`).

Операция typeof и несвязанные обобщенные типы

Во время выполнения открытых обобщенных типов не существует: они закрываются на этапе компиляции. Однако во время выполнения возможно существование *несвязанного* (unbound) обобщенного типа — исключительно как объекта Type. Единственным способом указания несвязанного обобщенного типа в C# является применение операции typeof:

```
class A<T> {}
class A<T1,T2> {}
...
Type a1 = typeof (A<>); // Несвязанный тип
Type a2 = typeof (A<,>); // Указание двух аргументов типа
Console.Write (a2.GetGenericArguments().Count()); // 2
```

Операцию typeof можно также использовать для указания закрытого типа:

```
Type a3 = typeof (A<int,int>);
```

или открытого типа (который закроется во время выполнения):

```
class B<T> { void X() { Type t = typeof (T); } }
```

Стандартное обобщенное значение

Ключевое слово default может применяться для получения стандартного значения обобщенного параметра типа. Стандартным значением для ссылочного типа является null, а для типа значения — результат побитового обнуления полей в этом типе:

```
static void Zap<T> (T[] array)
{
  for (int i = 0; i < array.Length; i++)
    array[i] = default(T);
}
```

Начиная с версии C# 7.1, аргумент типа можно не указывать в случаях, когда компилятор способен его вывести самостоятельно:

```
array[i] = default;
```

Ограничения обобщений

По умолчанию параметр типа может быть замещен любым типом. Чтобы затребовать более специфичные аргументы типа, к параметру типа можно применить *ограничения*. Существуют восемь видов ограничений:

```
where T : базовый-класс  // Ограничение базового класса
where T : интерфейс      // Ограничение интерфейса
where T : class          // Ограничение ссылочного типа
where T : class?         // (См. раздел "Ссылочные типы,
                  // допускающие значение null (C# 8)".)
where T : struct         // Ограничение типа значения
where T : unmanaged      // Ограничение неуправляемого типа
where T : new()          // Ограничение конструктора
                         //   без параметров
where U : T              // Неприкрытое ограничение типа
```

В следующем примере `GenericClass<T,U>` требует, чтобы тип `T` был производным от класса `SomeClass` (или идентичен ему) и реализовал интерфейс `Interface1`, а тип `U` предоставлял конструктор без параметров:

```
class SomeClass {}
interface Interface1 {}
class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{ ... }
```

Ограничения могут применяться везде, где определены параметры типа, как в методах, так и в определениях типов.

Ограничение базового класса указывает, что параметр типа должен быть подклассом заданного класса (или совпадать с ним); *ограничение интерфейса* указывает, что параметр типа должен реализовывать данный интерфейс. Такие ограничения позволяют экземплярам параметра типа быть неявно преобразуемыми в этот класс или интерфейс.

Ограничение `class` и *ограничение* `struct` указывают, что `T` должен быть ссылочным типом или типом значения (не допускающим `null`). *Ограничение* `unmanaged` является более сильной версией ограничения `struct`: тип `T` должен быть простым типом значения или структурой, которая (рекурсивно) не имеет ссылочных типов. *Ограничение конструктора без параметров* требует, что-

бы тип T имел открытый конструктор без параметров и позволял вызывать операцию new на T:

```
static void Initialize<T> (T[] array) where T : new()
{
  for (int i = 0; i < array.Length; i++)
    array[i] = new T();
}
```

Неприкрытое ограничение типа требует, чтобы один параметр типа был производным от другого параметра типа (или совпадал с ним).

Создание подклассов для обобщенных типов

Подклассы для обобщенного класса можно создавать точно так же, как в случае необобщенного класса. Подкласс может оставлять параметры типа базового класса открытыми, как показано в следующем примере:

```
class Stack<T>                    {...}
class SpecialStack<T> : Stack<T> {...}
```

Либо же подкласс может закрыть параметры обобщенного типа посредством конкретного типа:

```
class IntStack : Stack<int> {...}
```

Подкласс может также вводить новые аргументы типа:

```
class List<T>                      {...}
class KeyedList<T,TKey> : List<T> {...}
```

Самоссылающиеся объявления обобщений

Тип может указывать *самого себя* в качестве конкретного типа при закрытии аргумента типа:

```
public interface IEquatable<T> { bool Equals (T obj); }
public class Balloon : IEquatable<Balloon>
{
  public bool Equals (Balloon b) { ... }
}
```

Следующий код также допустим:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T>         { ... }
```

Статические данные

Статические данные являются уникальными для каждого закрытого типа:

```
class Bob<T> { public static int Count; }
...
Console.WriteLine (++Bob<int>.Count);        // 1
Console.WriteLine (++Bob<int>.Count);        // 2
Console.WriteLine (++Bob<string>.Count);     // 1
Console.WriteLine (++Bob<object>.Count);     // 1
```

Ковариантность

НА ЗАМЕТКУ!

Ковариантность и контравариантность — сложные концепции. Мотивация, лежащая в основе их введения в язык C#, заключалась в том, чтобы позволить обобщенным интерфейсам и обобщениям (в частности, определенным в .NET, таким как `IEnumerable<T>`) работать *более предсказуемым образом*. Вы можете извлечь выгоду из ковариантности и контравариантности, даже особо не вникая во все их детали.

Предполагая, что тип `A` может быть преобразован в `B`, тип `X` имеет ковариантный параметр типа, если `X<A>` поддается преобразованию в `X<B>`.

(Согласно понятию вариантности в C#, "поддается преобразованию" означает возможность преобразования через *неявное ссылочное преобразование* — такое как *A является подклассом B* или *A реализует B*. Сюда не входят числовые преобразования, упаковывающие преобразования и специальные преобразования.)

Например, тип `IFoo<T>` имеет ковариантный тип `T`, если справедливо следующее:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

Интерфейсы (и делегаты) допускают ковариантные параметры типа. В целях иллюстрации предположим, что класс `Stack<T>`,

который был написан в начале настоящего раздела, реализует показанный ниже интерфейс:

```
public interface IPoppable<out T> { T Pop(); }
```

Модификатор `out` для `T` указывает, что тип `T` используется только в *выходных позициях* (например, в возвращаемых типах для методов) и помечает параметр типа как *ковариантный*, разрешая написание такого кода:

```
// Предполагается, что Bear является подклассом Animal:
var bears = new Stack<Bear>();
bears.Push (new Bear());

// Поскольку bears реализует IPoppable<Bear>,
// его можно преобразовать в IPoppable<Animal>:
IPoppable<Animal> animals = bears;  // Допустимо
Animal a = animals.Pop();
```

Приведение `bears` к `animals` разрешено компилятором — в силу того, что параметр типа в интерфейсе является ковариантным.

НА ЗАМЕТКУ!

Интерфейсы `IEnumerator<T>` и `IEnumerable<T>` (см. раздел "Перечисление и итераторы" на стр. 152) помечены как имеющие ковариантный тип `T`. Это позволяет, например, приводить `IEnumerable<string>` к `IEnumerable<object>`.

Компилятор генерирует ошибку, если ковариантный параметр типа встречается во *входной* позиции (скажем, в параметре метода или в записываемом свойстве). Цель такого ограничения — гарантировать безопасность типов на этапе компиляции. Например, оно предотвращает добавление к этому интерфейсу метода `Push(T)`, который потребители могли бы неправильно применять для внешне безобидной операции заталкивания объекта, представляющего верблюда (camel), в реализацию `IPoppable<Animal>` (вспомните, что основным типом в нашем примере является стек объектов, представляющих медведей (bear)). Чтобы можно было определить метод `Push(T)`, параметр типа `T` в действительности должен быть *контравариантным*.

НА ЗАМЕТКУ!

В языке C# ковариантность (и контравариантность) поддерживается только для элементов со *ссылочными преобразованиями*, но не *упаковывающими преобразованиями*. Таким образом, если имеется метод, который принимает параметр типа `IPoppable<object>`, то его можно вызывать с `IPoppable<string>`, но не с `IPoppable<int>`.

Контравариантность

Как было показано ранее, если предположить, что `A` разрешает неявное ссылочное преобразование в `B`, то тип `X` имеет ковариантный параметр типа, когда `X<A>` допускает ссылочное преобразование в `X<B>`. Тип будет *контравариантным*, если возможно преобразование в обратном направлении — из `X<B>` в `X<A>`. Это поддерживается интерфейсами и делегатами, когда параметр типа встречается только во *входных* позициях, обозначенных с помощью модификатора `in`. Продолжая предыдущий пример, если класс `Stack<T>` реализует следующий интерфейс:

```
public interface IPushable<in T> { void Push (T obj); }
```

то вполне законно поступать так:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;  // Допустимо
bears.Push (new Bear());
```

Зеркально отражая ковариантность, компилятор сообщит об ошибке, если вы попытаетесь использовать контравариантный параметр типа в выходной позиции (например, в качестве возвращаемого значения или в читаемом свойстве).

Делегаты

Делегат связывает компонент, который вызывает метод, с его целевым методом во время выполнения. К делегатам применимы два аспекта: *тип* и *экземпляр*. *Тип делегата* определяет *протокол*, которому будут соответствовать вызывающий компонент и целевой метод; протокол включает список типов параметров и

возвращаемый тип. *Экземпляр делегата* — это объект, который ссылается на один (или более) целевых методов, удовлетворяющих данному протоколу.

Экземпляр делегата действует в вызывающем компоненте буквально как посредник: вызывающий компонент обращается к делегату, после чего делегат вызывает целевой метод. Такая косвенность отвязывает вызывающий компонент от целевого метода.

Объявление типа делегата предваряется ключевым словом `delegate`, но в остальном напоминает объявление (абстрактного) метода. Например:

```
delegate int Transformer (int x);
```

Чтобы создать экземпляр делегата, переменной делегата можно присвоить метод:

```
class Test
{
  static void Main()
  {
    Transformer t = Square; // Создать экземпляр делегата
    int result = t(3);          // Вызвать делегат
    Console.Write (result);     // 9
  }
  static int Square (int x) => x * x;
}
```

Вызов делегата очень похож на вызов метода (т.к. целью делегата является всего лишь обеспечение определенного уровня косвенности):

```
t(3);
```

Оператор `Transformer t = Square;` представляет собой сокращение для следующего оператора:

```
Transformer t = new Transformer (Square);
```

Вдобавок `t(3)` — сокращение для такого вызова:

```
t.Invoke (3);
```

Делегат похож на *обратный вызов* — общий термин, который охватывает конструкции, подобные указателям на функции C.

Написание подключаемых методов с помощью делегатов

Метод присваивается переменной делегата во время выполнения. Это удобно, когда нужно написать подключаемые методы. В следующем примере присутствует служебный метод по имени `Transform()`, который применяет трансформацию к каждому элементу в целочисленном массиве. Метод `Transform()` имеет параметр делегата, предназначенный для указания подключаемой трансформации.

```
public delegate int Transformer (int x);
class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Transform (values, Square);
    foreach (int i in values)
      Console.Write (i + " ");  // 1 4 9
  }
  static void Transform (int[] values, Transformer t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }
  static int Square (int x) => x * x;
}
```

Групповые делегаты

Все экземпляры делегатов обладают возможностью *группового вызова* (multicast). Это значит, что экземпляр делегата может ссылаться не только на одиночный целевой метод, но также и на список целевых методов. Экземпляры делегатов комбинируются с помощью операций + и +=. Например:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

Последняя строка функционально эквивалентна такой строке:

```
d = d + SomeMethod2;
```

Обращение к d теперь приведет к вызову методов SomeMethod1() и SomeMethod2(). Делегаты вызываются в порядке, в котором они добавлялись.

Операции - и -= удаляют правый операнд делегата из левого операнда делегата, например:

```
d -= SomeMethod1;
```

Обращение к d теперь приведет к вызову только метода SomeMethod2().

Применение операции + или += к переменной делегата со значением null законно, как и применение операции -= к переменной делегата с единственным целевым методом (в результате чего экземпляр делегата получает значение null).

НА ЗАМЕТКУ!

Делегаты являются *неизменяемыми*, так что при использовании операции += или -= фактически создается *новый* экземпляр делегата, который присваивается существующей переменной.

Если групповой делегат имеет возвращаемый тип, отличающийся от void, тогда вызывающий компонент получает возвращаемое значение из последнего вызванного метода. Предшествующие методы по-прежнему вызываются, но их возвращаемые значения отбрасываются. В большинстве сценариев групповые делегаты имеют возвращаемые типы void, поэтому такая тонкая ситуация не возникает.

Все типы делегатов неявно порождены от класса System.MulticastDelegate, который унаследован от System.Delegate. Операции +, -, += и -=, выполняемые над делегатом, транслируются в вызовы статических методов Combine() и Remove() класса System.Delegate.

Целевые методы экземпляра и целевые статические методы

Когда объекту делегата присваивается метод *экземпляра*, объект делегата должен поддерживать ссылку не только на метод,

но также и на *экземпляр*, которому данный метод принадлежит. Экземпляр представлен свойством `Target` класса `System.Delegate` (значением которого будет `null`, если делегат ссылается на статический метод).

Обобщенные типы делегатов

Тип делегата может содержать параметры обобщенного типа. Например:

```
public delegate T Transformer<T> (T arg);
```

Ниже показано, как можно было применить этот тип делегата:

```
static double Square (double x) => x * x;

static void Main()
{
  Transformer<double> s = Square;
  Console.WriteLine (s (3.3));  // 10.89
}
```

Делегаты `Func` и `Action`

Благодаря обобщенным делегатам становится возможным написание небольшого набора типов делегатов, которые являются настолько универсальными, что способны работать с методами, имеющими любой возвращаемый тип и любое (приемлемое) количество аргументов. Такими делегатами являются `Func` и `Action`, определенные в пространстве имен `System` (модификаторы `in` и `out` указывают *вариантность*, которая вскоре будет раскрыта в контексте делегатов):

```
delegate TResult Func <out TResult> ();
delegate TResult Func <in T, out TResult> (T arg);
delegate TResult Func <in T1, in T2, out TResult>
  (T1 arg1, T2 arg2);
... и так далее вплоть до T16

delegate void Action ();
delegate void Action <in T> (T arg);
delegate void Action <in T1, in T2>
  (T1 arg1, T2 arg2);
... и так далее вплоть до T16
```

Представленные делегаты исключительно универсальны. Делегат Transformer в предыдущем примере может быть заменен делегатом Func, который принимает один аргумент типа T и возвращает значение того же самого типа:

```
public static void Transform<T> (
  T[] values, Func<T,T> transformer)
{
  for (int i = 0; i < values.Length; i++)
    values[i] = transformer (values[i]);
}
```

Делегаты Func и Action не покрывают лишь практические сценарии, связанные с параметрами ref/out и параметрами указателей.

Совместимость делегатов

Все типы делегатов несовместимы друг с другом, даже если они имеют одинаковые сигнатуры:

```
delegate void D1(); delegate void D2();
...
D1 d1 = Method1;
D2 d2 = d1;  // Ошибка на этапе компиляции
```

Тем не менее, следующий код разрешен:

```
D2 d2 = new D2 (d1);
```

Экземпляры делегатов считаются равными, если они имеют одинаковые типы и целевые методы (или метод). Для групповых делегатов важен порядок следования целевых методов.

Вариантность возвращаемых типов

В результате вызова метода можно получить обратно тип, который является более специфическим, чем запрошенный. Это обычное *полиморфное поведение*. В соответствии с таким поведением целевой метод делегата может возвращать *более* специфический тип, чем описанный самим делегатом. Это *ковариантность*:

```
delegate object ObjectRetriever();
...
static void Main()
{
  ObjectRetriever o = new ObjectRetriever (GetString);
  object result = o();
  Console.WriteLine (result);  // поход
}
static string GetString() => "поход";
```

Делегат `ObjectRetriever` ожидает получить обратно `object`, но подойдет также и *подкласс* `object`, потому что возвращаемые типы делегатов являются *ковариантными*.

Вариантность параметров

При вызове метода можно предоставлять аргументы, которые имеют более специфический тип, чем параметры данного метода. Это обычное *полиморфное поведение*. В соответствии с таким поведением у целевого метода делегата могут быть *менее* специфические типы параметров, чем описанные самим делегатом. Это называется *контравариантностью*:

```
delegate void StringAction (string s);
...
static void Main()
{
  StringAction sa = new StringAction (ActOnObject);
  sa ("будущее");  // Выводит "будущее"
}
static void ActOnObject (object o) => Console.Write (o);
```

НА ЗАМЕТКУ!

Стандартный шаблон событий спроектирован так, чтобы использовать в своих интересах контравариантность параметров делегата за счет применения общего базового класса `EventArgs`. Например, можно иметь единственный метод, вызываемый двумя разными делегатами, одному из которых передается `MouseEventArgs`, а другому — `KeyEventArgs`.

Вариантность параметров типа для обобщенных делегатов

В разделе "Обобщения" на стр. 116 было указано, что параметры типа для обобщенных интерфейсов могут быть ковариантными и контравариантными. Аналогичная возможность существует также для обобщенных делегатов. При определении обобщенного типа делегата рекомендуется поступать следующим образом:

- помечать параметр типа, используемый только для возвращаемого значения, как ковариантный (`out`);
- помечать любой параметр типа, применяемый только для параметров, как контравариантный (`in`).

Это позволяет преобразованиям работать естественным образом, соблюдая отношения наследования между типами. Следующий делегат (определенный в пространстве имен `System`) является ковариантным для `TResult`:

```
delegate TResult Func<out TResult>();
```

разрешая записывать так:

```
Func<string> x = ...;
Func<object> y = x;
```

Представленный ниже делегат (определенный в пространстве имен `System`) является контравариантным для `T`:

```
delegate void Action<in T> (T arg);
```

позволяя записывать следующим образом:

```
Action<object> x = ...;
Action<string> y = x;
```

События

Когда используются делегаты, обычно возникают две независимые роли: *ретранслятор* и *подписчик*. *Ретранслятор* — это тип, который содержит поле делегата. Ретранслятор решает, когда делать передачу, вызывая делегат. *Подписчики* — это целевые методы-получатели. Подписчик решает, когда начинать и останавливать прослушивание, вызывая операции `+=` и `-=` на делегате ретранслятора. Подписчик ничего не знает о других подписчиках и не вмешивается в их работу.

События являются языковым средством, формализующим описанный шаблон. Конструкция `event` открывает только подмножество возможностей делегата, которые требуются для модели "ретранслятор/подписчик". Основной замысел событий — *предотвратить влияние подписчиков друг на друга.*

Объявить событие проще всего, поместив ключевое слово `event` перед членом делегата:

```
public class Broadcaster
{
  public event ProgressReporter Progress;
}
```

Код внутри типа Broadcaster имеет полный доступ к члену PriceChanged и может трактовать его как делегат. Код за пределами Broadcaster может только выполнять операции += и -= над событием PriceChanged.

В следующем примере класс Stock запускает свое событие PriceChanged каждый раз, когда изменяется свойство Price данного класса:

```
public delegate void PriceChangedHandler
  (decimal oldPrice, decimal newPrice);
public class Stock
{
  string symbol; decimal price;
  public Stock (string symbol) => this.symbol = symbol;
  public event PriceChangedHandler PriceChanged;
  public decimal Price
  {
    get => price;
    set
    {
      if (price == value) return;
      // Если список вызова не пуст,
      //   тогда запустить событие:
      if (PriceChanged != null)
        PriceChanged (price, value);
      price = value;
    }
  }
}
```

Если в приведенном выше примере убрать ключевое слово event, чтобы PriceChanged стало обычным полем делегата, то результаты окажутся теми же самыми. Однако класс Stock станет менее надежным в том, что подписчики смогут предпринимать следующие действия, влияя друг на друга:

- заменять других подписчиков, переустанавливая PriceChanged (вместо применения операции +=);
- очищать всех подписчиков (путем установки PriceChanged в null);
- выполнять групповую рассылку другим подписчикам, вызывая делегат.

События могут быть виртуальными, переопределенными, абстрактными или запечатанными. Они также могут быть статическими.

Стандартный шаблон событий

Почти во всех случаях, когда события определяются в библиотеке .NET Core, их определения придерживаются стандартного шаблона, предназначенного для обеспечения согласованности между библиотекой и пользовательским кодом. Ниже показан предыдущий пример, переделанный с учетом данного шаблона:

```
public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice, NewPrice;
  public PriceChangedEventArgs (decimal lastPrice,
                                decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}
public class Stock
{
  string symbol; decimal price;
  public Stock (string symbol) => this.symbol = symbol;
  public event EventHandler<PriceChangedEventArgs>
               PriceChanged;
  protected virtual void OnPriceChanged
                    (PriceChangedEventArgs e) =>
    // Сокращение для вызова PriceChanged,
    //   если не равно null:
    PriceChanged?.Invoke (this, e);
  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      OnPriceChanged (new PriceChangedEventArgs
                          (price, value));
      price = value;
    }
  }
}
```

В основе стандартного шаблона событий находится `System.EventArgs` — предопределенный класс .NET с единственным членом — статическим свойством `Empty`. Базовый класс `EventArgs` предназначен для передачи информации событию. В приведенном выше примере класса `Stock` мы создаем подкласс `EventArgs` для передачи старого и нового значений цены, когда инициируется событие `PriceChanged`.

Обобщенный делегат `System.EventHandler` также является частью .NET Core и определен следующим образом:

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e)
   where TEventArgs : EventArgs;
```

НА ЗАМЕТКУ!

До выхода версии C# 2.0 (где в язык были добавлены обобщения) решение предусматривало написание специального делегата обработки событий для каждого типа `EventArgs`:

```
delegate void PriceChangedHandler
  (object sender,
   PriceChangedEventArgs e);
```

По историческим причинам большинство событий в библиотеках .NET используют делегаты, определенные подобным образом.

Центральным местом генерации событий является защищенный виртуальный метод по имени `On-`*`имя-события`*`()`. Это позволяет подклассам запускать событие (что обычно желательно) и также вставлять код до и после генерации события.

Вот как можно было бы применить класс `Stock`:

```
static void Main()
{
  Stock stock = new Stock ("THPW");
  stock.Price = 27.10M;

  stock.PriceChanged += stock_PriceChanged;
  stock.Price = 31.59M;
}
```

```
static void stock_PriceChanged
  (object sender, PriceChangedEventArgs e)
{
  if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
   Console.WriteLine
              ("Внимание, цена увеличилась на 10%!");
}
```

Для событий, которые не содержат в себе дополнительную информацию, в .NET также предлагается необобщенный делегат `EventHandler`. Для демонстрации его использования можно переписать класс `Stock` так, чтобы событие `PriceChanged` инициировалось *после* изменения цены. Это означает, что с событием не нужно передавать какую-либо дополнительную информацию:

```
public class Stock
{
  string symbol; decimal price;

  public Stock (string symbol) => this.symbol = symbol;

  public event EventHandler PriceChanged;

  protected virtual void OnPriceChanged (EventArgs e) =>
    PriceChanged?.Invoke (this, e);

  public decimal Price
  {
    get => price;
    set
    {
      if (price == value) return;
      price = value;
      OnPriceChanged (EventArgs.Empty);
    }
  }
}
```

Обратите внимание на применение свойства `EventArgs.Empty`, что позволяет не создавать экземпляр `EventArgs`.

Средства доступа к событию

Средства доступа к событию представляют собой реализации его операций `+=` и `-=`. По умолчанию средства доступа реализуются неявно компилятором. Взгляните на следующее объявление события:

```
public event EventHandler PriceChanged;
```

Компилятор преобразует его в перечисленные ниже компоненты:

- закрытое поле делегата;
- пара открытых функций доступа к событию, реализации которых направляют операции += и -= к закрытому полю делегата.

Контроль над таким процессом можно взять на себя, определив *явные* средства доступа. Вот как выглядит ручная реализация события PriceChanged из предыдущего примера:

```
EventHandler priceChanged;  // Закрытый делегат
public event EventHandler PriceChanged
{
  add    { priceChanged += value; }
  remove { priceChanged -= value; }
}
```

Приведенный пример функционально идентичен стандартной реализации средств доступа C# (за исключением того, что компилятор C# также обеспечивает безопасность в отношении потоков во время обновления делегата). Определяя средства доступа к событию самостоятельно, мы указываем на то, что генерировать стандартное поле и логику средств доступа не требуется.

С помощью явных средств доступа к событию можно реализовать более сложные стратегии хранения и доступа для лежащего в основе делегата. Это полезно, когда средства доступа к событию просто поручают передачу события другому классу или когда явно реализуется интерфейс, в котором объявляется событие:

```
public interface IFoo { event EventHandler Ev; }
class Foo : IFoo
{
  EventHandler ev;
  event EventHandler IFoo.Ev
  {
    add { ev += value; } remove { ev -= value; }
  }
}
```

Лямбда-выражения

Лямбда-выражение представляет собой неименованный метод, записанный вместо экземпляра делегата. Компилятор немедленно

преобразовывает лямбда-выражение в одну из двух описанных ниже конструкций.

- Экземпляр делегата.
- *Дерево выражения*, которое имеет тип `Expression <TDelegate>` и представляет код внутри лямбда-выражения в виде объектной модели, поддерживающей обход. Оно делает возможной интерпретацию лямбда-выражения позже во время выполнения (весь процесс подробно описан в главе 8 книги *C# 8.0. Справочник. Полное описание языка*).

При наличии следующего типа делегата:

```
delegate int Transformer (int i);
```

вот как можно присвоить и обратиться к лямбда-выражению `x => x * x`:

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3));  // 9
```

НА ЗАМЕТКУ!

Внутренне компилятор преобразует лямбда-выражение такого типа в закрытый метод и помещает в его тело код выражения.

Лямбда-выражение имеет следующую форму:

(параметры) => выражение-или-блок-операторов

Для удобства круглые скобки можно опускать, но только в ситуации, когда есть в точности один параметр выводимого типа.

В данном примере единственным параметром является `x`, а выражением — `x * x`:

```
x => x * x;
```

Каждый параметр лямбда-выражения соответствует параметру делегата, а тип выражения (которым может быть `void`) — возвращаемому типу делегата.

В нашем примере `x` соответствует параметру `i`, а выражение `x * x` — возвращаемому типу `int` и, следовательно, оно совместимо с делегатом `Transformer`.

Код лямбда-выражения может быть *блоком операторов*, а не просто выражением. Мы можем переписать пример следующим образом:

```
x => { return x * x; };
```

Лямбда-выражения чаще всего используются с делегатами `Func` и `Action`, поэтому приведенное ранее выражение вы будете нередко видеть в такой форме:

```
Func<int,int> sqr = x => x * x;
```

Компилятор обычно способен *выводить* тип лямбда-выражения из контекста. В противном случае типы параметров можно указывать явно:

```
Func<int,int> sqr = (int x) => x * x;
```

Ниже приведен пример выражения, принимающего два параметра:

```
Func<string,string,int> totalLength =
  (s1, s2) => s1.Length + s2.Length;
int total = totalLength ("добро", "пожаловать"); //total=15
```

Предполагая, что `Clicked` — это событие типа `EventHandler`, следующий код присоединяет обработчик события через лямбда-выражение:

```
obj.Clicked += (sender,args) => Console.Write ("Щелчок");
```

Захватывание внешних переменных

Лямбда-выражение может ссылаться на локальные переменные и параметры метода, в котором оно определено (*внешние переменные*). Например:

```
static void Main()
{
  int factor = 2;
  Func<int, int> multiplier = n => n * factor;
  Console.WriteLine (multiplier (3));  // 6
}
```

Внешние переменные, на которые ссылается лямбда-выражение, называются *захваченными переменными*. Лямбда-выражение, которое захватывает переменные, называется *замыканием*.

Захваченные переменные оцениваются, когда делегат фактически *вызывается*, а не когда эти переменные были *захвачены*:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3)); // 30
```

Лямбда-выражения сами могут обновлять захваченные переменные:

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine (natural());      // 0
Console.WriteLine (natural());      // 1
Console.WriteLine (seed);           // 2
```

Захваченные переменные имеют свое время жизни, расширенное до времени жизни делегата. В следующем примере локальная переменная `seed` обычно покидала бы область видимости после того, как выполнение `Natural` завершено. Но поскольку переменная `seed` была *захвачена*, время жизни этой переменной расширяется до времени жизни захватившего ее делегата, т.е. `natural`:

```
static Func<int> Natural()
{
  int seed = 0;
  return () => seed++;  // Возвращает замыкание
}
static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());  // 0
  Console.WriteLine (natural());  // 1
}
```

НА ЗАМЕТКУ!

Переменные также могут быть захвачены анонимными и локальными методами. В таких случаях правила для захваченных переменных остаются теми же самыми.

Захватывание итерационных переменных

Когда захватывается итерационная переменная в цикле `for`, она трактуется так, как если бы она была объявлена *вне* цикла. Это значит, что в каждой итерации захватывается *та же самая* переменная. Приведенный ниже код выводит 333, а не 012:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
  actions [i] = () => Console.Write (i);
foreach (Action a in actions) a();  // 333
```

Каждое замыкание (выделенное полужирным) захватывает одну и ту же переменную `i`. (Утверждение действительно имеет смысл, когда принять во внимание, что `i` является переменной, значение которой сохраняется между итерациями цикла; при желании `i` можно даже явно изменять внутри тела цикла.) Последствие заключается в том, что при более позднем вызове каждый делегат видит значение `i` на момент *вызова*, т.е. 3. Если взамен требуется вывести на экран 012, то решение предусматривает присваивание итерационной переменной какой-то локальной переменной с областью видимости *внутри* цикла:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions [i] = () => Console.Write (loopScopedi);
}
foreach (Action a in actions) a();  // 012
```

В результате замыкание на каждой итерации будет захватывать *другую* переменную.

НА ЗАМЕТКУ!

Циклы `foreach` работали аналогичным образом, но правила изменились. Начиная с версии C# 5.0, можно безопасно замыкать итерационную переменную цикла `foreach` без необходимости в наличии временной переменной.

Сравнение лямбда-выражений и локальных методов

Функциональность локальных методов (см. раздел "Локальные методы" на стр. 78) частично совпадает с функциональностью лямбда-выражений. Преимущества локальных методов в том, что они допускают рекурсию и избавляют от беспорядка, связанного с указанием делегатов. Устранение косвенности, присущей делегатам, также делает их несколько более эффективными, и они могут получать доступ к локальным переменным содержащего метода без "переноса" компилятором захваченных переменных внутрь скрытого класса.

Тем не менее, во многих случаях делегат все-таки *нужен*, чаще всего при вызове функции более высокого порядка (т.е. метода с параметром, имеющим тип делегата):

```
public void Foo (Func<int,bool> predicate) { ... }
```

В сценариях подобного рода в любом случае необходим делегат, и они представляют собой в точности те ситуации, когда лямбда-выражения обычно короче и яснее.

Анонимные методы

Анонимные методы — это функциональная возможность, появившаяся в версии C# 2.0, которая по большей части относится к лямбда-выражениям. Анонимный метод похож на лямбда-выражение за исключением того, что он лишен неявно типизированных параметров, синтаксиса выражений (анонимный метод должен всегда быть блоком операторов) и способности компилироваться в дерево выражения. Чтобы написать анонимный метод, понадобится указать ключевое слово `delegate`, затем (необязательное) объявление параметра и, наконец, тело метода. Например, имея показанный ниже делегат:

```
delegate int Transformer (int i);
```

вот как написать и вызвать анонимный метод:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3));  // 9
```

Первая строка семантически эквивалентна следующему лямбда-выражению:

```
Transformer sqr =       (int x) => {return x * x;};
```

Или просто:

```
Transformer sqr =             x => x * x;
```

Уникальная особенность анонимных методов состоит в том, что можно полностью опускать объявление параметра, даже если делегат его ожидает. Поступать так удобно при объявлении событий со стандартным пустым обработчиком:

```
public event EventHandler Clicked = delegate { };
```

В итоге устраняется необходимость проверки на равенство `null` перед запуском события. Приведенный далее код также будет допустимым (обратите внимание на отсутствие параметров):

```
Clicked += delegate { Console.Write ("щелчок"); };
```

Анонимные методы захватывают внешние переменные тем же самым способом, что и лямбда-выражения.

Операторы `try` и исключения

Оператор `try` указывает блок кода, предназначенный для обработки ошибок или очистки. За *блоком* `try` должен следовать один или несколько *блоков* `catch` и/или *блок* `finally`. Блок `catch` выполняется, когда возникает ошибка в блоке `try`. Блок `finally` выполняется после того, как поток управления покидает блок `try` (или блок `catch`, если он присутствует), обеспечивая очистку независимо от того, было сгенерировано исключение или нет.

Блок `catch` имеет доступ к объекту `Exception`, который содержит информацию об ошибке. Блок `catch` применяется либо для корректировки ошибки, либо для *повторной генерации* исключения. Исключение генерируется повторно, если нужно просто зарегистрировать факт возникновения проблемы в журнале или если необходимо сгенерировать исключение нового типа более высокого уровня.

Блок `finally` добавляет детерминизма к программе за счет того, что выполняется несмотря ни на что. Он полезен для проведения задач очистки вроде закрытия сетевых подключений.

Оператор try выглядит следующим образом:

```
try
{
  ... // Во время выполнения этого блока может
      //   возникнуть исключение
}
catch (ExceptionA ex)
{
  ... // Обработать исключение типа ExceptionA
}
catch (ExceptionB ex)
{
  ... // Обработать исключение типа ExceptionB
}
finally
{
  ... // Код очистки
}
```

Взгляните на показанный ниже код:

```
int x = 3, y = 0;
Console.WriteLine (x / y);
```

Поскольку y имеет нулевое значение, исполняющая среда генерирует исключение DivideByZeroException и программа прекращает работу. Чтобы предотвратить такое поведение, мы перехватываем исключение:

```
try
{
  int x = 3, y = 0;
  Console.WriteLine (x / y);
}
catch (DivideByZeroException ex)
{
  Console.Write ("y не может быть равно нулю. ");
}
// После исключения выполнение возобновляется
//   с этого места...
```

НА ЗАМЕТКУ!

Целью приведенного простого примера была иллюстрация обработки исключений. На практике лучше явно проверять делитель на равенство нулю перед вычислением.

Обработка исключений является относительно затратной, занимая сотни тактов процессора.

Когда исключение сгенерировано внутри оператора `try`, среда CLR выполняет следующую проверку.

Имеет ли оператор `try` совместимые с исключением блоки `catch`?

- Если имеет, тогда управление переходит на совместимый блок `catch`, затем на блок `finally` (при его наличии) и далее выполнение продолжается обычным образом.
- Если не имеет, тогда управление переходит прямо на блок `finally` (при его наличии), а среда CLR ищет в стеке вызовов другие блоки `try` и в случае их обнаружения повторяет проверку.

Если ни одна функция в стеке вызовов не взяла на себя ответственность за исключение, то пользователю отображается диалоговое окно с сообщением об ошибке и программа прекращает работу.

Конструкция `catch`

Конструкция `catch` указывает тип исключения, подлежащего перехвату. Типом может быть либо `System.Exception`, либо какой-то подкласс `System.Exception`. Перехват `System.Exception` обеспечивает отлавливание всех возможных ошибок, что удобно в перечисленных ниже ситуациях:

- программа потенциально может восстановиться независимо от конкретного типа исключения;
- планируется повторная генерация исключения (возможно, после его регистрации в журнале);
- обработчик ошибок является последним средством перед тем, как программа прекратит работу.

Однако более обычной является ситуация, когда перехватываются *исключения специфических типов*, чтобы не иметь дела с условиями, на которые обработчик не был рассчитан (например, `OutOfMemoryException`).

Перехватывать исключения нескольких типов можно с помощью множества конструкций `catch`:

```
try
{
  DoSomething();
}
catch (IndexOutOfRangeException ex) { ... }
catch (FormatException ex)          { ... }
catch (OverflowException ex)        { ... }
```

Для заданного исключения выполняется только одна конструкция `catch`. Если вы хотите предусмотреть сетку безопасности для перехвата общих исключений (вроде `System.Exception`), то должны размещать более специфические обработчики *первыми*.

Исключение можно перехватывать без указания переменной, если доступ к свойствам исключения не нужен:

```
catch (OverflowException)  // переменная не указана
{ ... }
```

Кроме того, можно опускать и переменную, и тип (тогда будут перехватываться все исключения):

```
catch { ... }
```

Фильтры исключений

Начиная с версии C# 6.0, в конструкции `catch` можно указывать *фильтр исключений* с помощью конструкции `when`:

```
catch (WebException ex)
  when (ex.Status == WebExceptionStatus.Timeout)
{
  ...
}
```

Если в приведенном примере генерируется исключение `WebException`, тогда вычисляется булевское выражение, находящееся после ключевого слова `when`. Если результатом оказывается `false`, то данный блок `catch` игнорируется и просматриваются любые последующие конструкции `catch`. Благодаря филь-

трам исключений может появиться смысл в повторном перехвате исключения того же самого типа:

```
catch (WebException ex) when (ex.Status ==
                                    некоторое_состояние)
{ ... }
catch (WebException ex) when (ex.Status ==
                                    другое_состояние)
{ ... }
```

Булевское выражение в конструкции `when` может иметь побочные эффекты, например, вызывать метод, который регистрирует в журнале сведения об исключении для целей диагностики.

Блок `finally`

Блок `finally` выполняется всегда — независимо от того, возникало ли исключение, и полностью ли был выполнен блок `try`. Блоки `finally` обычно используются для размещения кода очистки.

Блок `finally` выполняется в одном из следующих случаев:

- после завершения блока `catch`;
- после того, как поток управления покидает блок `try` из-за оператора перехода (например, `return` или `goto`);
- после окончания блока `try`.

Блок `finally` содействует повышению детерминизма программы. В приведенном далее примере открываемый файл *всегда* закрывается независимо от перечисленных ниже обстоятельств:

- блок `try` завершается нормально;
- происходит преждевременный возврат из-за того, что файл пуст (`EndOfStream`);
- во время чтения файла возникает исключение `IOException`.

Вот пример:

```
static void ReadFile()
{
  StreamReader reader = null;   // Из пространства
                                //   имен System.IO
  try
  {
```

```
    reader = File.OpenText ("file.txt");
    if (reader.EndOfStream) return;
    Console.WriteLine (reader.ReadToEnd());
  }
  finally
  {
    if (reader != null) reader.Dispose();
  }
}
```

В предложенном примере мы закрываем файл с помощью вызова `Dispose()` на `StreamReader`. Вызов `Dispose()` на объекте внутри блока `finally` представляет собой стандартное соглашение, соблюдаемое повсеместно в .NET, и оно явно поддерживается в языке C# через оператор `using`.

Оператор `using`

Многие классы инкапсулируют неуправляемые ресурсы, такие как файловые и графические дескрипторы или подключения к базам данных. Классы подобного рода реализуют интерфейс `System.IDisposable`, в котором определен единственный метод без параметров по имени `Dispose()`, предназначенный для очистки этих ресурсов. Оператор `using` предлагает элегантный синтаксис для вызова `Dispose()` на объекте реализации `IDisposable` внутри блока `finally`.

Оператор:

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
  ...
}
```

в точности эквивалентен следующему коду:

```
{
  StreamReader reader = File.OpenText ("file.txt");
  try
  {
    ...
  }
  finally
  {
    if (reader != null) ((IDisposable)reader).Dispose();
  }
}
```

Объявления using (C# 8)

Если опустить круглые скобки и блок операторов, следующий за оператором using, то мы получим *объявление using*. В результате ресурс освобождается, когда поток управления выходит за пределы *включающего* блока операторов:

```
if (File.Exists ("file.txt"))
{
  using var reader = File.OpenText ("file.txt");
  Console.WriteLine (reader.ReadLine());
  ...
}
```

В данном случае reader освободится после того, как поток управления выйдет за пределы тела оператора if.

Генерация исключений

Исключения могут генерироваться либо исполняющей средой, либо пользовательским кодом. В приведенном далее примере метод Display() генерирует исключение System.ArgumentNullException:

```
static void Display (string name)
{
  if (name == null)
    throw new ArgumentNullException (nameof (name));
  Console.WriteLine (name);
}
```

Выражения throw

Начиная с версии C# 7, конструкция throw может появляться как выражение в функциях, сжатых до выражения:

```
public string Foo() =>
  throw new NotImplementedException();
```

Выражение throw может также находиться внутри тернарной условной операции:

```
string ProperCase (string value) =>
  value == null ? throw new ArgumentException ("value") :
  value == "" ? "" :
  char.ToUpper (value[0]) + value.Substring (1);
```

Повторная генерация исключения

Исключение можно перехватывать и генерировать повторно:

```
try { ... }
catch (Exception ex)
{
  // Зарегистрировать в журнале информацию об ошибке
  ...
  throw; // Повторно сгенерировать то же самое исключение
}
```

Повторная генерация в подобной манере дает возможность зарегистрировать в журнале информацию об ошибке, не *подавляя* ее. Она также позволяет отказаться от обработки исключения, если обстоятельства сложились не так, как ожидалось.

НА ЗАМЕТКУ!

Если `throw` заменить `throw ex`, то пример сохранит работоспособность, но свойство `StackTrace` исключения больше не будет отражать исходную ошибку.

Еще один распространенный сценарий предусматривает повторную генерацию исключения более специфического или содержательного типа:

```
try
{
  ... // Произвести разбор даты рождения из элемента XML
}
catch (FormatException ex)
{
 throw new XmlException ("Некорректная дата рождения", ex);
}
```

При повторной генерации другого исключения в свойстве `InnerException` можно указать исходное исключение, чтобы помочь в отладке. Почти все типы исключений предоставляют конструктор для такой цели (как в рассмотренном примере).

Основные свойства System.Exception

Ниже описаны наиболее важные свойства класса System.Exception.

StackTrace

Строка, представляющая все методы, которые были вызваны, начиная с источника исключения и заканчивая блоком catch.

Message

Строка с описанием ошибки.

InnerException

Внутреннее исключение (если есть), которое привело к генерации внешнего исключения. Само это свойство может иметь отличающееся свойство InnerException.

Общие типы исключений

Перечисленные ниже типы исключений широко применяются в CLR и .NET. Их можно генерировать либо использовать в качестве базовых классов для порождения специальных типов исключений.

System.ArgumentException

Генерируется, когда функция вызывается с недопустимым аргументом. Как правило, указывает на наличие ошибки в программе.

System.ArgumentNullException

Подкласс ArgumentException, который генерируется, когда аргумент функции (непредвиденно) оказывается null.

System.ArgumentOutOfRangeException

Подкласс ArgumentException, который генерируется, когда (обычно числовой) аргумент имеет чересчур большое или слишком малое значение. Например, это исключение возникает при передаче отрицательного числа в функцию, которая принимает только положительные значения.

System.InvalidOperationException

Генерируется, когда состояние объекта оказывается неподходящим для успешного выполнения метода независимо от любых заданных значений аргументов. В качестве примеров можно назвать чтение неоткрытого файла или получение следующего элемента из перечислителя в случае, если лежащий в основе список был изменен на середине выполнения итерации.

`System.NotSupportedException`

Генерируется для указания на то, что специфическая функциональность не поддерживается. Хорошим примером может служить вызов метода `Add()` на коллекции, для которой `IsReadOnly` возвращает `true`.

`System.NotImplementedException`

Генерируется для указания на то, что функция пока еще не реализована.

`System.ObjectDisposedException`

Генерируется, когда объект, на котором вызывается функция, был освобожден.

Перечисление и итераторы

Перечисление

Перечислитель — это допускающий только чтение однонаправленный курсор по *последовательности значений*. Перечислитель представляет собой объект, который реализует либо интерфейс `System.Collections.IEnumerator`, либо интерфейс `System.Collections.Generic.IEnumerator<T>`.

Оператор `foreach` выполняет итерацию по *перечислимому* объекту. Перечислимый объект является логическим представлением последовательности. Это не собственно курсор, а объект, который производит курсор на себе самом. Перечислимый объект либо реализует интерфейс `IEnumerable`/`IEnumerable<T>`, либо имеет метод по имени `GetEnumerator()`, который возвращает *перечислитель*.

Шаблон перечисления выглядит следующим образом:

```
class Enumerator  // Обычно реализует IEnumerator<T>
{
  public IteratorVariableType Current { get {...} }
```

```
  public bool MoveNext() {...}
}
class Enumerable  // Обычно реализует IEnumerable<T>
{
  public Enumerator GetEnumerator() {...}
}
```

Ниже показан высокоуровневый способ выполнения итерации по символам в слове "вода" с использованием оператора `foreach`:

```
foreach (char c in "вода") Console.WriteLine (c);
```

А вот низкоуровневый метод проведения итерации по символам в слове "вода" без применения оператора `foreach`:

```
using (var enumerator = "вода".GetEnumerator())
  while (enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine (element);
  }
```

Если перечислитель реализует интерфейс `IDisposable`, то оператор `foreach` также действует как оператор `using`, неявно освобождая объект перечислителя.

Инициализаторы коллекций

Перечислимый объект можно создать и заполнить за один шаг. Например:

```
using System.Collections.Generic;
...
List<int> list = new List<int> {1, 2, 3};
```

Компилятор транслирует код следующим образом:

```
List<int> list = new List<int>();
list.Add (1); list.Add (2); list.Add (3);
```

Здесь требуется, чтобы перечислимый объект реализовывал интерфейс `System.Collections.IEnumerable` и таким образом имел метод `Add()`, который принимает подходящее количество параметров для вызова. Аналогичным способом можно инициализировать словари (типы, реализующие интерфейс `System.Collections.IDictionary`):

```
var dict = new Dictionary<int, string>()
{
  { 5, "пять"   },
  { 10, "десять" }
};
```

Или более кратко:

```
var dict = new Dictionary<int, string>()
{
  [5] = "пять",
  [10] = "десять"
};
```

Второй вариант записи допустим не только со словарями, но и с любым типом, для которого существует индексатор.

Итераторы

Наряду с тем, что оператор `foreach` можно рассматривать как *потребитель* перечислителя, итератор следует считать *поставщиком* перечислителя. В приведенном ниже примере итератор используется для возвращения последовательности чисел Фибоначчи (где каждое число является суммой двух предыдущих чисел):

```
using System;
using System.Collections.Generic;
class Test
{
  static void Main()
  {
    foreach (int fib in Fibs(6))
      Console.Write (fib + " ");
  }
  static IEnumerable<int> Fibs(int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1;
        i < fibCount;
        i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
}
```

ВЫВОД: 1 1 2 3 5 8

В то время как оператор `return` выражает: "Вот значение, которое должно быть возвращено из этого метода", оператор `yield return` сообщает: "Вот следующий элемент, который должен быть выдан этим перечислителем". При встрече каждого оператора `yield` управление возвращается вызывающему компоненту, но состояние вызываемого метода сохраняется, так что данный метод может продолжить свое выполнение, как только вызывающий компонент перечислит следующий элемент. Жизненный цикл состояния ограничен перечислителем, поэтому состояние может быть освобождено, когда вызывающий компонент завершит перечисление.

НА ЗАМЕТКУ!

Компилятор преобразует методы итератора в закрытые классы, которые реализуют интерфейсы `IEnumerable<T>` и/или `IEnumerator<T>`. Логика внутри блока итератора "инвертируется", после чего сращивается с методом `MoveNext()` и свойством `Current` класса перечислителя, сгенерированного компилятором. Это значит, что при вызове метода итератора всего лишь создается экземпляр сгенерированного компилятором класса; никакой написанный вами код на самом деле не выполняется! Ваш код запускается только когда начинается перечисление по результирующей последовательности, обычно с помощью оператора `foreach`.

Семантика итератора

Итератор представляет собой метод, свойство или индексатор, который содержит один или большее количество операторов `yield`. Итератор должен возвращать реализацию одного из следующих четырех интерфейсов (иначе компилятор сгенерирует сообщение об ошибке):

```
System.Collections.IEnumerable
System.Collections.IEnumerator
System.Collections.Generic.IEnumerable<T>
System.Collections.Generic.IEnumerator<T>
```

Итераторы, которые возвращают реализацию интерфейса IEnumerator (IEnumerator<T>), как правило, применяются менее часто. Они удобны при написании специального класса коллекции: обычно вы назначаете итератору имя GetEnumerator() и обеспечиваете реализацию классом интерфейса IEnumerable<T>.

Итераторы, возвращающие реализацию интерфейса IEnumerable (IEnumerable<T>), являются более распространенными — и они проще в использовании, т.к. вам не приходится разрабатывать класс коллекции. "За кулисами" компилятор генерирует закрытый класс, реализующий IEnumerable<T> (а также IEnumerator<T>).

Множество операторов yield

Итератор может включать несколько операторов yield:

```
static void Main()
{
  foreach (string s in Foo())
    Console.Write (s + " ");  // Один Два Три
}
static IEnumerable<string> Foo()
{
  yield return "Один";
  yield return "Два";
  yield return "Три";
}
```

Оператор yield break

Наличие оператора return в блоке итератора не допускается; взамен вы обязаны использовать оператор yield break, для указания на то, что блок итератора должен быть завершен преждевременно, не возвращая больше элементов. Чтобы продемонстрировать его работу, можно модифицировать метод Foo(), как показано ниже:

```
static IEnumerable<string> Foo (bool breakEarly)
{
  yield return "Один";
  yield return "Два";
  if (breakEarly) yield break;
  yield return "Три";
}
```

Компоновка последовательностей

Итераторы в высшей степени компонуемы. Мы можем расширить наш пример с числами Фибоначчи, добавив к классу следующий метод:

```
static IEnumerable<int> EvenNumbersOnly
  (IEnumerable<int> sequence)
{
  foreach (int x in sequence)
    if ((x % 2) == 0)
      yield return x;
}
```

После этого можно выводить четные числа Фибоначчи:

```
foreach (int fib in EvenNumbersOnly (Fibs (6)))
  Console.Write (fib + " ");  // 2 8
```

Каждый элемент не вычисляется вплоть до последнего момента — когда он запрашивается операцией `MoveNext()`. На рис. 5 показаны запросы данных и их вывод с течением времени.

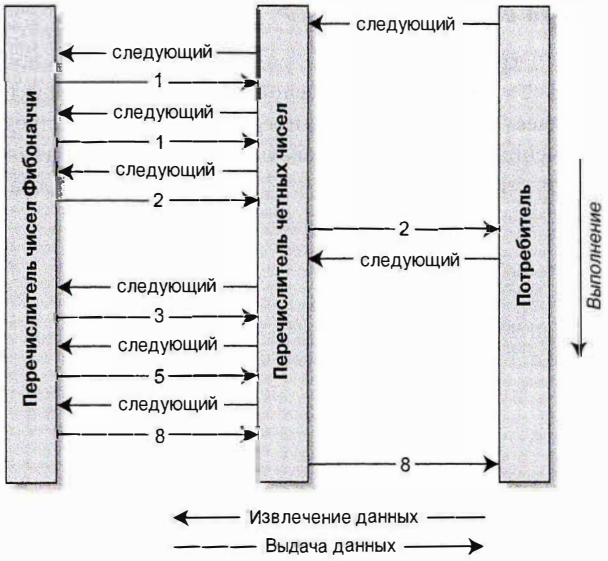


Рис. 5. Пример компоновки последовательностей

Возможность компоновки, поддерживаемая шаблоном итератора, жизненно необходима при построении запросов LINQ.

Типы (значений), допускающие null

Ссылочные типы могут представлять несуществующее значение с помощью ссылки null. Однако типы значений не способны представлять значения null обычным образом. Например:

```
string s = null;       // Нормально, ссылочный тип
int i = null;          // Ошибка на этапе компиляции,
                       //   тип int не может быть null
```

Чтобы представить null с помощью типа значения, необходимо использовать специальную конструкцию, которая называется *типом, допускающим значение null*. Тип, допускающий значение null, обозначается как тип значения, за которым следует символ ?:

```
int? i = null;         // Нормально; тип, допускающий
                       //   значение null
Console.WriteLine (i == null);  // True
```

Структура Nullable<T>

Тип T? транслируется в System.Nullable<T>. Тип Nullable<T> является легковесной неизменяемой структурой, которая имеет только два поля, предназначенные для представления значения (Value) и признака наличия значения (HasValue). В сущности, структура System.Nullable<T> очень проста:

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault();
  public T GetValueOrDefault (T defaultValue);
  ...
}
```

Код:

```
int? i = null;
Console.WriteLine (i == null);         // True
```

транслируется в:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);      // True
```

Попытка извлечь значение `Value`, когда `HasValue` равно `false`, приводит к генерации исключения `InvalidOperationException`. Метод `GetValueOrDefault()` возвращает значение `Value`, если `HasValue` равно `true`, и результат `new T()` или заданное стандартное значение в противном случае.

Стандартным значением `T?` является `null`.

Преобразования типов, допускающих значение `null`

Преобразование из `T` в `T?` является неявным, а из `T?` в `T` — явным. Например:

```
int? x = 5;          // неявное
int y = (int)x;      // явное
```

Явное приведение полностью эквивалентно обращению к свойству `Value` объекта типа, допускающего `null`. Следовательно, если `HasValue` равно `false`, тогда генерируется исключение `InvalidOperationException`.

Упаковка и распаковка значений типов, допускающих `null`

Когда `T?` упаковывается, упакованное значение в куче содержит `T`, а не `T?`. Такая оптимизация возможна из-за того, что упакованное значение относится к ссылочному типу, который уже способен выражать `null`.

В C# также разрешено распаковывать типы, допускающие `null`, с помощью операции `as`. Если приведение не удается, то результатом будет `null`:

```
object o = "строка";
int? x = o as int?;
Console.WriteLine (x.HasValue);  // False
```

Подъем операций

В структуре `Nullable<T>` не определены такие операции, как `<`, `>` или даже `==`. Несмотря на это, следующий код успешно компилируется и выполняется:

```
int? x = 5;
int? y = 10;
bool b = x < y;  // true
```

Код работает благодаря тому, что компилятор заимствует, или "поднимает", операцию "меньше чем" у лежащего в основе типа значения. Семантически предыдущее выражение сравнения транслируется так:

```
bool b = (x.HasValue && y.HasValue)
         ? (x.Value < y.Value)
         : false;
```

Другими словами, если x и y имеют значения, то сравнение производится посредством операции "меньше чем" типа int; в противном случае результатом будет false.

Подъем операций означает возможность неявного использования операций из T с типом T?. Вы можете определить операции для T?, чтобы предоставить специализированное поведение в отношении null, но в подавляющем большинстве случаев лучше полагаться на автоматическое применение компилятором систематической логики работы со значением null.

В зависимости от категории операции компилятор представляет логику в отношении null по-разному.

Операции эквивалентности (== и !=)

Поднятые операции эквивалентности обрабатывают значения null точно так же, как поступают ссылочные типы. Это означает, что два значения null равны:

```
Console.WriteLine (       null ==       null); //True
Console.WriteLine ((bool?)null == (bool?)null); //True
```

Более того:

- если в точности один операнд имеет значение null, тогда операнды не равны;
- если оба операнда отличаются от null, то сравниваются их свойства Value.

Операции отношения (<, <=, >=, >)

Работа операций отношения основана на принципе, согласно которому сравнение операндов null не имеет смысла. Это означает, что сравнение null либо с null, либо со значением, отличающимся от null, дает в результате false.

```
bool b = x < y;  // Транслируется в:
bool b = (x == null || y == null)
  ? false
  : (x.Value < y.Value);
// b равно false (предполагая, что x равно 5, а y - null)
```

Остальные операции (+, –, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

Остальные операции возвращают `null`, когда любой из операндов равен `null`. Такой шаблон должен быть хорошо знакомым пользователям SQL.

```
int? c = x + y;  // Транслируется в:
int? c = (x == null || y == null)
  ? null
  : (int?) (x.Value + y.Value);
// c равно null (предполагая, что x равно 5, а y - null)
```

Исключением является ситуация, когда операции `&` и `|` применяются к `bool?`, что мы вскоре обсудим.

Смешивание в операциях типов, допускающих и не допускающих `null`

Типы, допускающие и не допускающие `null`, можно смешивать (это работает, поскольку существует неявное преобразование из `T` в `T?`):

```
int? a = null;
int b = 2;
int? c = a + b; //c равно null - эквивалентно a + (int?)b
```

Тип `bool?` и операции `&` и `|`

Когда предоставленные операнды имеют тип `bool?`, операции `&` и `|` трактуют `null` как *неизвестное значение*. Таким образом, `null | true` дает `true` по следующим причинам:

- если неизвестное значение равно `false`, то результатом будет `true`;
- если неизвестное значение равно `true`, то результатом будет `true`.

Аналогичным образом `null & false` дает `false`. Подобное поведение должно быть знакомым пользователям SQL. Ниже приведены другие комбинации:

```
bool? n = null, f = false, t = true;
Console.WriteLine (n | n);  // (null)
Console.WriteLine (n | f);  // (null)
Console.WriteLine (n | t);  // True
Console.WriteLine (n & n);  // (null)
Console.WriteLine (n & f);  // False
Console.WriteLine (n & t);  // (null)
```

Типы, допускающие `null`, и операции для работы со значениями `null`

Типы, допускающие значение `null`, особенно хорошо работают с операцией `??` (см. раздел "Операция объединения с `null`" на стр. 60).

Например:

```
int? x = null;
int y = x ?? 5;  // y равно 5

int? a = null, b = null, c = 123;
Console.WriteLine (a ?? b ?? c);  // 123
```

Использование операции `??` эквивалентно вызову `GetValueOrDefault()` с явным стандартным значением за исключением того, что выражение для стандартного значения никогда не оценивается, если переменная не равна `null`.

Типы, допускающие значение `null`, также удобно применять с `null`-условной операцией (см. раздел "`null`-условная операция" на стр. 60). В следующем примере переменная `length` получает значение `null`:

```
System.Text.StringBuilder sb = null;
int? length = sb?.ToString().Length;
```

Скомбинировав этот код и операцию объединения с `null`, переменной `length` можно присвоить значение `0` вместо `null`:

```
int length = sb?.ToString().Length ?? 0;
```

Ссылочные типы, допускающие значение null (C# 8)

В то время как *типы, допускающие null*, привносят поддержку null в типы значений, *ссылочные типы, допускающие значение null*, делают противоположное и обеспечивают (в некоторой степени) *поддержку не null* ссылочными типами для того, чтобы помочь избегать исключений NullReferenceException.

Ссылочные типы, допускающие null, вводят уровень безопасности, который приводится в действие самим компилятором в форме предупреждений, когда он обнаруживает код, подвергающийся риску генерации исключения NullReferenceException.

Чтобы запустить в работу ссылочные типы, допускающие null, потребуется или добавить элемент Nullable в файл проекта .csproj (если они нужны для всего проекта):

```
<Nullable>enable</Nullable>
```

или/и использовать в тех местах кода, где такие типы должны возыметь действие, следующие директивы:

```
#nullable enable   // Включает ссылочные типы,
                   // допускающие null, начиная с этой точки
#nullable disable  // Отключает ссылочные типы,
                   // допускающие null, начиная с этой точки
#nullable restore  // Приводит ссылочные типы,
                   // допускающие null, в состояние,
                   // соответствующее настройке проекта
```

После активизации ссылочных типов, допускающих null, компилятор делает поддержку не null принятой по умолчанию: если необходимо, чтобы ссылочный тип получал значения null, тогда придется применить к нему суффикс ? для указания *ссылочного типа, допускающего null*. В показанном ниже примере s1 не допускает null, тогда как s2 допускает:

```
#nullable enable        // Включает ссылочные типы,
                        //   допускающие null

string s1 = null;       // Приводит к выдаче компилятором
                        //   предупреждения!
string? s2 = null;      // Нормально: s2 имеет ссылочный
                        //   тип, допускающий null
```

НА ЗАМЕТКУ!

Поскольку ссылочные типы, допускающие `null`, являются конструкциями этапа компиляции, во время выполнения между `string` и `string?` нет никаких отличий. И напротив, типы значений, допускающие `null`, привносят в систему типов кое-что конкретное, а именно — структуру `Nullable<T>`.

Для приведенного далее кода компилятор также выдаст предупреждение из-за отсутствия инициализации x:

```
class Foo { string x; }
```

Предупреждение исчезнет, если инициализировать x либо через инициализатор поля, либо посредством кода в конструкторе.

Компилятор предупреждает и в случае разыменования ссылочного типа, допускающего `null`, если предположительно может возникнуть исключение `NullReferenceException`. В следующем примере доступ к свойству `Length` строки приводит к выдаче предупреждения:

```
void Foo (string? s) => Console.Write (s.Length);
```

Чтобы убрать предупреждение, можно использовать *null-терпимую* (null-forgiving) *операцию* (`!`):

```
void Foo (string? s) => Console.Write (s!.Length);
```

Подобное применение `null`-терпимой операции опасно тем, что мы могли бы в итоге получить то же самое исключение `NullReferenceException`, которого в первую очередь пытались избежать. Вот как можно было бы исправить ситуацию:

```
void Foo (string? s)
{
  if (s != null) Console.Write (s.Length);
}
```

Обратите внимание, что теперь `null`-терпимая операция не нужна. Причина в том, что компилятор проводит статический анализ и достаточно интеллектуален для того, чтобы сделать вывод (по крайней мере, в простых случаях) о том, когда разыменование безопасно и потому исключение `NullReferenceException` не возникнет.

Способность компилятора к обнаружению и предупреждению отнюдь не безупречна, к тому же существуют пределы того, что она может охватить. Скажем, компилятор не имеет возможности узнать, был ли заполнен массив элементами, а потому показанный ниже код не приводит к выдаче предупреждения:

```
var strings = new string[10];
Console.WriteLine (strings[0].Length);
```

Расширяющие методы

Расширяющие методы позволяют расширять существующий тип новыми методами, не изменяя определение исходного типа. Расширяющий метод — это статический метод статического класса, в котором к первому параметру применен модификатор `this`. Типом первого параметра должен быть тип, подвергающийся расширению. Например:

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty (s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

Расширяющий метод `IsCapitalized()` может вызываться так, как если бы он был методом экземпляра класса `string`:

```
Console.Write ("Перт".IsCapitalized());
```

Вызов расширяющего метода при компиляции транслируется в обычный вызов статического метода:

```
Console.Write (StringHelper.IsCapitalized ("Перт"));
```

Интерфейсы также можно расширять:

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;
  throw new InvalidOperationException ("Элементы
                                        отсутствуют!");
}
...
Console.WriteLine ("Сиэтл".First()); // С
```

Цепочки расширяющих методов

Как и методы экземпляра, расширяющие методы предлагают аккуратный способ для связывания функций в цепочки. Взгляните на следующие две функции:

```
public static class StringHelper
{
  public static string Pluralize (this string s)  {...}
  public static string Capitalize (this string s) {...}
}
```

Строковые переменные x и y эквивалентны и получают значение "Строка", но x использует расширяющие методы, тогда как y — статические:

```
string x = "строка".Pluralize().Capitalize();
string y = StringHelper.Capitalize
  (StringHelper.Pluralize ("строка"));
```

Неоднозначность и распознавание

Пространства имен

Расширяющий метод не может быть доступен до тех пор, пока его пространство имен не окажется в области видимости (обычно за счет импорта посредством оператора `using`).

Расширяющий метод или метод экземпляра

Любой совместимый метод экземпляра всегда будет иметь преимущество над расширяющим методом — даже когда параметры расширяющего метода дают более точное соответствие по типам.

Расширяющий метод или другой расширяющий метод

Если два расширяющих метода имеют одинаковые сигнатуры, то расширяющий метод должен вызываться как обычный статический метод, чтобы устранить неоднозначность при вызове. Однако если один расширяющий метод имеет более специфичные аргументы, тогда предпочтение будет отдано ему.

Анонимные типы

Анонимный тип — это простой класс, созданный на лету с целью хранения набора значений. Для создания анонимного типа применяется ключевое слово `new` с инициализатором объекта, указывающим свойства и значения, которые будет содержать тип. Например:

```
var dude = new { Name = "Боб", Age = 1 };
```

Компилятор преобразует данное объявление в закрытый вложенный тип со свойствами, допускающими только чтение, для `Name` (типа `string`) и `Age` (типа `int`). При ссылке на анонимный тип должно использоваться ключевое слово `var`, т.к. имя этого типа генерируется компилятором.

Имя свойства анонимного типа может быть выведено из выражения, которое само по себе является идентификатором. Таким образом, приведенный далее код:

```
int Age = 1;
var dude = new { Name = "Боб", Age };
```

эквивалентен следующему коду:

```
var dude = new { Name = "Боб", Age = Age };
```

Можно создавать массивы анонимных типов, как показано ниже:

```
var dudes = new[]
{
  new { Name = "Боб", Age = 30 },
  new { Name = "Мэри", Age = 40 }
};
```

Анонимные типы применяются главным образом при написании запросов LINQ.

Кортежи

Подобно анонимным типам кортежи (C# 7+) предлагают простой способ хранения набора значений. Главная цель кортежей — безопасно возвращать множество значений из метода, не прибегая к параметрам `out` (то, что невозможно делать с помощью анонимных типов). Создать *литеральный кортеж* проще

всего, указав в круглых скобках список желаемых значений. В результате создается кортеж с *неименованными* элементами:

```
var bob = ("Боб", 23);
Console.WriteLine (bob.Item1);  // Боб
Console.WriteLine (bob.Item2);  // 23
```

В отличие от анонимных типов применять ключевое слово `var` необязательно и *тип кортежа* можно указывать явно:

```
(string,int) bob = ("Боб", 23);
```

Это означает, что кортеж можно успешно возвращать из метода:

```
static (string,int) GetPerson() => ("Боб", 23);

static void Main()
{
  (string,int) person = GetPerson();
  Console.WriteLine (person.Item1);  // Боб
  Console.WriteLine (person.Item2);  // 23
}
```

Кортежи хорошо сочетаются с обобщениями, так что все следующие типы законны:

```
Task<(string,int)>
Dictionary<(string,int),Uri>
IEnumerable<(int ID, string Name)>  // См. ниже...
```

Кортежи являются *типами значений* с *изменяемыми* (допускающими чтение/запись) элементами. Таким образом, после создания кортежа можно модифицировать `Item1`, `Item2` и т.д.

Именование элементов кортежа

При создании литеральных кортежей элементам можно дополнительно назначать содержательные имена:

```
var tuple = (Name:"Боб", Age:23);
Console.WriteLine (tuple.Name);  // Боб
Console.WriteLine (tuple.Age);   // 23
```

То же самое разрешено делать при указании *типов кортежей*:

```
static (string Name, int Age) GetPerson() => ("Боб",23);
```

Имена элементов *выводятся* автоматически из имен свойств или полей:

```
var now = DateTime.Now;
var tuple = (now.Day, now.Month, now.Year);
Console.WriteLine (tuple.Day);  // Нормально
```

НА ЗАМЕТКУ!

Кортежи представляют собой "синтаксический сахар" для использования семейства обобщенных структур `ValueTuple<T1>` и `ValueTuple<T1,T2>`, которые имеют поля с именами `Item1`, `Item2` и т.д. Следовательно, `(string,int)` является псевдонимом для `ValueTuple<string,int>`. Это означает, что "именованные элементы" существуют только в исходном коде, а также в "воображении" компилятора, и во время выполнения обычно исчезают.

Деконструирование кортежей

Кортежи неявно поддерживают шаблон деконструирования (см. раздел "Деконструкторы" на стр. 80), так что кортеж легко *деконструировать* в отдельные переменные. Таким образом, взамен:

```
var bob = ("Боб", 23);
string name = bob.Item1;
int age = bob.Item2;
```

можно написать:

```
var bob = ("Боб", 23);
(string name, int age) = bob; // Деконструировать bob
                              //   в name и age.
Console.WriteLine (name);
Console.WriteLine (age);
```

Синтаксис деконструирования похож на синтаксис объявления кортежа с именованными элементами, что может привести к путанице. Следующий код подчеркивает разницу:

```
(string name, int age) = bob;      // Деконструирование
(string name, int age) bob2 = bob; // Объявление кортежа
```

LINQ

Язык интегрированных запросов (Language Integrated Query — LINQ) дает возможность писать структурированные безопасные в отношении типов запросы к локальным коллекциям объектов и удаленным источникам данных.

Язык LINQ позволяет отправлять запросы любой коллекции, реализующей интерфейс `IEnumerable<T>`, будь то массив, список, DOM-модель XML или удаленный источник данных (такой как таблица в базе данных SQL Server). Язык LINQ обеспечивает преимущества как проверки типов на этапе компиляции, так и составления динамических запросов.

НА ЗАМЕТКУ!

Экспериментировать с LINQ удобно, загрузив LINQPad (`https://www.linqpad.net/`). Инструмент LINQPad позволяет интерактивно запрашивать локальные коллекции и базы данных SQL с помощью LINQ без какой-либо настройки и сопровождается многочисленными примерами.

Основы LINQ

Базовыми единицами данных в LINQ являются *последовательности* и *элементы*. Последовательность представляет собой любой объект, который реализует обобщенный интерфейс `IEnumerable`, а элемент является членом в этой последовательности. В следующем примере `names` будет последовательностью, а `"Том"`, `"Кирк"` и `"Гарри"` — элементами:

```
string[] names = { "Том", "Кирк", "Гарри" };
```

Последовательность такого рода называется *локальной последовательностью*, потому что она представляет локальную коллекцию объектов в памяти.

Операция запроса — это метод, который трансформирует последовательность. Типичная операция запроса принимает *входную последовательность* и выпускает трансформированную *выходную последовательность*. В классе `Enumerable` из пространства имен `System.Linq` имеется около 40 операций запросов,

которые реализованы в виде статических методов. Их называют *стандартными операциями запросов.*

НА ЗАМЕТКУ!

Язык LINQ также поддерживает последовательности, которые могут динамически наполняться из удаленного источника данных, подобного SQL Server. Такие последовательности дополнительно реализуют интерфейс `IQueryable<T>` и поддерживаются через соответствующий набор стандартных операций запросов в классе `Queryable`.

Простой запрос

Запрос — это выражение, которое трансформирует последовательности с помощью одной или большего количества операций запросов. Простейший запрос состоит из одной входной последовательности и одной операции. Например, мы можем применить операцию `Where()` к простому массиву для извлечения элементов с длиной минимум четыре символа:

```
string[] names = { "Том", "Кирк", "Гарри" };
IEnumerable<string> filteredNames =
  System.Linq.Enumerable.Where (
    names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.Write (n + "|");  // Кирк|Гарри|
```

Поскольку стандартные операции запросов реализованы в виде расширяющих методов, мы можем вызывать операцию `Where()` прямо на `names` — как если бы она была методом экземпляра:

```
IEnumerable<string> filteredNames =
  names.Where (n => n.Length >= 4);
```

(Чтобы такой код скомпилировался, потребуется импортировать пространство имен `System.Linq` с помощью директивы `using`.) Метод `Where()` в классе `System.Linq.Enumerable` имеет следующую сигнатуру:

```
static IEnumerable<TSource> Where<TSource> (
  this IEnumerable<TSource> source,
  Func<TSource,bool> predicate)
```

В `source` указывается *входная последовательность*, а в `predicate` — делегат, который вызывается для каждого входного *элемента*. Метод `Where()` помещает в *выходную последовательность* все элементы, для которых делегат `predicate` возвращает `true`. Внутренне он реализован посредством итератора — ниже показан исходный код:

```
foreach (TSource element in source)
  if (predicate (element))
    yield return element;
```

Проецирование

Еще одной фундаментальной операцией запроса является метод `Select()`, который трансформирует (*проецирует*) каждый элемент во входной последовательности с помощью заданного лямбда-выражения:

```
string[] names = { "Том", "Кирк", "Гарри" };
IEnumerable<string> upperNames =
  names.Select (n => n.ToUpper());
foreach (string n in upperNames)
  Console.Write (n + "|");  // ТОМ|КИРК|ГАРРИ|
```

Запрос может выполнять проецирование в анонимный тип:

```
var query = names.Select (n => new {
                                Name = n,
                                Length = n.Length
                              });
foreach (var row in query)
  Console.WriteLine (row);
```

Вот результат:

```
{ Name = Том, Length = 3 }
{ Name = Кирк, Length = 4 }
{ Name = Гарри, Length = 5 }
```

Take() и Skip()

В LINQ важен первоначальный порядок следования элементов внутри входной последовательности. На это поведение полагаются некоторые операции запросов, такие как `Take()`, `Skip()` и

Reverse(). Операция Take() выдает первые *x* элементов, отбрасывая остальные:

```
int[] numbers = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3);
// firstThree содержит { 10, 9, 8 }
```

Операция Skip() пропускает первые *x* элементов и выдает остальные:

```
IEnumerable<int> lastTwo = numbers.Skip (3);
```

Операции над элементами

Не все операции запросов возвращают последовательность. Операции над *элементами* извлекают из входной последовательности один элемент; примерами таких операций служат First(), Last(), Single() и ElementAt():

```
int[] numbers = { 10, 9, 8, 7, 6 };
int firstNumber = numbers.First();                    // 10
int lastNumber = numbers.Last();                       // 6
int secondNumber = numbers.ElementAt (2);              // 8
int firstOddNum = numbers.First (n => n%2 == 1);       // 9
```

Все указанные выше операции генерируют исключение, если элементы отсутствуют. Чтобы избежать исключения, необходимо использовать FirstOrDefault(), LastOrDefault(), SingleOrDefault() или ElementAtOrDefault() — когда ни одного элемента не найдено, они возвращают null (или значение default для типов значений).

Методы Single() и SingleOrDefault() эквивалентны методам First() и FirstOrDefault(), но генерируют исключение при наличии более одного совпадения. Такое поведение удобно во время запрашивания строки по первичному ключу из таблицы базы данных.

Операции агрегирования

Операции *агрегирования* возвращают скалярное значение, обычно принадлежащее числовому типу. Наиболее распространенными операциями агрегирования являются Count(), Min(), Max() и Average():

```
int[] numbers = { 10, 9, 8, 7, 6 };
int count = numbers.Count();     // 5
```

```
int min = numbers.Min();              // 6
int max = numbers.Max();              // 10
double avg = numbers.Average();       // 8
```

Операция `Count()` принимает необязательный предикат, который указывает, должен ли включаться указанный элемент. Следующий код подсчитывает четные числа:

```
int evenNums = numbers.Count (n => n % 2 == 0);   // 3
```

Операции `Min()`, `Max()` и `Average()` принимают необязательный аргумент, который трансформирует каждый элемент до того, как он будет подвергнут агрегированию:

```
int maxRemainderAfterDivBy5 = numbers.Max
                                (n => n % 5);       // 4
```

Приведенный ниже код вычисляет среднеквадратическое значение последовательности `numbers`:

```
double rms = Math.Sqrt (numbers.Average (n => n * n));
```

Квалификаторы

Квалификаторы возвращают значение типа `bool`. Квалификаторами являются операции `Contains()`, `Any()` и `All()`, а также операция `SequenceEquals()`, которая сравнивает две последовательности:

```
int[] numbers = { 10, 9, 8, 7, 6 };
bool hasTheNumberNine = numbers.Contains (9);  // true
bool hasMoreThanZeroElements = numbers.Any();  // true
bool hasOddNum = numbers.Any (n => n % 2 == 1); // true
bool allOddNums = numbers.All (n => n % 2 == 1); // false
```

Операции над множествами

Операции над *множествами* принимают две входных последовательности одного и того же типа. Операция `Concat()` добавляет одну последовательность в конец другой; операция `Union()` делает то же самое, но с удалением дубликатов:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
IEnumerable<int>
  concat = seq1.Concat (seq2),  // { 1, 2, 3, 3, 4, 5 }
  union = seq1.Union   (seq2),  // { 1, 2, 3, 4, 5 }
```

В данной категории есть еще две операции — `Intersect()` и `Except()`:

```
IEnumerable<int>
  commonality = seq1.Intersect (seq2),  // { 3 }
  difference1 = seq1.Except    (seq2),  // { 1, 2 }
  difference2 = seq2.Except    (seq1);  // { 4, 5 }
```

Отложенное выполнение

Важная характеристика многих операций запросов заключается в том, что они выполняются не тогда, когда создаются, а когда происходит *перечисление* (другими словами, когда вызывается метод `MoveNext()` на перечислителе). Рассмотрим следующий запрос:

```
var numbers = new List<int> { 1 };

IEnumerable<int> query = numbers.Select (n => n * 10);
numbers.Add (2);  // Вставить дополнительный элемент

foreach (int n in query)
  Console.Write (n + "|");  // 10|20|
```

Дополнительное число, вставленное в список *после* конструирования запроса, присутствует в результате, поскольку любая фильтрация или сортировка не происходит вплоть до выполнения оператора `foreach`. Это называется *отложенным* или *ленивым* выполнением. Отложенное выполнение отвязывает *конструирование* запроса от его *выполнения*, позволяя строить запрос на протяжении нескольких шагов, а также делает возможным запрашивание базы данных без извлечения всех строк для клиента. Все стандартные операции запросов обеспечивают отложенное выполнение со следующими исключениями:

- операции, которые возвращают одиночный элемент или скалярное значение (*операции над элементами*, *агрегирования* и *квалификации*);
- операции преобразования `ToArray()`, `ToList()`, `ToDictionary()`, `ToLookup()` и `ToHashSet()`.

Операции преобразования удобны отчасти тем, что отменяют отложенное выполнение. Это может быть полезно для "замораживания" или кеширования результатов в определенный момент времени, чтобы избежать повторного выполнения запроса с

крупным объемом вычислений или запроса к удаленному источнику, такому как таблица Entity Framework. (Побочный эффект отложенного выполнения связан с тем, что запрос будет выполнен повторно, если позже будет предпринято его перечисление заново.)

В следующем примере демонстрируется операция `ToList()`:

```
var numbers = new List<int>() { 1, 2 };
List<int> timesTen = numbers
  .Select (n => n * 10)
  .ToList(); // Выполняется немедленно с помещением
             //   в List<int>
numbers.Clear();
Console.WriteLine (timesTen.Count);  // По-прежнему 2
```

НА ЗАМЕТКУ!

Подзапросы обеспечивают еще один уровень косвенности. Все, что находится в подзапросе, подпадает под отложенное выполнение, включая методы агрегирования и преобразования, т.к. сами подзапросы выполняются только отложенным образом по требованию. Предполагая, что `names` — строковый массив, подзапрос выглядит примерно так:

```
names.Where (
  n => n.Length ==
    names.Min (n2 => n2.Length))
```

Стандартные операции запросов

Стандартные операции запросов (реализованные в классе `System.Linq.Enumerable`) могут быть разделены на 12 категорий, которые кратко описаны в табл. 1.

В табл. 2–13 приведены описания всех операций запросов. Операции, выделенные полужирным, имеют специальную поддержку в языке C# (см. раздел "Выражения запросов" на стр. 182).

Таблица 1. Категории операций запросов

Категория	Описание	Применяется ли отложенное выполнение
Фильтрация	Возвращают подмножество элементов, которые удовлетворяют заданному условию	Да
Проецирование	Трансформируют каждый элемент с помощью лямбда-функции, дополнительно расширяя подпоследовательности	Да
Соединение	Объединяют элементы одной последовательности с элементами другой, используя эффективную с точки зрения времени стратегию поиска	Да
Упорядочение	Возвращают переупорядоченную последовательность	Да
Группирование	Группируют последовательность в подпоследовательности	Да
Работа с множествами	Принимают две последовательности одного и того же типа и возвращают их общность, сумму или разницу	Да
Работа с элементами	Выбирают одиночный элемент из последовательности	Нет
Агрегирование	Выполняют вычисление над последовательностью, возвращая скалярное значение (обычно число)	Нет
Квалификация	Выполняют проверку последовательности, возвращая `true` или `false`	Нет
Преобразование: импортирование	Преобразуют необобщенную последовательность в (поддерживающую запросы) обобщенную последовательность	Да
Преобразование: экспортирование	Преобразуют последовательность в массив, список, словарь или объект `Lookup`, вызывая немедленное выполнение	Нет
Генерация	Производят простую последовательность	Да

Таблица 2. Операции фильтрации

Метод	Описание
`Where()`	Возвращает подмножество элементов, удовлетворяющих заданному условию
`Take()`	Возвращает первые *x* элементов и отбрасывает остальные
`Skip()`	Пропускает первые *x* элементов и возвращает остальные
`TakeWhile()`	Выдает элементы входной последовательности до тех пор, пока заданный предикат остается равным `true`
`SkipWhile()`	Пропускает элементы входной последовательности до тех пор, пока заданный предикат остается равным `true`, и затем выдает остальные элементы
`Distinct()`	Возвращает последовательность, из которой исключены дубликаты

Таблица 3. Операции проецирования

Метод	Описание
`Select()`	Трансформирует каждый входной элемент с помощью заданного лямбда-выражения
`SelectMany()`	Трансформирует каждый входной элемент, а затем выравнивает и объединяет результирующие подпоследовательности

Таблица 4. Операции соединения

Метод	Описание
`Join()`	Применяет стратегию поиска для сопоставления элементов из двух коллекций, выдавая плоский результирующий набор
`GroupJoin()`	Подобен `Join()`, но выдает *иерархический* результирующий набор
`Zip()`	Перечисляет две последовательности за раз, возвращая последовательность, в которой к каждой паре элементов применена функция

Таблица 5. Операции упорядочения

Метод	Описание
`OrderBy()`, `ThenBy()`	Возвращают элементы, отсортированные в возрастающем порядке
`OrderByDescending()`, `ThenByDescending()`	Возвращают элементы, отсортированные в убывающем порядке
`Reverse()`	Возвращает элементы в обратном порядке

Таблица 6. Операция группирования

Метод	Описание
`GroupBy()`	Группирует последовательность в подпоследовательности

Таблица 7. Операции над множествами

Метод	Описание
`Concat()`	Выполняет конкатенацию двух последовательностей
`Union()`	Выполняет конкатенацию двух последовательностей, удаляя дубликаты
`Intersect()`	Возвращает элементы, присутствующие в обеих последовательностях
`Except()`	Возвращает элементы, присутствующие в первой, но не во второй последовательности

Таблица 8. Операции над элементами

Метод	Описание
`First()`, `FirstOrDefault()`	Возвращают первый элемент в последовательности или первый элемент, удовлетворяющий заданному предикату
`Last()`, `LastOrDefault()`	Возвращают последний элемент в последовательности или последний элемент, удовлетворяющий заданному предикату
`Single()`, `SingleOrDefault()`	Эквивалентны `First()`/`FirstOrDefault()`, но генерируют исключение, если обнаружено более одного совпадения
`ElementAt()`, `ElementAtOrDefault()`	Возвращают элемент в указанной позиции
`DefaultIfEmpty()`	Возвращает последовательность из одного элемента, значением которого является `null` или `default(TSource)`, если последовательность не содержит элементов

Таблица 9. Операции агрегирования

Метод	Описание
`Count()`, `LongCount()`	Возвращают общее количество элементов во входной последовательности или количество элементов, удовлетворяющих заданному предикату
`Min()`, `Max()`	Возвращают наименьший или наибольший элемент в последовательности
`Sum()`, `Average()`	Подсчитывают числовую сумму или среднее значение для элементов в последовательности
`Aggregate()`	Выполняет специальное агрегирование

Таблица 10. Операции квалификации

Метод	Описание
`Contains()`	Возвращает `true`, если входная последовательность содержит заданный элемент
`Any()`	Возвращает `true`, если какие-нибудь элементы удовлетворяют заданному предикату
`All()`	Возвращает `true`, если все элементы удовлетворяют заданному предикату
`SequenceEqual()`	Возвращает `true`, если вторая последовательность содержит элементы, идентичные элементам во входной последовательности

Таблица 11. Операции преобразования: импортирование

Метод	Описание
`OfType()`	Преобразует `IEnumerable` в `IEnumerable<T>`, отбрасывая элементы неподходящих типов
`Cast()`	Преобразует `IEnumerable` в `IEnumerable<T>`, генерируя исключение при наличии элементов неподходящих типов

Таблица 12. Операции преобразования: экспортирование

Метод	Описание
`ToArray()`	Преобразует `IEnumerable<T>` в `T[]`
`ToList()`	Преобразует `IEnumerable<T>` в `List<T>`
`ToDictionary()`	Преобразует `IEnumerable<T>` в `Dictionary<TKey,TValue>`
`ToHashSet()`	Преобразует `IEnumerable<T>` в `HashSet<T>`
`ToLookup()`	Преобразует `IEnumerable<T>` в `ILookup<TKey,TElement>`
`AsEnumerable()`	Приводит вниз к `IEnumerable<T>`
`AsQueryable()`	Приводит или преобразует в `IQueryable<T>`

Таблица 13. Операции генерации

Метод	Описание
`Empty()`	Создает пустую последовательность
`Repeat()`	Создает последовательность повторяющихся элементов
`Range()`	Создает последовательность целочисленных значений

Цепочки операций запросов

Для построения более сложных запросов допускается объединять операции запросов в цепочки. Например, следующий запрос извлекает все строки, содержащие букву "а", сортирует их по длине и затем преобразует результаты в верхний регистр:

```
string[] names = { "Том","Кирк","Габи","Банни","Гас" };
IEnumerable<string> query = names
  .Where (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select (n => n.ToUpper());
foreach (string name in query)
  Console.Write (name + "|");
РЕЗУЛЬТАТ:
ГАС|ГАБИ|БАННИ|
```

Where(), OrderBy() и Select() — это стандартные операции запросов, которые распознаются как вызовы расширяющих методов класса Enumerable. Операция Where() выдает отфильтрованную версию входной последовательности; операция OrderBy() — отсортированную версию входной последовательности; операция Select() — последовательность, в которой каждый входной элемент трансформирован, или *спроецирован*, с помощью заданного лямбда-выражения (n.ToUpper() в рассмотренном случае). Данные протекают слева направо через цепочку операций, поэтому они сначала фильтруются, затем сортируются и, наконец, проецируются. Конечный результат напоминает производственную линию с ленточными конвейерами, показанную на рис. 6.

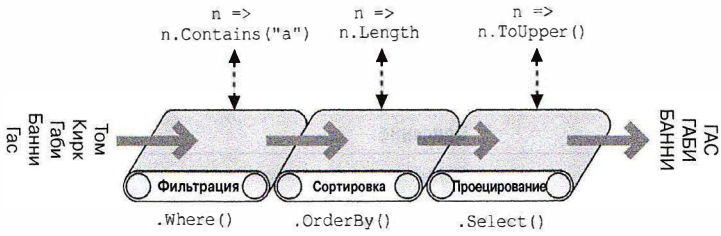


Рис. 6. *Пример цепочки операций запросов*

Отложенное выполнение соблюдается операциями повсеместно, так что ни фильтрация, ни сортировка, ни проецирование не происходят до тех пор, пока не начнется фактическое перечисление результатов запроса.

Выражения запросов

До сих пор мы писали запросы, вызывая расширяющие методы в классе Enumerable. В настоящей книге мы называем такое *текучим синтаксисом*. В C# также обеспечивается специальная языковая поддержка для написания запросов, которая называется *выражениями запросов*. Вот как предыдущий запрос выглядит в форме выражения запроса:

```
IEnumerable<string> query =
  from n in names
  where n.Contains ("a")
  orderby n.Length
  select n.ToUpper();
```

Выражение запроса всегда начинается с конструкции `from` и заканчивается либо конструкцией `select`, либо конструкцией `group`. Конструкция `from` объявляет *переменную диапазона* (в данном случае `n`), которую можно воспринимать как переменную, предназначенную для обхода входной последовательности — почти как в цикле `foreach`. На рис. 7 иллюстрируется полный синтаксис.

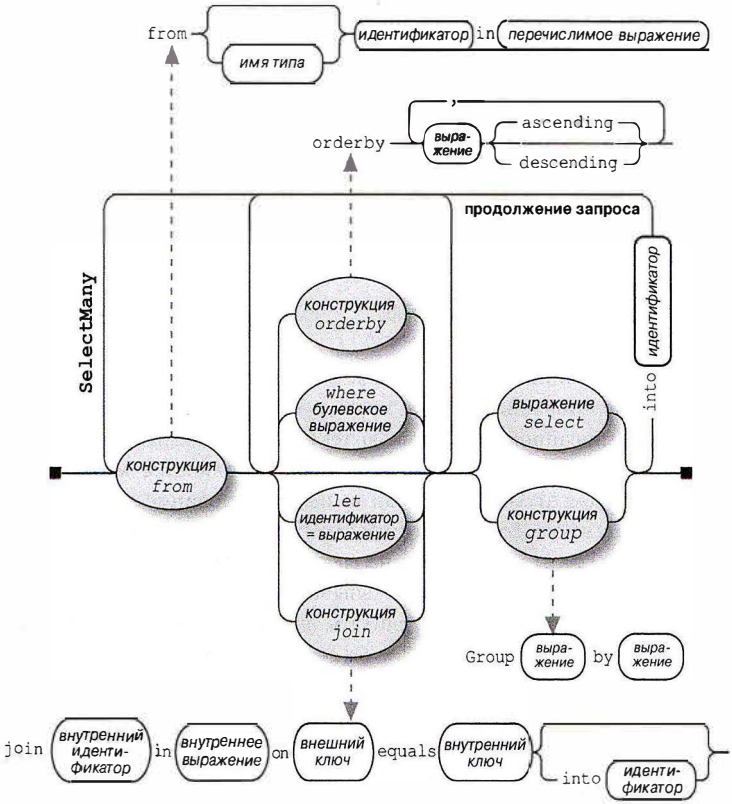


Рис. 7. *Синтаксис выражений запросов*

НА ЗАМЕТКУ!

Если вы знакомы с языком SQL, то синтаксис выражений запросов LINQ — с конструкцией `from` в начале и конструкцией `select` в конце — может выглядеть странным. На самом деле синтаксис выражений запросов более логичен, поскольку конструкции появляются *в порядке, в котором они выполняются*. Это позволяет среде Visual Studio с помощью средства IntelliSense предлагать подсказки по мере набора запроса, а также упрощает правила установления области видимости для подзапросов.

Компилятор обрабатывает выражения запросов, транслируя их в текучий синтаксис. Он делает это почти механически — примерно так, как транслирует операторы `foreach` в вызовы `GetEnumerator()` и `MoveNext()`:

```
IEnumerable<string> query = names
  .Where (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select (n => n.ToUpper());
```

Затем операции `Where()`, `OrderBy()` и `Select()` распознаются с использованием тех же правил, которые применялись бы к запросу, написанному с помощью текучего синтаксиса. В данном случае операции привязываются к расширяющим методам в классе `Enumerable` (предполагая импортирование пространства имен `System.Linq`), потому что `names` реализует интерфейс `IEnumerable<string>`. Тем не менее, при трансляции синтаксиса запросов компилятор не оказывает специальной поддержки классу `Enumerable`. Можете считать, что компилятор механически вводит слова *`Where`*, *`OrderBy`* и *`Select`* внутрь оператора, после чего компилирует его, как если бы вы набирали имена методов самостоятельно. В итоге обеспечивается гибкость в способе их распознавания — например, операции в запросах Entity Framework привязываются к расширяющим методам в классе `Queryable`.

Сравнение синтаксиса выражений запросов и текучего синтаксиса

И синтаксису выражений запросов, и текучему синтаксису присущи свои преимущества.

Выражения запросов поддерживают только небольшое подмножество операций запросов, в частности:

- `Where()`,
- `Select()`,
- `SelectMany()`
- `OrderBy()`,
- `ThenBy()`,
- `OrderByDescending()`,
- `ThenByDescending()`
- `GroupBy()`,
- `Join()`,
- `GroupJoin()`

Запросы, которые используют другие операции, придется записывать либо полностью в текучем синтаксисе, либо в смешанном синтаксисе, например:

```
string[] names = { "Том","Кирк","Габи","Банни","Гас" };
IEnumerable<string> query =
  from n in names
  where n.Length == names.Min (n2 => n2.Length)
  select n;
```

Приведенный запрос возвращает имена с наименьшей длиной (`Том` и `Гас`). Подзапрос (выделенный полужирным) вычисляет минимальную длину имен и получает значение 3. Для данного подзапроса должен применяться текучий синтаксис, т.к. операция `Min()` в синтаксисе выражений запросов не поддерживается. Однако для внешнего запроса по-прежнему можно использовать синтаксис выражений запросов.

Главное преимущество синтаксиса выражений запросов заключается в том, что он способен радикально упростить запросы, в которых задействованы следующие компоненты:

- конструкция `let` для введения новой переменной параллельно с переменной диапазона;
- множество генераторов (`SelectMany()`), за которыми следует ссылка на внешнюю переменную диапазона;
- эквивалент операции `Join()` или `GroupJoin()`, за которым следует ссылка на внешнюю переменную диапазона.

Ключевое слово `let`

Ключевое слово `let` вводит новую переменную параллельно с переменной диапазона. В качестве примера предположим, что необходимо вывести список имен, длина которых без учета гласных составляет более двух символов:

```
string[] names = { "Том","Кирк","Габи","Банни","Гас" };
IEnumerable<string> query =
  from n in names
  let vowelless = Regex.Replace (n, "[аоиеёэыуюя]", "")
  where vowelless.Length > 2
  orderby vowelless
  select n + " - " + vowelless;
```

Вот вывод, полученный при перечислении результатов этого запроса:

```
Кирк - Крк
Банни - Бнн
```

Конструкция `let` выполняет вычисление над каждым элементом, не утрачивая исходный элемент. В нашем запросе последующие конструкции (`where`, `orderby` и `select`) имеют доступ к `n` и `vowelless`. Запрос способен включать любое количество конструкций `let`, и они могут сопровождаться дополнительными конструкциями `where` и `join`.

Компилятор транслирует ключевое слово `let` путем проецирования во временный анонимный тип, который содержит исходные и трансформированные элементы:

```
IEnumerable<string> query = names
  .Select (n => new
    {
      n = n,
      vowelless = Regex.Replace (n, "[аоиеёэыуюя]", "")
    }
  )
  .Where (temp0 => (temp0.vowelless.Length > 2))
  .OrderBy (temp0 => temp0.vowelless)
  .Select (temp0 => ((temp0.n + "-") + temp0.vowelless))
```

Продолжение запросов

Если необходимо добавить конструкции *после* конструкции `select` или `group`, тогда придется использовать ключевое слово `into`, чтобы "продолжить" запрос. Например:

```
from c in "Большой хитрый быстрый лис".Split()
select c.ToUpper() into upper
where upper.StartsWith ("Б")
select upper
```

```
РЕЗУЛЬТАТ:
"БОЛЬШОЙ", "БЫСТРЫЙ"
```

После конструкции `into` предыдущая переменная диапазона находится за пределами области видимости.

Компилятор просто транслирует запросы с ключевым словом `into` в более длинную цепочку операций:

```
"Большой хитрый быстрый лис".Split()
  .Select (c => c.ToUpper())
  .Where (upper => upper.StartsWith ("Б"))
```

(Компилятор опускает финальную конструкцию `Select(upper=> upper)`, потому что она избыточна.)

Множество генераторов

Запрос может включать несколько генераторов (конструкций `from`). Например:

```
int[] numbers = { 1, 2, 3 };
string[] letters = { "a", "б" };
IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;
```

Результатом является векторное произведение, очень похожее на то, что можно было бы получить с помощью вложенных циклов `foreach`:

```
"1a", "1б", "2a", "2б", "3a", "3б"
```

При наличии в запросе более одной конструкции `from` компилятор выпускает вызов метода `SelectMany()`:

```
IEnumerable<string> query = numbers.SelectMany (
  n => letters,
  (n, l) => (n.ToString() + l));
```

Метод `SelectMany()` выполняет вложенные циклы. Он проходит по всем элементам в исходной коллекции (`numbers`), трансформируя каждый элемент с помощью лямбда-выражения (`letters`). В итоге генерируется последовательность *подпоследовательностей*, которая затем подвергается перечислению. Финальные выходные элементы определяются вторым лямбда-выражением (`n.ToString() + l`).

Если дополнительно применить конструкцию `where`, то векторное произведение можно отфильтровать и спроецировать результат подобно *соединению*:

```
string[] players = { "Том", "Гас", "Мэри" };
IEnumerable<string> query =
  from name1 in players
  from name2 in players
  where name1.CompareTo (name2) < 0
  orderby name1, name2
  select name1 + " или " + name2;
РЕЗУЛЬТАТ:
{ "Гас или Мэри", "Гас или Том", "Мэри или Том" }
```

Трансляция такого запроса в текучий синтаксис сложнее и требует временной анонимной проекции. Возможность автоматического выполнения такой трансляции является одним из основных преимуществ выражений запросов.

Выражению во втором генераторе разрешено пользоваться первой переменной диапазона:

```
string[] fullNames =
  { "Анна Вильямс", "Джон Фред Смит", "Сью Грин" };
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()
  select name + " из " + fullName;
РЕЗУЛЬТАТЫ:
Анна из Анна Вильямс
Вильямс из Анна Вильямс
Джон из Джон Фред Смит
Фред из Джон Фред Смит
Смит из Джон Фред Смит
Сью из Сью Грин
Грин из Сью Грин
```

Запрос работает, поскольку выражение `fullName.Split()` выдает *последовательность* (массив строк).

Множество генераторов широко применяется в запросах к базам данных для выравнивания отношений "родительский–дочерний" и для выполнения ручных соединений.

Соединение

В LINQ доступны три операции *соединения*, из которых главными являются `Join()` и `GroupJoin()`, выполняющие соединения на основе ключей поиска. Операции `Join()` и `GroupJoin()` поддерживают только подмножество функциональности, которую вы получаете благодаря множеству генераторов или `SelectMany()`, но они обладают более высокой производительностью при использовании в локальных запросах, потому что применяют стратегию поиска на базе хеш-таблиц, а не выполняют вложенные циклы. (В случае запросов Entity Framework операции соединения не имеют никаких преимуществ перед множеством генераторов.)

Операции `Join()` и `GroupJoin()` поддерживают только *эквисоединения* (т.е. условие соединения должно использовать операцию эквивалентности). Существуют два метода: `Join()` и `GroupJoin()`. Метод `Join()` выдает плоский результирующий набор, тогда как метод `GroupJoin()` — иерархический результирующий набор.

Синтаксис выражений запросов для плоского соединения выглядит так:

```
from внешняя-переменная in внешняя-последовательность
join внутренняя-переменная in внутренняя-последовательность
  on внешнее-выражение-ключей
  equals внутреннее-выражение-ключей
```

Например, имея следующие коллекции:

```
var customers = new[]
{
  new { ID = 1, Name = "Том" },
  new { ID = 2, Name = "Кирк" },
  new { ID = 3, Name = "Гарри" }
};
var purchases = new[]
{
```

```
  new { CustomerID = 1, Product = "Дом" },
  new { CustomerID = 2, Product = "Лодка" },
  new { CustomerID = 2, Product = "Автомобиль" },
  new { CustomerID = 3, Product = "Путевка" }
};
```

мы можем выполнить соединение следующим образом:

```
IEnumerable<string> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  select c.Name + " приобрел " + p.Product;
```

Компилятор транслирует этот запрос так, как показано ниже:

```
customers.Join (      // внешняя коллекция
  purchases,          // внутренняя коллекция
  c => c.ID,          // внешний селектор ключей
  p => p.CustomerID, // внутренний селектор ключей
  (c, p) =>          // селектор результатов
  c.Name + " приобрел " + p.Product
);
```

Вот результат:

```
Том приобрел Дом
Кирк приобрел Лодка
Кирк приобрел Автомобиль
Гарри приобрел Путевка
```

В случае локальных последовательностей при обработке крупных коллекций операции `Join()` и `GroupJoin()` более эффективны, чем `SelectMany()`, поскольку они сначала загружают внутреннюю последовательность в хеш-таблицу поиска по ключу. Тем не менее, того же самого результата в равной степени эффективно можно достичь и так:

```
from c in customers
from p in purchases
where c.ID == p.CustomerID
select c.Name + " приобрел " + p.Product;
```

GroupJoin()

Операция `GroupJoin()` делает ту же работу, что и `Join()`, но вместо плоского результата выдает иерархический результат, сгруппированный по каждому внешнему элементу.

Синтаксис выражений запросов для GroupJoin() такой же, как и для Join(), но за конструкцией join следует ключевое слово into. Ниже приведен простейший пример, в котором задействованы коллекции customers и purchases, подготовленные в предыдущем разделе:

```
IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
  select custPurchases;    // custPurchases -
                           //   последовательность
```

НА ЗАМЕТКУ!

Конструкция into транслируется в GroupJoin(), только когда она появляется прямо после конструкции join. При расположении после конструкции select или group она означает *продолжение запроса*. Такие два применения ключевого слова into существенно отличаются, хотя и обладают одной общей характеристикой: в обоих случаях вводится новая переменная диапазона.

Результатом будет последовательность последовательностей, по которой можно было бы организовать перечисление следующим образом:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

Однако это не особенно полезно, т.к. purchaseSequence не имеет ссылок на внешнего заказчика из customers. Чаще всего в проекции производится ссылка на внешнюю переменную диапазона:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Тот же самый результат (но менее эффективно, для локальных запросов) можно было бы получить за счет проецирования в анонимный тип, который включает подзапрос:

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases =
    purchases.Where (p => c.ID == p.CustomerID)
}
```

Zip()

Операция Zip() является простейшей операцией соединения. Она перечисляет две последовательности за раз (подобно застежке-молнии (zipper)) и возвращает последовательность, полученную в результате применения функции к каждой паре элементов.

Например:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "три", "пять", "семь", "проигнорирован" };
IEnumerable<string> zip =
  numbers.Zip (words, (n, w) => n + "=" + w);
```

приводит к получению последовательности с такими элементами:

```
3=три
5=пять
7=семь
```

Добавочные элементы в любой из входных последовательностей игнорируются. Операция Zip() не поддерживается в запросах к базам данных.

Упорядочение

С помощью ключевого слова orderby последовательность сортируется. Разрешено указывать любое количество выражений, по которым нужно сортировать:

```
string[] names = { "Том", "Кирк", "Габи", "Банни", "Гас" };
IEnumerable<string> query = from n in names
                            orderby n.Length, n
                            select n;
```

Сортировка осуществляется сначала по длине и затем по имени, поэтому результаты получаются такими:

```
Гас, Том, Габи, Кирк, Банни
```

Компилятор транслирует первое выражение `orderby` в вызов `OrderBy()`, а последующие выражения — в вызовы `ThenBy()`:

```
IEnumerable<string> query = names
  .OrderBy (n => n.Length)
  .ThenBy (n => n)
```

Операция `ThenBy()` скорее *уточняет* результаты предшествующей сортировки, нежели *заменяет* их.

После любого выражения `orderby` можно помещать ключевое слово `descending`:

```
orderby n.Length descending, n
```

Запрос транслируется в:

```
.OrderByDescending (n => n.Length).ThenBy (n => n)
```

НА ЗАМЕТКУ!

Операции упорядочения возвращают расширенный тип `IEnumerable<T>` по имени `IOrderedEnumerable<T>`. В данном интерфейсе определена дополнительная функциональность, требуемая операцией `ThenBy()`.

Группирование

Операция `GroupBy()` превращает плоскую входную последовательность в последовательность *групп*. Например, приведенный ниже код группирует имена по их длине:

```
string[] names = { "Том","Кирк","Габи","Банни","Гас" };
var query = from name in names
            group name by name.Length;
```

Компилятор транслирует этот запрос в:

```
IEnumerable<IGrouping<int,string>> query =
  names.GroupBy (name => name.Length);
```

Вот как выполнить перечисление результата:

```
foreach (IGrouping<int,string> grouping in query)
{
  Console.Write ("\r\n Length=" + grouping.Key + ":");
  foreach (string name in grouping)
    Console.Write (" " + name);
}
```

```
РЕЗУЛЬТАТЫ:
Length=3: Том Гас
Length=4: Кирк Габи
Length=5: Банни
```

Метод `Enumerable.GroupBy()` работает путем чтения входных элементов во временный словарь списков, так что все элементы с одинаковыми ключами попадают в один и тот же подсписок. Затем он выдает последовательность *групп*. Группа представляет собой последовательность со свойством `Key`:

```
public interface IGrouping <TKey,TElement>
  : IEnumerable<TElement>, IEnumerable
{
  // Ключ применяется к подпоследовательности
  // как к единому целому
  TKey Key { get; }
}
```

По умолчанию элементы в каждой группе являются нетрансформированными входными элементами, если только не указан аргумент `elementSelector`. Следующий запрос проецирует входные элементы в верхний регистр:

```
from name in names
group name.ToUpper() by name.Length
```

Он транслируется в:

```
names.GroupBy (
  name => name.Length,
  name => name.ToUpper() )
```

Подколлекции не выдаются в порядке следования ключей. Операция `GroupBy()` не выполняет *сортировку* (на самом деле она предохраняет исходное упорядочение). Чтобы отсортировать, понадобится добавить операцию `OrderBy()` (что в первую очередь означает добавление конструкции `into`, т.к. `group by` обычно заканчивает запрос):

```
from name in names
group name.ToUpper() by name.Length into grouping
orderby grouping.Key
select grouping
```

Продолжения запроса часто используются в запросах `group by`.

Следующий запрос отфильтровывает группы, которые имеют в них точно два совпадения:

```
from name in names
group name.ToUpper() by name.Length into grouping
where grouping.Count() == 2
select grouping
```

НА ЗАМЕТКУ!

Конструкция `where` после `group by` эквивалентна конструкции `HAVING` в языке SQL. Она применяется к каждой подпоследовательности или группе как к единому целому, а не к содержащимся в ней индивидуальным элементам.

`OfType()` и `Cast()`

Операции `OfType()` и `Cast()` принимают необобщенную коллекцию `IEnumerable` и выдают обобщенную последовательность `IEnumerable<T>`, которой впоследствии можно отправить запрос:

```
var classicList = new System.Collections.ArrayList();
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Польза указанных операций в том, что появляется возможность запрашивать коллекции, разработанные до выхода версии C# 2.0 (где был введен интерфейс `IEnumerable<T>`), такие как `ControlCollection` из пространства имен `System.Windows.Forms`.

Операции `Cast()` и `OfType()` отличаются своим поведением, когда встречается входной элемент с несовместимым типом: `Cast()` генерирует исключение, а `OfType()` игнорирует такой элемент.

Правила совместимости элементов соответствуют аналогичным правилам для операции `is` в языке C#. Ниже показана внутренняя реализация `Cast()`:

```
public static IEnumerable<TSource> Cast <TSource>
              (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

Язык C# поддерживает операцию `Cast()` в синтаксисе запросов — нужно просто поместить тип элемента непосредственно после ключевого слова `from`:

```
from int x in classicList ...
```

Это транслируется в:

```
from x in classicList.Cast <int>() ...
```

Динамическое связывание

Динамическое связывание откладывает *связывание* — процесс распознавания типов, членов и операций — с этапа компиляции до времени выполнения. Динамическое связывание удобно, когда на этапе компиляции *вы* знаете, что определенная функция, член или операция существует, но *компилятору* об этом неизвестно. Обычно подобное происходит при взаимодействии с динамическими языками (такими как IronPython) и COM, а также в сценариях, в которых иначе использовалась бы рефлексия.

Динамический тип объявляется с помощью контекстного ключевого слова `dynamic`:

```
dynamic d = GetSomeObject();
d.Quack();
```

Динамический тип предлагает компилятору смягчить требования. Мы ожидаем, что тип времени выполнения `d` должен иметь метод `Quack()`. Мы просто не можем проверить это статически. Поскольку `d` относится к динамическому типу, компилятор откладывает связывание `Quack()` с `d` до времени выполнения. Понимание смысла такого действия требует уяснения различий между *статическим связыванием* и *динамическим связыванием*.

Сравнение статического и динамического связывания

Канонический пример связывания предусматривает отображение имени на специфическую функцию при компиляции выражения. Для компиляции следующего выражения компилятор должен найти реализацию метода по имени `Quack()`:

```
d.Quack();
```

Давайте предположим, что статическим типом d является Duck:

```
Duck d = ...
d.Quack();
```

В самом простом случае компилятор осуществляет связывание за счет поиска в типе Duck метода без параметров по имени Quack(). Если найти такой метод не удалось, тогда компилятор распространяет поиск на методы, принимающие необязательные параметры, методы базовых классов Duck и расширяющие методы, которые принимают Duck в своем первом параметре. Если совпадений не обнаружено, то возникает ошибка компиляции. Независимо от того, к какому методу произведено связывание, суть в том, что связывание делается компилятором, и оно полностью зависит от статических сведений о типах операндов (в данном случае d). Именно поэтому такой процесс называется *статическим связыванием*.

А теперь изменим статический тип d на object:

```
object d = ...
d.Quack();
```

Вызов Quack() приводит к ошибке на этапе компиляции, т.к. несмотря на то, что хранящееся в d значение способно содержать метод по имени Quack(), компилятор не может об этом знать, поскольку единственная информация, которой он располагает — тип переменной, которым в рассматриваемом случае является object. Но давайте изменим статический тип d на dynamic:

```
dynamic d = ...
d.Quack();
```

Тип dynamic похож на object — он в равной степени не описывает тип. Отличие заключается в том, что тип dynamic допускает применение способами, которые на этапе компиляции не известны. Динамический объект связывается во время выполнения на основе своего типа времени выполнения, а не типа при компиляции. Когда компилятор встречает динамически связываемое выражение (которым в общем случае является выражение, содержащее любое значение типа dynamic), он просто упаковывает его так, чтобы связывание могло быть произведено позже во время выполнения.

Если динамический объект реализует интерфейс IDynamicMetaObjectProvider, то во время выполнения для связыва-

ния используется данный интерфейс. Если же нет, тогда связывание проходит в основном так же, как в ситуации, когда компилятору известен тип динамического объекта времени выполнения. Эти две альтернативы называются *специальным связыванием* и *языковым связыванием*.

Специальное связывание

Специальное связывание происходит, когда динамический объект реализует интерфейс `IDynamicMetaObjectProvider`. Хотя интерфейс `IDynamicMetaObjectProvider` можно реализовать в типах, которые вы пишете на языке C#, и поступать так удобно, более распространенный случай предусматривает запрос объекта, реализующего `IDynamicMetaObjectProvider`, из динамического языка, который внедрен в .NET посредством исполняющей среды динамического языка (Dynamic Language Runtime — DLR), скажем, IronPython или IronRuby. Объекты из таких языков неявно реализуют интерфейс `IDynamicMetaObjectProvider` в качестве способа для прямого управления смыслом выполняемых над ними операций. Ниже приведен простой пример:

```
using System;
using System.Dynamic;

public class Test
{
  static void Main()
  {
    dynamic d = new Duck();
    d.Quack();   // Выводит Вызван метод Quack
    d.Waddle();  // Выводит Вызван метод Waddle
  }
}
public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args,
    out object result)
  {
    Console.WriteLine ("Вызван метод " + binder.Name);
    result = null;
    return true;
  }
}
```

Класс `Duck` в действительности не имеет метода `Quack()`. Взамен он применяет специальное связывание для перехвата и интерпретации всех обращений к методам.

Специальное связывание более подробно обсуждается в главе 20 книги *C# 8.0. Справочник. Полное описание языка*.

Языковое связывание

Языковое связывание происходит, когда динамический объект не реализует интерфейс `IDynamicMetaObjectProvider`. Языковое связывание удобно при работе с неудачно спроектированными типами или внутренними ограничениями системы типов .NET. Например, встроенные числовые типы неудачны в том, что они не имеют общего интерфейса. Ранее было показано, что методы могут быть привязаны динамически; то же самое справедливо и для операций:

```
static dynamic Mean (dynamic x, dynamic y) => (x+y) / 2;
static void Main()
{
  int x = 3, y = 4;
  Console.WriteLine (Mean (x, y));
}
```

Преимущество очевидно — не приходится дублировать код для каждого числового типа. Тем не менее, утрачивается безопасность типов, из-за чего возрастает риск генерации исключений во время выполнения вместо получения ошибок на этапе компиляции.

НА ЗАМЕТКУ!

Динамическое связывание обходит статическую безопасность типов, но не динамическую безопасность типов. В отличие от рефлексии с помощью динамического связывания невозможно обойти правила доступности членов.

Языковое связывание во время выполнения преднамеренно ведет себя максимально похожим на статическое связывание образом, как будто типы времени выполнения динамических объектов были известны еще на этапе компиляции. В предыдущем примере поведение программы окажется идентичным, если метод

Mean() жестко закодировать для работы с типом int. Наиболее заметным исключением при проведении аналогии между статическим и динамическим связыванием являются расширяющие методы, которые рассматриваются в разделе "Невызываемые функции" на стр. 204.

НА ЗАМЕТКУ!

Динамическое связывание также наносит ущерб производительности. Однако из-за механизмов кеширования среды DLR повторяющиеся обращения к одному и тому же динамическому выражению оптимизируются, позволяя эффективно работать с динамическими выражениями в цикле. Такая оптимизация снижает типичные временные издержки при выполнении простого динамического выражения на современном оборудовании до менее чем 100 наносекунд.

Исключение RuntimeBinderException

Если привязка к члену не удается, тогда генерируется исключение RuntimeBinderException. Его можно считать ошибкой этапа компиляции, перенесенной на время выполнения:

```
dynamic d = 5;
d.Hello();  // Генерируется RuntimeBinderException
```

Исключение генерируется оттого, что тип int не имеет метода Hello().

Представление типа dynamic во время выполнения

Между типами dynamic и object имеется глубокая эквивалентность. Исполняющая среда трактует следующее выражение как true:

```
typeof (dynamic) == typeof (object)
```

Данный принцип распространяется на составные типы и массивы:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Подобно объектной ссылке динамическая ссылка может указывать на объект любого типа (за исключением типов указателей):

```
dynamic x = "строка";
Console.WriteLine (x.GetType().Name);  // String
x = 123;     // Ошибки нет (несмотря на то,
             //   что переменная та же самая)
Console.WriteLine (x.GetType().Name);  // Int32
```

Структурно какие-либо отличия между объектной ссылкой и динамической ссылкой отсутствуют. Динамическая ссылка просто разрешает выполнение динамических операций над объектом, на который она указывает. Чтобы выполнить любую динамическую операцию над `object`, тип `object` можно преобразовать в `dynamic`:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("строка");
Console.WriteLine (o);  // строка
```

Динамические преобразования

Тип `dynamic` поддерживает неявные преобразования в и из всех остальных типов. Чтобы преобразование прошло успешно, тип времени выполнения динамического объекта должен быть неявно преобразуемым в целевой статический тип.

В следующем примере генерируется исключение `RuntimeBinderException`, т.к. тип `int` не может быть неявно преобразован в `short`:

```
int i = 7;
dynamic d = i;
long l = d; // Нормально - работает неявное преобразование
short j = d; // Генерируется RuntimeBinderException
```

Сравнение `var` и `dynamic`

Несмотря на внешнее сходство типов `var` и `dynamic`, разница между ними существенна:

- `var` говорит: позволить *компилятору* выяснить тип;
- `dynamic` говорит: позволить *исполняющей среде* выяснить тип.

Вот иллюстрация:

```
dynamic x = "строка";       // Статическим типом
                            //   является dynamic
var y = "строка"; // Статическим типом является string
int i = x;        // Ошибка во время выполнения
int j = y;        // Ошибка на этапе компиляции
```

Динамические выражения

Поля, свойства, методы, события, конструкторы, индексаторы, операции и преобразования могут вызываться динамически.

Попытка потребления результата динамического выражения с возвращаемым типом `void` пресекается — точно как в случае статически типизированного выражения. Отличие связано с тем, что ошибка возникает во время выполнения.

Выражения, содержащие динамические операнды, обычно сами являются динамическими, т.к. эффект отсутствия информации о типе имеет каскадный характер:

```
dynamic x = 2;
var y = x * 3;  // Статическим типом y является dynamic
```

Из этого правила существует пара очевидных исключений. Во-первых, приведение динамического выражения к статическому типу дает статическое выражение. Во-вторых, вызовы конструкторов всегда дают статические выражения — даже если они производятся с динамическими аргументами.

Кроме того, существует несколько краевых случаев, когда выражение, содержащее динамический аргумент, является статическим, включая передачу индекса массиву и выражения для создания делегатов.

Распознавание перегруженных версий динамических членов

В каноническом сценарии использования `dynamic` участвует динамический *получатель*. Это значит, что получателем динамического вызова функции является динамический объект:

```
dynamic x = ...;
x.Foo (123);  // x - получатель
```

Тем не менее, динамическое связывание не ограничивается получателями: аргументы методов также пригодны для динами-

ческого связывания. Следствием вызова функции с динамическими аргументами будет откладывание распознавания перегруженных версий с этапа компиляции до времени выполнения:

```
static void Foo (int x)    => Console.WriteLine ("int");
static void Foo (string x) => Console.WriteLine ("str");

static void Main()
{
  dynamic x = 5;
  dynamic y = "строка";

  Foo (x);  // int
  Foo (y);  // str
}
```

Распознавание перегруженных версий во время выполнения также называется *множественной диспетчеризацией* и полезно в реализации паттернов проектирования, таких как *"Посетитель"* (Visitor).

Если динамический получатель не задействован, то компилятор может статически выполнить базовую проверку успешности динамического вызова: он проверяет, существует ли функция с правильным именем и корректным количеством параметров. Если кандидаты не найдены, тогда возникает ошибка на этапе компиляции.

Если функция вызывается со смесью динамических и статических аргументов, то окончательный выбор метода будет отражать смесь решений динамического и статического связывания:

```
static void X(object x, object y) => Console.Write ("oo");
static void X(object x, string y) => Console.Write ("os");
static void X(string x, object y) => Console.Write ("so");
static void X(string x, string y) => Console.Write ("ss");

static void Main()
{
  object o = "Привет";
  dynamic d = "Пока";
  X (o, d);  // os
}
```

Вызов `X(o,d)` привязывается динамически, потому что один из его аргументов, `d`, определен как `dynamic`. Но поскольку переменная `o` статически известна, связывание — хотя оно происходит динамически — будет применять ее. В рассматриваемом примере

механизм распознавания перегруженных версий выберет вторую реализацию `Foo()` из-за статического типа `o` и типа времени выполнения `d`. Другими словами, компилятор является "настолько статическим, насколько он способен быть".

Невызываемые функции

Некоторые функции не могут быть вызваны динамически. Вызывать нельзя:

- расширяющие методы (через синтаксис расширяющих методов);
- любые члены интерфейса (через интерфейс);
- члены базового класса, сокрытые подклассом.

Причина в том, что динамическое связывание требует двух порций информации: имени вызываемой функции и объекта, на котором должна вызываться функция. Однако в каждом из трех невызываемых сценариев участвует *дополнительный тип*, который известен только на этапе компиляции. И нет никакого способа указать такие дополнительные типы динамически.

При вызове расширяющих методов этот дополнительный тип представляет собой расширяющий класс, выбранный неявно посредством директив `using` в исходном коде (которые после компиляции исчезают). При обращении к членам через интерфейс дополнительный тип сообщается через неявное или явное приведение. (При явной реализации фактически невозможно вызвать член без приведения к типу интерфейса.) Похожая ситуация возникает при вызове скрытого члена базового класса: дополнительный тип должен быть указан либо через приведение, либо через ключевое слово `base` — и во время выполнения этот дополнительный тип утрачивается.

Перегрузка операций

Операции могут быть перегружены для предоставления специальным типам более естественного синтаксиса. Перегрузку операций наиболее целесообразно использовать при реализации специальных структур, которые представляют относительно примитивные типы данных. Например, хорошим кандидатом на перегрузку операций может служить специальный числовой тип.

Разрешено перегружать следующие символические операции:

```
+   -   *   /   ++   --   !   ~   %   &   |   ^
==  !=  <   <<  >>   >
```

Явные и неявные преобразования также могут быть перегружены (с применением ключевых слов `explicit` и `implicit`), равно как литералы `true` и `false`, а также унарные операции + и –.

Составные операции присваивания (например, += и /=) автоматически перегружаются при перегрузке обычных операций (т.е. + и /).

Функции операций

Операция перегружается за счет объявления *функции операции* (с ключевым словом `operator`). Функция операции должна быть статической и, по крайней мере, один из операндов обязан иметь тип, в котором объявлена функция операции. В следующем примере мы определяем структуру по имени `Note`, представляющую музыкальную ноту, и затем перегружаем операцию +:

```
public struct Note
{
  int value;

  public Note (int semitonesFromA)
    => value = semitonesFromA;

  public static Note operator + (Note x, int semitones)
  {
    return new Note (x.value + semitones);
  }
}
```

Перегруженная версия позволяет добавлять к `Note` значение `int`:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Поскольку мы перегрузили операцию +, можно также использовать операцию +=:

```
CSharp += 2;
```

Подобно методам и свойствам, начиная с версии C# 6, функции операций, которые состоят из одиночного выражения, разре-

шено записывать более кратко с помощью синтаксиса функций, сжатых до выражений:

```
public static Note operator + (Note x, int semitones)
  => new Note (x.value + semitones);
```

Перегрузка операций эквивалентности и сравнения

Операции эквивалентности и сравнения часто перегружаются при написании структур и в редких случаях — при написании классов. При перегрузке операций эквивалентности и сравнения должны соблюдаться специальные правила и обязательства.

Парность

Компилятор C# требует, чтобы операции, которые представляют собой логические пары, были определены обе. Такими операциями являются (== !=), (< >) и (<= >=).

`Equals()` и `GetHashCode()`

При перегрузке операций == и != для типа обычно необходимо переопределять методы `Equals()` и `GetHashCode()` класса `object`, чтобы коллекции и хеш-таблицы могли надежно работать с типом.

`IComparable` и `IComparable<T>`

Если перегружаются операции < и >, то обычно должны быть реализованы интерфейсы `IComparable` и `IComparable<T>`.

Расширим предыдущий пример, чтобы показать, каким образом можно было бы перегрузить операции эквивалентности структуры `Note`:

```
public static bool operator == (Note n1, Note n2)
  => n1.value == n2.value;
public static bool operator != (Note n1, Note n2)
  => !(n1.value == n2.value);
public override bool Equals (object otherNote)
{
  if (!(otherNote is Note)) return false;
  return this == (Note)otherNote;
}
// Для нашего хеш-кода будет использоваться хеш-код value:
public override int GetHashCode() => value.GetHashCode();
```

Специальные неявные и явные преобразования

Неявные и явные преобразования являются перегружаемыми операциями. Как правило, эти операции перегружаются для того, чтобы сделать преобразования между тесно связанными типами (такими как числовые типы) лаконичными и естественными.

Как объяснялось при обсуждении типов, логическое обоснование неявных преобразований заключается в том, что они должны всегда выполняться успешно и не приводить к потере информации при преобразовании. В противном случае должны быть определены явные преобразования.

В следующем примере мы определяем преобразования между типом `Note` и типом `double` (с помощью которого представляется частота в герцах данной ноты):

```
...
// Преобразование в герцы
public static implicit operator double (Note x)
  => 440 * Math.Pow (2,(double) x.value / 12 );

// Преобразование из герц
//   (с точностью до ближайшего полутона)
public static explicit operator Note (double x)
  => new Note ((int) (0.5 + 12 * (Math.Log(x/440)
            / Math.Log(2)) ));
...
Note n = (Note)554.37;      // явное преобразование
double x = n;               // неявное преобразование
```

НА ЗАМЕТКУ!

Кое в чем данный пример нельзя считать естественным: в действительности такие преобразования можно реализовать эффективнее с помощью метода `ToFrequency()` и (статического) метода `FromFrequency()`.

Операции `as` и `is` игнорируют специальные преобразования.

Атрибуты

Вам уже знакомо понятие снабжения элементов кода признаками в форме модификаторов, таких как `virtual` или `ref`. Эти

конструкции встроены в язык. *Атрибуты* представляют собой расширяемый механизм для добавления специальной информации к элементам кода (сборкам, типам, членам, возвращаемым значениям и параметрам). Такая расширяемость удобна для служб, глубоко интегрированных в систему типов, и не требует специальных ключевых слов или конструкций в языке C#.

Хороший сценарий для атрибутов касается *сериализации* — процесса преобразования произвольных объектов в определенный формат и обратно с целью хранения или передачи. В таком случае атрибут на поле может указывать трансляцию между представлением поля в C# и его представлением в применяемом формате.

Классы атрибутов

Атрибут определяется классом, который унаследован (прямо или косвенно) от абстрактного класса `System.Attribute`. Чтобы присоединить атрибут к элементу кода, перед элементом кода понадобится указать имя типа атрибута в квадратных скобках. Например, в приведенном ниже коде к классу `Foo` присоединяется атрибут `ObsoleteAttribute`:

```
[ObsoleteAttribute]
public class Foo {...}
```

Атрибут `ObsoleteAttribute` распознается компилятором и приводит к тому, что компилятор выдаст предупреждение, если встретится ссылка на тип или член, помеченный как устаревший (obsolete). По соглашению имена всех типов атрибутов оканчиваются словом *`Attribute`*. Данное соглашение поддерживается компилятором C# и позволяет опускать суффикс `Attribute`, когда присоединяется атрибут:

```
[Obsolete]
public class Foo {...}
```

Тип `ObsoleteAttribute` объявлен в пространстве имен `System` следующим образом (для краткости код упрощен):

```
public sealed class ObsoleteAttribute : Attribute {...}
```

Именованные и позиционные параметры атрибутов

Атрибуты могут иметь параметры. В показанном ниже примере мы применяем к классу атрибут `XmlElementAttribute`, который сообщает классу `XmlSerializer` (из пространства имен `System.Xml.Serialization`) о том, что объект представлен в XML, и принимает несколько *параметров атрибута*. В итоге атрибут отображает класс `CustomerEntity` на XML-элемент по имени `Customer`, принадлежащий пространству имен `http://oreilly.com`:

```
[XmlElement ("Customer",
             Namespace="http://oreilly.com")]
public class CustomerEntity { ... }
```

Параметры атрибутов относятся к одной из двух категорий: позиционные и именованные. В предыдущем примере первый аргумент является *позиционным параметром*, а второй — *именованным параметром*. Позиционные параметры соответствуют параметрам открытых конструкторов типа атрибута. Именованные параметры соответствуют открытым полям или открытым свойствам типа атрибута.

При указании атрибута должны включаться позиционные параметры, которые соответствуют одному из конструкторов класса атрибута. Именованные параметры необязательны.

Цели атрибутов

Неявно целью атрибута является элемент кода, находящийся непосредственно за атрибутом, который обычно представляет собой тип или член типа. Тем не менее, атрибуты можно присоединять к сборке. При этом требуется явно указывать цель атрибута. Вот как с помощью атрибута `CLSCompliant` задать соответствие общеязыковой спецификации (Common Language Specification — CLS) для целой сборки:

```
[assembly:CLSCompliant(true)]
```

Указание нескольких атрибутов

Для одного элемента кода допускается указывать несколько атрибутов. Атрибуты могут быть заданы либо внутри единственной пары квадратных скобок (и разделяться запятыми), либо в

отдельных парах квадратных скобок (или с помощью комбинации двух способов). Следующие два примера семантически идентичны:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}
[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}
```

Определение специальных атрибутов

За счет создания подклассов класса `System.Attribute` можно определять собственные атрибуты. Например, мы могли бы использовать следующий специальный атрибут для пометки метода, подлежащего модульному тестированию:

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int Repetitions;
  public string FailureMessage;
  public TestAttribute () : this (1) { }
  public TestAttribute (int repetitions)
    => Repetitions = repetitions;
}
```

Ниже показано, как можно было бы применить данный атрибут:

```
class Foo
{
  [Test]
  public void Method1() { ... }
  [Test(20)]
  public void Method2() { ... }
  [Test(20, FailureMessage="Время отладки!")]
  public void Method3() { ... }
}
```

Атрибут `AttributeUsage` указывает конструкцию или комбинацию конструкций, к которым может быть применен специальный атрибут. Перечисление `AttributeTargets` включает такие члены, как `Class`, `Method`, `Parameter` и `Constructor` (а также `All` для объединения всех целей).

Извлечение атрибутов во время выполнения

Существуют два стандартных способа извлечения атрибутов во время выполнения:

- вызов метода `GetCustomAttributes()` на любом объекте `Type` или `MemberInfo`;
- вызов метода `Attribute.GetCustomAttribute()` или `Attribute.GetCustomAttributes()`.

Последние два метода перегружены для приема любого объекта рефлексии, который соответствует допустимой цели атрибута (`Type`, `Assembly`, `Module`, `MemberInfo` или `ParameterInfo`).

Вот как можно выполнить перечисление всех методов предшествующего класса `Foo`, которые имеют атрибут `TestAttribute`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
  TestAttribute att = (TestAttribute)
    Attribute.GetCustomAttribute
      (mi, typeof (TestAttribute));
  if (att != null)
    Console.WriteLine (
      "{0} будет тестироваться; reps={1}; msg={2}",
      mi.Name, att.Repetitions, att.FailureMessage);
}
```

Вывод выглядит так:

```
Method1 будет тестироваться; reps=1; msg=
Method2 будет тестироваться; reps=20; msg=
Method3 будет тестироваться; reps=20; msg=Время отладки!
```

Атрибуты информации о вызывающем компоненте

Начиная с версии C# 5.0, необязательные параметры можно помечать одним из трех *атрибутов информации о вызывающем компоненте*, которые инструктируют компилятор о необходимости передачи информации, полученной из исходного кода вызывающего компонента, в стандартное значение параметра:

- `[CallerMemberName]` применяет имя члена вызывающего компонента;
- `[CallerFilePath]` применяет путь к файлу исходного кода вызывающего компонента;
- `[CallerLineNumber]` применяет номер строки в файле исходного кода вызывающего компонента.

В следующем методе `Foo()` демонстрируется использование всех трех атрибутов:

```
using System;
using System.Runtime.CompilerServices;

class Program
{
  static void Main() => Foo();

  static void Foo (
    [CallerMemberName] string memberName = null,
    [CallerFilePath] string filePath = null,
    [CallerLineNumber] int lineNumber = 0)
  {
    Console.WriteLine (memberName);
    Console.WriteLine (filePath);
    Console.WriteLine (lineNumber);
  }
}
```

Предполагая, что код находится в файле `c:\source\test\Program.cs`, вывод будет таким:

```
Main
c:\source\test\Program.cs
6
```

Как и со стандартными необязательными параметрами, подстановка делается в *месте вызова*. Следовательно, показанный выше метод `Main()` является "синтаксическим сахаром" для следующего кода:

```
static void Main()
  => Foo ("Main", @"c:\source\test\Program.cs", 6);
```

Атрибуты информации о вызывающем компоненте удобны при написании функций регистрации в журнале, а также при реализации шаблонов уведомления об изменениях. Например, мы можем вызвать метод, подобный приведенному ниже, изнутри

средства доступа set определенного свойства без необходимости в указании имени этого свойства:

```
void RaisePropertyChanged (
  [CallerMemberName] string propertyName = null)
{
  ...
}
```

Асинхронные функции

Ключевые слова await и async (введенные в версии C# 5.0) поддерживают *асинхронное программирование* — стиль программирования, при котором длительно выполняющиеся функции делают бо́льшую часть или даже всю свою работу *после* того, как управление возвращено вызывающему компоненту. Оно отличается от нормального *синхронного* программирования, когда длительно выполняющиеся функции *блокируют* вызывающий компонент до тех пор, пока операция не будет завершена. Асинхронное программирование подразумевает *параллелизм*, потому что длительно выполняющаяся операция продолжается *параллельно* с функционированием вызывающего компонента. Разработчик асинхронной функции инициирует такой параллелизм либо через многопоточность (для операций, интенсивных в плане вычислений), либо посредством механизма обратных вызовов (для операций, интенсивных в плане ввода-вывода).

НА ЗАМЕТКУ!

Многопоточность, параллелизм и асинхронное программирование — обширные темы. Им посвящены две главы в книге *C# 8.0. Справочник. Полное описание языка* и вдобавок они обсуждаются по ссылке http://albahari.com/threading.

Например, рассмотрим следующий *синхронный* метод, который является длительно выполняющимся и интенсивным в плане вычислений:

```
int ComplexCalculation()
{
```

```
  double x = 2;
  for (int i = 1; i < 100000000; i++)
    x += Math.Sqrt (x) / i;
  return (int)x;
}
```

Метод `ComplexCalculation()` блокирует вызывающий компонент на протяжении нескольких секунд, пока он выполняется, и только затем возвращает результат вычисления вызывающему компоненту:

```
int result = ComplexCalculation();
// В какой-то момент позже:
Console.WriteLine (result);  // 116
```

В среде CLR определен класс по имени `Task<TResult>` (из пространства имен `System.Threading.Tasks`), предназначенный для инкапсуляции понятия операции, которая завершается в будущем. Сгенерировать объект `Task<TResult>` для операции, интенсивной в плане вычислений, можно с помощью вызова метода `Task.Run()`, который сообщает среде CLR о необходимости запуска указанного делегата в отдельном потоке, выполняющемся параллельно вызывающему компоненту:

```
Task<int> ComplexCalculationAsync()
{
  return Task.Run (() => ComplexCalculation());
}
```

Этот метод является *асинхронным*, потому что он немедленно возвращает управление вызывающему компоненту и продолжает выполняться параллельно. Однако нам нужен какой-то механизм, который дал бы возможность вызывающему компоненту указывать, что должно произойти, когда операция завершится и результат станет доступным. Класс `Task<TResult>` решает проблему, открывая доступ к методу `GetAwaiter()`, который позволяет вызывающему компоненту присоединять *продолжение*:

```
Task<int> task = ComplexCalculationAsync();
var awaiter = task.GetAwaiter();
awaiter.OnCompleted (() =>     // Продолжение
{
  int result = awaiter.GetResult();
  Console.WriteLine (result);  // 116
});
```

Тем самым операции сообщается о том, что по завершении она должна выполнить указанный делегат. Продолжение сначала вызывает метод `GetResult()`, который возвращает результат вычисления. (Или, если задача *потерпела неудачу*, сгенерировав исключение, то вызов `GetResult()` сгенерирует это исключение повторно.) Затем продолжение выводит на консоль результат через `Console.WriteLine()`.

Ключевые слова `await` и `async`

Ключевое слово `await` упрощает присоединение продолжений. Начиная с базового сценария, компилятор расширяет конструкции:

```
var результат = await выражение;
оператор(ы);
```

в код, функционально подобный показанному ниже:

```
var awaiter = выражение.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var результат = awaiter.GetResult();
  оператор(ы);
});
```

НА ЗАМЕТКУ!

Компилятор также выпускает код для оптимизации сценария синхронного (немедленного) завершения операции. Распространенная причина для немедленного завершения асинхронной операции возникает, когда операция реализует внутренний механизм кеширования и результат уже находится в кеше.

Следовательно, вот как мы можем вызвать определенный ранее метод `ComplexCalculationAsync()`:

```
int result = await ComplexCalculationAsync();
Console.WriteLine (result);
```

Чтобы код скомпилировался, к содержащему его методу понадобится добавить модификатор `async`:

```
async void Test()
{
  int result = await ComplexCalculationAsync();
  Console.WriteLine (result);
}
```

Модификатор `async` инструктирует компилятор о том, что `await` необходимо трактовать как ключевое слово, а не идентификатор, иначе внутри метода возникла бы неоднозначность (в итоге код, написанный до выхода версии C# 5.0, где слово `await` могло быть выбрано для идентификатора, по-прежнему будет успешно компилироваться). Модификатор `async` может применяться только к методам (и лямбда-выражениям), которые возвращают `void` либо (как будет показано позже) объект `Task` или `Task<TResult>`.

НА ЗАМЕТКУ!

Модификатор `async` подобен модификатору `unsafe` в том, что не оказывает никакого влияния на сигнатуру метода или открытые метаданные; он воздействует только на то, что происходит *внутри* метода.

Методы с модификатором `async` называются *асинхронными функциями*, потому что они сами обычно являются асинхронными. Чтобы увидеть почему, давайте посмотрим, каким образом процесс выполнения проходит через асинхронную функцию.

Встретив выражение `await`, управление (обычно) возвращается вызывающему компоненту, что очень похоже на поведение `yield return` в итераторе. Но перед возвратом исполняющая среда присоединяет к ожидающей задаче признак продолжения, который гарантирует, что когда задача завершится, то поток управления перейдет обратно в метод и продолжит с того места, где он его оставил. Если в задаче возникает ошибка, тогда ее исключение генерируется повторно (благодаря вызову `GetResult()`); в противном случае выражению `await` присваивается возвращаемое значение задачи.

НА ЗАМЕТКУ!

Реализация средой CLR метода `OnCompleted()` объекта ожидания задачи гарантирует, что по умолчанию сигналы продолжения отправляются через текущий *контекст синхронизации* при его наличии. На деле это означает, что если в сценариях с обогащенными пользовательскими интерфейсами (WPF, UWP (универсальная платформа Windows) и Windows Forms) используется `await` внутри потока пользовательского интерфейса, то выполнение кода будет продолжено в том же самом потоке. В итоге упрощается обеспечение безопасности к потокам.

Выражение, на котором применяется `await`, обычно является задачей. Тем не менее, компилятор устроит любой объект с методом `GetAwaiter()`, который возвращает *объект с возможностью ожидания*. Такой объект реализует метод `INotifyCompletion.OnCompleted()`, а также имеет надлежащим образом типизированный метод `GetResult()` и булевское свойство `IsCompleted`, которое выполняет проверку на предмет синхронного завершения.

Обратите внимание, что выражение `await` оценивается как имеющее тип `int`; причина в том, что ожидаемым выражением было `Task<int>` (чей метод `GetAwaiter().GetResult()` возвращает значение `int`).

Ожидание необобщенной задачи вполне законно и генерирует выражение `void`:

```
await Task.Delay (5000);
Console.WriteLine ("Прошло пять секунд!");
```

Статический метод `Task.Delay()` возвращает объект `Task`, который завершается за указанное количество миллисекунд. *Синхронным* эквивалентом `Task.Delay()` является `Thread.Sleep()`.

Тип `Task` представляет собой необобщенный базовый класс для `Task<TResult>` и функционально эквивалентен `Task<TResult>`, но не производит какого-либо результата.

Захват локального состояния

Реальная мощь выражений `await` заключается в том, что они могут находиться почти где угодно в коде. В частности, выражение `await` может появляться на месте любого выражения (внутри асинхронной функции) кроме блока `catch` или `finally`, выражения `lock` либо контекста `unsafe`.

В следующем примере `await` используется внутри цикла:

```
async void Test()
{
  for (int i = 0; i < 10; i++)
  {
    int result = await ComplexCalculationAsync();
    Console.WriteLine (result);
  }
}
```

При первом выполнении `ComplexCalculationAsync()` управление возвращается вызывающему компоненту благодаря выражению `await`. Когда метод завершается (или терпит неудачу), выполнение возобновляется с того места, которое оно ранее покинуло, с сохраненными значениями локальных переменных и счетчиков циклов. Компилятор достигает этого путем превращения такого кода в конечный автомат, как он поступает с итераторами.

В отсутствие ключевого слова `await` ручное применение продолжений означает необходимость в написании чего-то эквивалентного конечному автомату, что традиционно было фактором, усложняющим асинхронное программирование.

Написание асинхронных функций

В любой асинхронной функции возвращаемый тип `void` можно заменить типом `Task`, чтобы сделать сам метод *пригодным* для асинхронного выполнения (и поддержки `await`). Вносить какие-то другие изменения не потребуется:

```
async Task PrintAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  Console.WriteLine (answer);
}
```

Обратите внимание, что в теле метода мы явно не возвращаем задачу. Компилятор самостоятельно произведет задачу, которая сигнализирует о завершении данного метода (или о возникновении необработанного исключения). В результате облегчается создание цепочек асинхронных вызовов:

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Готово");
}
```

(И поскольку `Go()` возвращает `Task`, сам метод `Go()` поддерживает ожидание посредством `await`.) Компилятор разворачивает асинхронные функции, возвращающие задачи, в код, косвенно использующий класс `TaskCompletionSource` для создания задачи, которая затем отправляет сигнал о завершении или отказе.

НА ЗАМЕТКУ!

`TaskCompletionSource` — это тип CLR, позволяющий создавать задачи, которыми вы управляете вручную, сигнализируя об их завершении посредством выдачи результата (или об отказе с помощью генерации исключения). В отличие от `Task.Run()` тип `TaskCompletionSource` не связывает поток на протяжении выполнения операции. Он также применяется при написании методов, интенсивных в плане ввода-вывода и возвращающих объекты задач (вроде `Task.Delay()`).

Цель в том, чтобы при завершении асинхронного метода, возвращающего объект задачи, через продолжение обеспечить возможность передачи управления в то место кода, где происходит его ожидание.

Возвращение `Task<TResult>`

Если в теле метода возвращается тип `TResult`, то можно возвращать `Task<TResult>`:

```
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  // answer имеет тип int, поэтому метод
  //   возвращает Task<int>
  return answer;
}
```

Продемонстрировать работу метода `GetAnswerToLife()` можно, вызвав его из метода `PrintAnswerToLife()` (который в свою очередь вызывается из `Go()`):

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Готово");
}
async Task PrintAnswerToLife()
{
  int answer = await GetAnswerToLife();
  Console.WriteLine (answer);
}
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;
}
```

Асинхронные функции делают асинхронное программирование похожим на синхронное. Ниже приведен синхронный эквивалент нашей схемы вызовов, где вызов `Go()` дает тот же самый результат после блокирования в течение пяти секунд:

```
void Go()
{
  PrintAnswerToLife();
  Console.WriteLine ("Готово");
}
void PrintAnswerToLife()
{
  int answer = GetAnswerToLife();
  Console.WriteLine (answer);
}
```

```
int GetAnswerToLife()
{
  Thread.Sleep (5000);
  int answer = 21 * 2;
  return answer;
}
```

Это также иллюстрирует базовый принцип проектирования с использованием асинхронных функций в C#, который предусматривает написание синхронных версий методов и последующую замену вызовов *синхронных* методов вызовами *асинхронных* методов, а также применение к ним `await`.

Параллелизм

Мы только что продемонстрировали наиболее распространенный подход, при котором ожидание функций, возвращающих объекты задач, производится немедленно после вызова. В результате получается последовательный поток выполнения программы, который логически подобен своему синхронному эквиваленту.

Вызов асинхронного метода без его ожидания позволяет писать код, который следует выполнять параллельно. Например, показанный ниже код два раза параллельно запускает метод `PrintAnswerToLife()`:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

За счет применения `await` к обеим операциям мы "заканчиваем" параллелизм в данной точке (и повторно генерируем любые исключения, которые могли поступить из этих задач). Класс `Task` предоставляет статический метод по имени `WhenAll()`, позволяющий достичь того же результата чуть более эффективно. Метод `WhenAll()` возвращает задачу, которая завершается, когда завершаются все переданные ему задачи:

```
await Task.WhenAll (PrintAnswerToLife(),
                    PrintAnswerToLife());
```

Метод `WhenAll()` называется *комбинатором задач*. (Класс `Task` также предлагает комбинатор задач по имени `WhenAny()`, возвращающий задачу, которая завершается, когда завершается *любая* из задач, переданных `WhenAny()`.) Комбинаторы задач под-

робно рассматриваются в книге *C# 8.0. Справочник. Полное описание языка.*

Асинхронные лямбда-выражения

Подобно тому, как обычные *именованные* методы могут быть асинхронными:

```
async Task NamedMethod()
{
  await Task.Delay (1000);
  Console.WriteLine ("Начало");
}
```

асинхронными могут быть и *неименованные* методы (лямбда-выражения и анонимные методы), если предварить их ключевым словом async:

```
Func<Task> unnamed = async () =>
{
  await Task.Delay (1000);
  Console.WriteLine ("Начало");
};
```

Вызывать их и применять к ним await можно тем же самым способом:

```
await NamedMethod();
await unnamed();
```

Асинхронные лямбда-выражения могут использоваться при присоединении обработчиков событий:

```
myButton.Click += async (sender, args) =>
{
  await Task.Delay (1000);
  myButton.Content = "Готово";
};
```

Это более лаконично, чем следующий код, который обеспечивает тот же самый эффект:

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
{
  await Task.Delay (1000);
  myButton.Content = "Готово";
};
```

Асинхронные лямбда-выражения могут также возвращать Task<TResult>:

```
Func<Task<int>> unnamed = async () =>
{
  await Task.Delay (1000);
  return 123;
};
int answer = await unnamed();
```

Асинхронные потоки (C# 8)

В версии C# 8 появилась явная поддержка асинхронных перечислителей и итераторов (*асинхронных потоков*), основанная на следующей паре интерфейсов, которые являются асинхронными аналогами интерфейсов перечисления, описанных в разделе "Перечисление и итераторы" на стр. 152:

```
public interface IAsyncEnumerable<out T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator (...);
}

public interface IAsyncEnumerator<out T>: IAsyncDisposable
{
  T Current { get; }
  ValueTask<bool> MoveNextAsync();
}
```

Тип ValueTask<T> — это структура, представляющая собой оболочку для Task<T>, которая по поведению эквивалентна Task<T> за исключением того, что она делает возможным более эффективное выполнение, когда задача завершается синхронно (что может произойти при перечислении последовательности). Интерфейс IAsyncDisposable является асинхронной версией IDisposable и обеспечивает возможность выполнения очистки в случае ручной реализации интерфейсов:

```
public interface IAsyncDisposable
{
  ValueTask DisposeAsync();
}
```

НА ЗАМЕТКУ!

Действие по извлечению каждого элемента из последовательности (`MoveNextAsync()`) представляет собой асинхронную операцию, поэтому асинхронные потоки подходят, когда элементы поступают постепенно (как при обработке данных из видеопотока). И наоборот, тип `Task<IEnumerable<T>>` больше подходит, когда последовательность задерживается *как одно целое*, а элементы поступают все вместе.

Для генерации асинхронного потока понадобится написать метод, который сочетает в себе принципы итераторов и асинхронных методов. Другими словами, метод должен включать `yield return` и `await`, а также возвращать `IAsyncEnumerable<T>`:

```
async IAsyncEnumerable<int> RangeAsync (
  int start, int count, int delay)
{
  for (int i = start; i < start + count; i++)
  {
    await Task.Delay (delay);
    yield return i;
  }
}
```

Чтобы задействовать асинхронный поток, необходимо использовать оператор `await foreach`:

```
await foreach (var number in RangeAsync (0, 10, 100))
  Console.WriteLine (number);
```

Небезопасный код и указатели

Язык C# поддерживает прямые манипуляции с памятью через указатели внутри блоков кода, которые помечены как небезопасные и скомпилированы с опцией компилятора `/unsafe`. Типы указателей полезны главным образом при взаимодействии с API-интерфейсами C, но могут также применяться для доступа в память за пределами управляемой кучи или для "горячих" точек, критичных к производительности.

Основы указателей

Для каждого типа значения или ссылочного типа *V* имеется соответствующий тип указателя *V**. Экземпляр указателя хранит адрес переменной. Тип указателя может быть (небезопасно) приведен к любому другому типу указателя. Ниже описаны основные операции над указателями.

Операция	Описание
`&`	Операция *взятия адреса* возвращает указатель на адрес переменной
`*`	Операция *разыменования* возвращает переменную по адресу, который задан указателем
`->`	Операция *указателя на член* является синтаксическим сокращением, т.е. `x->y` эквивалентно `(*x).y`

Небезопасный код

Помечая тип, член типа или блок операторов ключевым словом `unsafe`, вы разрешаете использовать типы указателей и выполнять операции над указателями в стиле C++ внутри этой области видимости. Ниже показан пример применения указателей для быстрой обработки битовой карты:

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

Небезопасный код способен выполняться быстрее, чем соответствующая ему безопасная реализация. В последнем случае код потребовал бы вложенного цикла с индексацией в массиве и проверкой границ. Небезопасный метод C# может также оказаться быстрее, чем вызов внешней функции C, поскольку не будет никаких накладных расходов, связанных с покиданием управляемой среды выполнения.

Оператор fixed

Оператор fixed необходим для закрепления управляемого объекта, такого как битовая карта в предыдущем примере. Во время выполнения программы многие объекты распределяются в куче и впоследствии освобождаются. Во избежание нежелательных затрат или фрагментации памяти сборщик мусора перемещает объекты внутри кучи. Указатель на объект бесполезен, если адрес объекта может измениться во время обращения к нему, а потому оператор fixed сообщает сборщику мусора о необходимости "закрепления" объекта, чтобы он никуда не перемещался. Это может оказать влияние на эффективность программы во время выполнения, так что фиксированные блоки должны использоваться только кратковременно, а распределения памяти в куче внутри фиксированного блока следует избегать.

В операторе fixed можно получать указатель на тип значения, массив типов значений или строку. В случае массивов и строк указатель будет фактически указывать на первый элемент, который относится к типу значения.

Типы значений, объявленные непосредственно внутри ссылочных типов, требуют закрепления ссылочных типов, как показано ниже:

```
class Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    fixed (int* p = &test.x)  // Закрепляет test
    {
      *p = 9;
    }
    System.Console.WriteLine (test.x);
  }
}
```

Операция указателя на член

В дополнение к операциям & и * язык C# также предлагает операцию -> в стиле C++, которая может применяться при работе со структурами:

```
struct Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    Test* p = &test;
    p->x = 9;
    System.Console.WriteLine (test.x);
  }
}
```

Ключевое слово stackalloc

Вы можете явно выделять память в блоке внутри стека с помощью ключевого слова `stackalloc`. Из-за распределения в стеке время жизни блока памяти ограничивается выполнением метода, в точности как для любой другой локальной переменной. Блок может использовать операцию `[]` для индексации внутри памяти:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
  Console.WriteLine (a[i]);  // Вывод низкоуровневых
                             //   значений из памяти
```

Буферы фиксированных размеров

Для выделения памяти в блоке внутри структуры применяется ключевое слово `fixed`:

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30];
}

unsafe class UnsafeClass
{
  UnsafeUnicodeString uus;

  public UnsafeClass (string s)
  {
    uus.Length = (short)s.Length;
    fixed (byte* p = uus.Buffer)
    for (int i = 0; i < s.Length; i++)
      p[i] = (byte) s[i];
  }
}
```

Буферы фиксированных размеров — это не массивы: если бы поле `Buffer` было массивом, то оно содержало бы ссылку на объект, хранящийся в (управляемой) куче, а не 30 байтов внутри самой структуры.

В приведенном примере ключевое слово `fixed` также используется для закрепления в куче объекта, содержащего буфер (который будет экземпляром `UnsafeClass`).

void*

Указатель void (`void*`) не выдвигает никаких предположений относительно типа лежащих в основе данных и удобен для функций, которые имеют дело с низкоуровневой памятью. Существует неявное преобразование из любого типа указателя в `void*`. Указатель `void*` не допускает разыменования и выполнения над ним арифметических операций. Например:

```
unsafe static void Main()
{
  short[] a = {1,1,2,3,5,8,13,21,34,55};
  fixed (short* p = a)
  {
    // Операция sizeof возвращает размер типа
    //   значения в байтах
    Zap (p, a.Length * sizeof (short));
  }
  foreach (short x in a)
    System.Console.WriteLine (x);  // Выводит все нули
}
unsafe static void Zap (void* memory, int byteCount)
{
  byte* b = (byte*) memory;
  for (int i = 0; i < byteCount; i++)
    *b++ = 0;
}
```

Директивы препроцессора

Директивы препроцессора снабжают компилятор дополнительной информацией о разделах кода. Наиболее распространенными директивами препроцессора являются директивы условной компиляции, которые предоставляют способ включения либо исключения разделов кода из процесса компиляции.

Например:

```
#define DEBUG
class MyClass
{
  int x;
  void Foo()
  {
    #if DEBUG
    Console.WriteLine ("Тестирование: x = {0}", x);
    #endif
  }
  ...
}
```

В классе `MyClass` оператор внутри метода `Foo()` компилируется условно в зависимости от существования символа `DEBUG`. Если удалить определение символа `DEBUG`, тогда этот оператор в `Foo()` компилироваться не будет. Символы препроцессора могут определяться внутри файла исходного кода (что и было сделано в примере), а также передаваться компилятору с помощью опции командной строки `/define:`*символ* либо в файле проекта в случае использования Visual Studio или MSBuild.

В директивах `#if` и `#elif` можно применять операции `||`, `&&` и `!` для выполнения логических действий *ИЛИ*, *И* и *НЕ* над несколькими символами. Представленная ниже директива указывает компилятору на необходимость включения следующего за ней кода, если определен символ `TESTMODE` и не определен символ `DEBUG`:

```
#if TESTMODE && !DEBUG
...
```

Однако имейте в виду, что вы не строите обычное выражение C#, а символы, которыми вы оперируете, не имеют абсолютно никакого отношения к *переменным* — статическим или каким-то другим.

Директивы `#error` и `#warning` предотвращают случайное неправильное использование директив условной компиляции, заставляя компилятор генерировать предупреждение или сообщение об ошибке, которое вызвано неподходящим набором символов компиляции.

Ниже перечислены все директивы препроцессора.

Директива препроцессора	Действие
`#define` *символ*	Определяет символ
`#undef` *символ*	Отменяет определение символа
`#if` *символ* [*операция символ2*] ...	Условная компиляция (операциями являются ==, !=, && и ||)
`#else`	Компилирует код до следующей директивы `#endif`
`#elif` *символ* [*операция символ2*]	Комбинирует ветвь `#else` и проверку `#if`
`#endif`	Заканчивает директивы условной компиляции
`#warning` *текст*	Заставляет компилятор вывести предупреждение с указанным текстом
`#error` *текст*	Заставляет компилятор вывести сообщение об ошибке с указанным текстом
`#line` [*номер* ["*файл*"] | `hidden`]	Номер задает строку в исходном коде; в "*файл*" указывается имя файла для помещения в вывод компилятора; `hidden` инструктирует инструменты отладки о необходимости пропуска кода от этой точки до следующей директивы `#line`
`#region` *имя*	Обозначает начало раздела
`#endregion`	Обозначает конец раздела
`#pragma warning`	См. следующий раздел
`#nullable` *вариант*	См. раздел "Ссылочные типы, допускающие значение `null` (C# 8)" на стр. 163

Директива #pragma warning

Компилятор генерирует предупреждение, когда обнаруживает в коде что-то, не кажущееся преднамеренным. В отличие от ошибок предупреждения обычно не препятствуют компиляции приложения.

Предупреждения компилятора могут быть исключительно полезными при выявлении ошибок. Тем не менее, их полезность снижается в случае выдачи *ложных* предупреждений. В крупном приложении очень важно поддерживать подходящее соотношение "сигнал-шум", если должны быть замечены "настоящие" предупреждения.

С этой целью компилятор позволяет избирательно подавлять выдачу предупреждений с помощью директивы `#pragma warning`. В следующем примере мы указываем компилятору о том, чтобы он не выдавал предупреждения о том, что поле `Message` не используется:

```
public class Foo
{
  static void Main() { }
  #pragma warning disable 414
  static string Message = "Сообщение";
  #pragma warning restore 414
}
```

Если в директиве `#pragma warning` отсутствует число, то будет отключена или восстановлена выдача предупреждений со всеми кодами. Если вы интенсивно применяли эту директиву, тогда можете скомпилировать код с переключателем `/warnaserror`, который сообщит компилятору о необходимости трактовать любые оставшиеся предупреждения как ошибки.

XML-документация

Документирующий комментарий — это порция встроенного XML-кода, которая документирует тип или член типа. Документирующий комментарий располагается непосредственно перед объявлением типа или члена и начинается с трех символов косой черты:

```
/// <summary>Прекращает выполняющийся запрос.</summary>
public void Cancel() { ... }
```

Многострочные комментарии записываются следующим образом:

```
/// <summary>
/// Прекращает выполняющийся запрос.
/// </summary>
public void Cancel() { ... }
```

или так (обратите внимание на дополнительную звездочку в начале):

```
/**
  <summary>Прекращает выполняющийся запрос.</summary>
*/
public void Cancel() { ... }
```

В случае компиляции с переключателем /doc (или при включении XML-документации в файле проекта) компилятор извлекает и накапливает документирующие комментарии в специальном XML-файле. С данным файлом связаны два основных сценария использования.

- Если он размещен в той же папке, что и скомпилированная сборка, то Visual Studio автоматически читает этот XML-файл и применяет информацию из него для предоставления списка членов IntelliSense потребителям сборки с таким же именем, как у XML-файла.
- Сторонние инструменты (такие как Sandcastle и NDoc) могут трансформировать этот XML-файл в справочный HTML-файл.

Стандартные XML-дескрипторы документации

Ниже перечислены стандартные XML-дескрипторы, которые распознаются Visual Studio и генераторами документации.

<summary>

```
<summary>...</summary>
```

Указывает всплывающую подсказку, которую средство IntelliSense должно отображать для типа или члена. Обычно это одиночная фраза или предложение.

<remarks>

```
<remarks>...</remarks>
```

Дополнительный текст, который описывает тип или член. Генераторы документации объединяют его с полным описанием типа или члена.

<param>

```
<param name="имя">...</param>
```

Объясняет параметр метода.

<returns>

```
<returns>...</returns>
```

Объясняет возвращаемое значение метода.

<exception>

```
<exception [cref="тип"]>...</exception>
```

Указывает исключение, которое метод может генерировать (в `cref` задается тип исключения).

<permission>

```
<permission [cref="тип"]>...</permission>
```

Указывает тип `IPermission`, требуемый документируемым типом или членом.

<example>

```
<example>...</example>
```

Обозначает пример (используемый генераторами документации). Как правило, содержит текст описания и исходный код (исходный код обычно заключен в дескриптор <c> или <code>).

<c>

```
<c>...</c>
```

Указывает внутристрочный фрагмент кода. Этот дескриптор обычно применяется внутри блока <example>.

<code>

```
<code>...</code>
```

Указывает многострочный пример кода. Этот дескриптор обычно используется внутри блока <example>.

<see>

```
<see cref="член">...</see>
```

Вставляет внутристрочную перекрестную ссылку на другой тип или член. Генераторы HTML-документации обычно преобразуют это в гиперссылку. Компилятор выдает предупреждение, если указано недопустимое имя типа или члена.

<seealso>

```
<seealso cref="член">...</seealso>
```

Вставляет перекрестную ссылку на другой тип или член. Генераторы документации обычно записывают это в отдельный раздел "See Also" ("См. также") в нижней части страницы.

<paramref>

```
<paramref name="имя"/>
```

Вставляет ссылку на параметр внутри дескриптора <summary> или <remarks>.

<list>

```
<list type=[ bullet | number | table ]>
  <listheader>
    <term>...</term>
    <description>...</description>
  </listheader>
  <item>
    <term>...</term>
    <description>...</description>
  </item>
</list>
```

Инструктирует генератор документации о необходимости выдачи маркированного (`bullet`), нумерованного (`number`) или табличного (`table`) списка.

<para>

```
<para>...</para>
```

Инструктирует генератор документации о необходимости форматирования содержимого в виде отдельного абзаца.

<include>

```
<include file='имя-файла'
  path='путь-к-дескриптору[@name="идентификатор"]'>
  ...
</include>
```

Выполняет объединение с внешним XML-файлом, содержащим документацию. В атрибуте `path` задается XPath-запрос к конкретному элементу из этого файла.

Предметный указатель

П

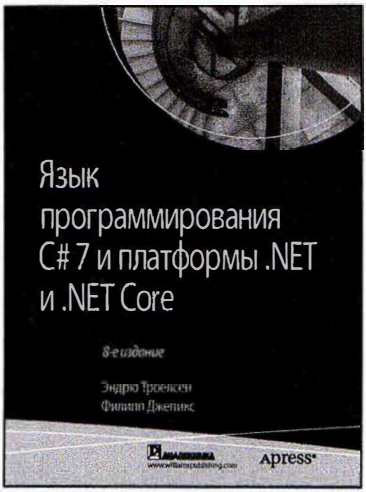
Р

С

Т

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 7 И ПЛАТФОРМЫ .NET И .NET CORE
8-Е ИЗДАНИЕ

Эндрю Троелсен
Филипп Джепикс



Эта классическая книга представляет собой всеобъемлющий источник сведений о языке программирования C# и о связанной с ним инфраструктуре. В 8-м издании книги вы найдете описание функциональных возможностей самых последних версий C# 7.0 и 7.1 и .NET 4.7, а также совершенно новые главы о легковесной межплатформенной инфраструктуре Microsoft .NET Core, включая версию .NET Core 2.0. Книга охватывает ASP.NET Core, Entity Framework (EF) Core и т.д. наряду с последними обновлениями платформы .NET, в том числе внесенными в Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF) и ASP.NET MVC. Книга предназначена для опытных разработчиков ПО, заинтересованных в освоении новых средств .NET 4.7, .NET Core и языка C#. Она будет служить всеобъемлющим руководством и настольным справочником как для тех, кто впервые переходит на платформу .NET, так и для тех, кто ранее писал приложения для предшествующих версий .NET.

www.williamspublishing.com

ISBN 978-5-6040723-1-8

в продаже