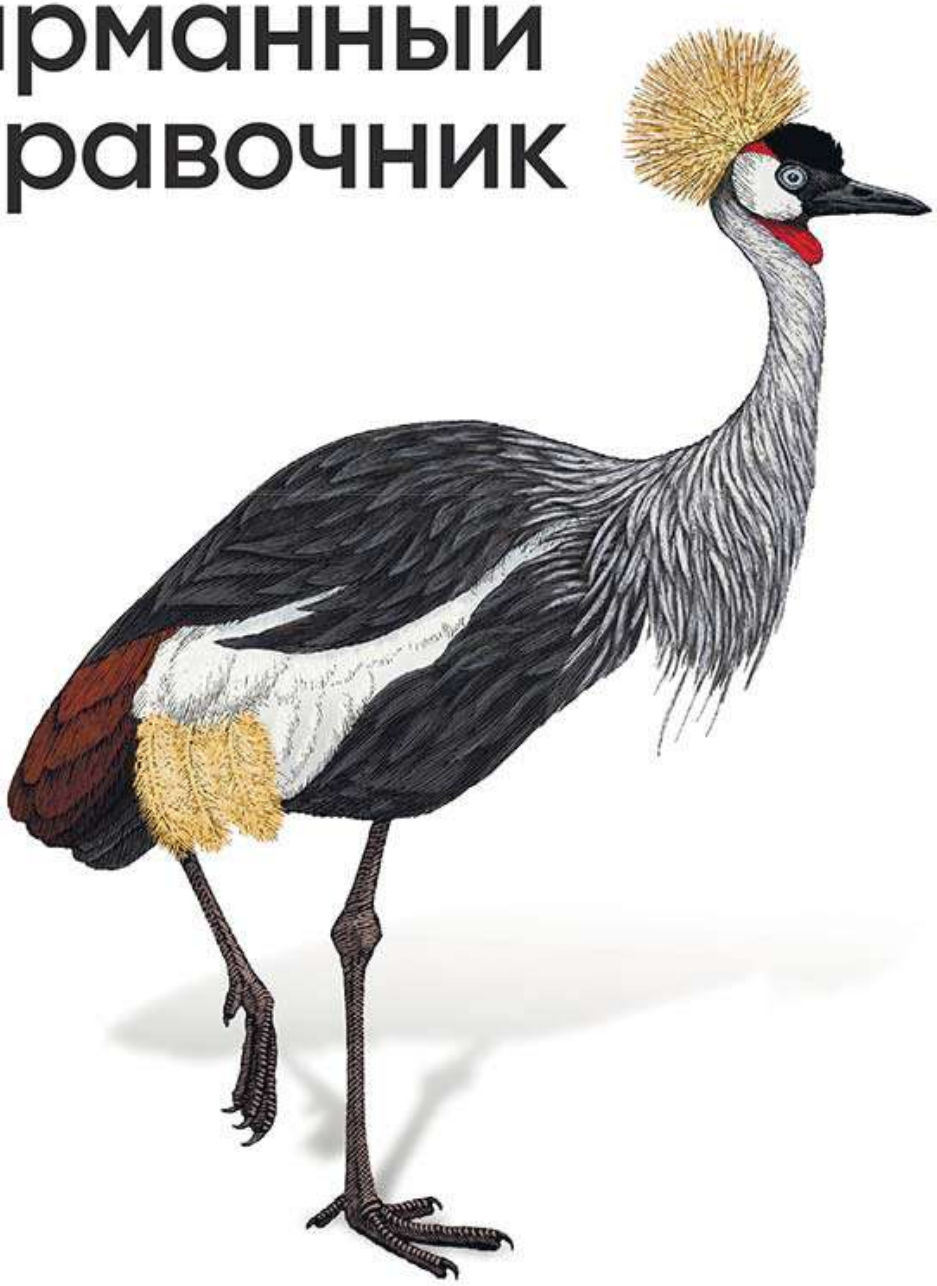
O'REILLY®

C# 9.0

Карманный справочник



Джозеф Албахари
и Бен Албахари

C# 9.0
Карманный справочник

C# 9.0 Pocket Reference

Instant Help for C# 9.0 Programmers

Joseph Albahari
and Ben Albahari

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

C# 9.0
Карманный справочник

Джозеф Албахари
и Бен Албахари

Диалектика

Москва · Санкт-Петербург
2021

ББК 32.973.26-018.2.75
А45
УДК 004.432

ООО “Диалектика”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

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

Албахари, Джозеф, **Албахари**, Бен.

А45 C# 9.0. Карманный справочник. : Пер. с англ. — СПб. : ООО “Диалектика”, 2021. — 256 с. : ил. — Парал. тит. англ.

ISBN 978-5-907365-36-0 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Джозеф Албахари, Бен Албахари

C# 9.0. Карманный справочник

Подписано в печать 30.03.2021. Формат 84×108/32
Усл. печ. л. 13,4. Уч.-изд. л. 8,8.
Тираж 500 экз. Заказ № 0000.

Отпечатано в ОАО “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 978-5-907365-36-0 (рус.)

ISBN 978-1-098-10113-8 (англ.)

Содержание

Об авторах

Джозеф Албахари — автор книг *C# 8.0 in a Nutshell* (*C# 8. Справочник. Полное описание языка*) и *C# 8.0 Pocket Reference* (*C# 8.0. Карманный справочник*), а также книги *LINQ Pocket Reference*. Разработал LINQPad — популярную утилиту для подготовки кода и проверки запросов LINQ.

Бен Албахари — бывший руководитель проектов в Microsoft, где работал над Entity Framework и .NET Compact Framework. Кроме того, соавтор книги *C# Essentials*, первой книги по языку C# от издательства O'Reilly, и предыдущих изданий *C# in a Nutshell*.

Об изображении на обложке

Животное на обложке — восточный венценосный журавль (*Balearica Regulorum*). Ареал обитания этой птицы охватывает территории Кении и Уганды на востоке и юге Африки. Эти журавли предпочитают жить на открытой болотистой местности и в лугах.

Взрослые птицы имеют рост около метра и весят около 3,5 килограмма. Визуально это яркие птицы с серым телом, бледно-серой шеей и бело-золотыми крыльями. На голове у них есть большой хохолок из жестких золотистых перьев, благодаря которому птица и получила свое название.

Венценосные журавли могут прожить в дикой природе до 20 лет, тратя большую часть времени бодрствования на хождение по траве, охоту на мелких животных и насекомых, а также на поиск семян и зерен. Это один из двух видов журавлей, которые проводят ночь на деревьях, что возможно благодаря цепкому заднему пальцу, который позволяет им хвататься за ветки. В кладке этих птиц бывает до четырех яиц; через несколько часов после вылупления птенцы способны следовать за своими родителями и кормиться самостоятельно.

Несмотря на широкий ареал обитания, в настоящее время эти птицы считаются находящимися под угрозой исчезновения. Среди основных негативных факторов называют быстрый рост населения, осушение и сельскохозяйственное использование земель и применение пестицидов. Свой вклад вносят и браконьеры, охотящиеся за яйцами этих птиц.

Обложка сделана Карен Монтгомери по мотивам черно-белой гравюры из книги *Естественная история* Касселя (1896).

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 вебсайт 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@gmail.com`

WWW: `http://www.dialektika.com`

Язык C# 9.0. Карманный справочник

C# является универсальным, безопасным в отношении типов, объектно-ориентированным языком программирования, целью которого является обеспечение продуктивности работы программистов. Для этого в нем соблюдается баланс между простотой, выразительностью и производительностью. Версия C# 9 рассчитана на работу с исполняемой средой Microsoft *.NET 5* (в то время как C# 8 ориентирован на *.NET Core 3*, а версия C# 7 — на *.NET Core 2* и Microsoft *.NET Framework 4.6/4.7/4.8*).

ПРИМЕЧАНИЕ

Программы и фрагменты кода в этой книге соответствуют примерам, рассмотренным в главах 2–4 книги *C# 9.0. Справочник. Полное описание языка* и доступным в виде интерактивных примеров в LINQPad (`http://www.linqpad.net`). Проработка примеров в сочетании с чтением настоящей книги ускоряет процесс изучения, так как вы можете редактировать код и немедленно видеть результаты без необходимости настраивать проекты и решения в среде Visual Studio.

Для загрузки примеров перейдите на вкладку Samples (Примеры) в окне LINQPad и щелкните на ссылке Download more samples (Загрузить дополнительные примеры). Утилита LINQPad бесплатна и доступна для загрузки на веб-сайте `www.linqpad.net`.

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*:

```
int x = 12 * 30;                // Инструкция 1
System.Console.WriteLine(x);    // Инструкция 2
```

Наша программа состоит из двух инструкций. Инструкции в C# выполняются последовательно и завершаются точкой с запятой. Первый оператор вычисляет *выражение* `12*30` и сохраняет результат в *переменной* с именем `x`, тип которой — 32-битное целое число (`int`). Вторая инструкция вызывает *метод* `WriteLine` *класса* с именем `Console`, который определен в *пространстве имен* `System`. Эта инструкция выводит переменную `x` в текстовое окно на экране.

Метод выполняет функцию; класс группирует функции-члены и элементы данных и образует строительный блок объектно-ориентированного программирования. Класс `Console` группирует члены, которые обрабатывают функциональность ввода-вывода в командной строке, такую, как предоставляемая методом `WriteLine`. Класс — это разновидность *типа* (о типах мы поговорим в разделе "Типы в C#").

На внешнем уровне типы организованы в *пространства имен*. Многие часто используемые типы, включая класс `Console`, находятся в пространстве имен `System`. Библиотеки `.NET` организованы во вложенные пространства имен. Например, пространство имен `System.Text` содержит типы для обработки текста, а `System.IO` — типы для ввода-вывода.

Упоминание класса `Console` с указанием пространства имен `System` при каждом применении вносит определенный беспорядок. Директива `using` позволяет избежать этого беспорядка, *импортируя* пространство имен:

```
using System;           // Импорт пространства имен System
int x = 12 * 30;
Console.WriteLine(x); // Указывать System не обязательно
```

Базовая разновидность повторного использования кода — написание функций более высокого уровня, которые вызывают функции нижнего уровня. Мы можем *рефакторизовать* нашу программу с помощью повторно используемого метода `FeetToInches`, который умножает целое значение на `12`, как показано ниже:

```
using System;
Console.WriteLine(FeetToInches(30));  // 360
Console.WriteLine(FeetToInches(100)); // 1200
```

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

Наш метод содержит ряд инструкций, окруженных парой фигурных скобок, — эта конструкция называется *блоком инструкций*.

Метод может получать *входные* данные от вызывающего метода через указанные в нем *параметры* и возвращать данные обратно вызывающей стороне с помощью указания возвращаемого типа. В нашем методе `FeetToInches` имеется входной параметр `feet` для ввода числа футов и возвращаемый тип для вывода дюймов:

```
int FeetToInches(int feet)
...
```

Литералы `30` и `100` — это *аргументы*, передаваемые в метод `FeetToInches`.

Если метод не получает входные данные, используйте пустые круглые скобки. Если он ничего не возвращает, используйте ключевое слово `void`:

```
using System;
SayHello();

void SayHello()
{
    Console.WriteLine("Hello, world");
}
```

В C# методы — одна из разновидностей функций. Другая разновидность функций, использованная в нашем примере, — это *оператор* `*`, выполняющий умножение. Кроме того, имеются *конструкторы*, *свойства*, *события*, *индексаторы* и *финализаторы*.

Компиляция

Компилятор C# компилирует исходный код (набор файлов с расширением `.cs`) в *сборку*, которая представляет собой единицу упаковки и развертывания в .NET. Сборка может быть либо *приложением*, либо *библиотекой*. Обычное консольное приложение или приложение Windows имеет *точку входа*, а библиотека — нет. Предназначение библиотеки — вызовы (*обращения*) к ней при-

ложений или других библиотек. Инфраструктура .NET 5 сама по себе является набором библиотек (а также средой выполнения).

Каждая из программ в предыдущем разделе начиналась непосредственно с ряда инструкций (называемых *инструкциями верхнего уровня*). Наличие инструкций верхнего уровня неявно создает входную точку консольного приложения или приложения Windows. (Без инструкций верхнего уровня точкой входа в приложение является метод `Main`; см. раздел "Примеры пользовательских типов".)

Для вызова компилятора можно использовать интегрированную среду разработки (IDE), такую как Visual Studio или Visual Studio Code, либо вызывать его вручную из командной строки. Чтобы вручную скомпилировать консольное приложение с .NET, сначала загрузите .NET 5 SDK, а затем создайте новый проект следующим образом:

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

Таким образом создается папка `MyFirstProgram`, которая содержит исходный файл C# с именем `Program.cs`, который затем можно редактировать. Для вызова компилятора вызовите `dotnet build` (или `dotnet run` — эта команда скомпилирует, а затем запустит программу). Вывод компилятора будет записан в подкаталог `bin\debug`, где будут находиться `MyFirstProgram.dll` (выходная сборка), а также файл `MyFirstProgram.exe` (который непосредственно выполняет скомпилированную программу).

Синтаксис

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

```
using System;
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 из примера программы в порядке их появления:

```
System    x    Console    WriteLine
```

Идентификатор должен быть единым словом, состоящим из символов Unicode и начинающимся с буквы или подчеркивания. Идентификаторы C# чувствительны к регистру. По соглашению параметры, локальные переменные и закрытые поля должны именоваться с использованием *верблюжьего стиля* (например, `myVariable`), а все остальные идентификаторы должны быть в *стиле Pascal* (например, `MyMethod`).

Ключевые слова — это имена, которые для компилятора означают что-то особенное. В нашем примере программы есть два ключевых слова, `using` и `int`.

Большинство ключевых слов *зарезервированы*, что означает, что вы не можете использовать их как идентификаторы. 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 использовать идентификатор с именем, которое конфликтует с ключевым словом, то добавьте к нему префикс @. Например:

```
class class  {...} // Запрещено
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*, что означает, что их можно использовать и в качестве идентификаторов без символа @. Такие ключевые слова перечислены ниже:

```
add             equals          nameof          set
alias           from            not             unmanaged
and             get             on              value
ascending       global          or              var
async           group           orderby         with
await           in              partial         when
by              into            record          where
descending      join            remove          yield
dynamic         let             select
```

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

Литералы, знаки пунктуации и операторы

Литералы — это элементарные порции данных, лексически встраиваемые в программу. В рассматриваемом примере программы используются литералы `12` и `30`. *Знаки пунктуации* помогают размечать структуру программы. Примером может служить точка с запятой, которая завершает инструкцию. Инструкции могут охватывать несколько строк:

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

Оператор преобразует и объединяет выражения. Большинство операторов в C# обозначаются с помощью некоторого сим-

вола, например оператор умножения имеет следующий вид: *. Вот операторы 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 скобки означают, что метод не принимает аргументов. Знак "равно" выполняет присваивание (двойной знак "равно", ==, производит сравнение на равенство).

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 о присваивании
           // переменной x значения 3
```

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

```
int x = 3; /* Комментарий о присваивании
              переменной x значения 3 */
```

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

Типы в C#

Тип определяет шаблон значения. В рассматриваемом примере мы использовали два литерала типа `int` со значениями `12` и `30`. Мы также объявляем переменную типа `int` с именем `x`.

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

Все значения в C# являются *экземплярами* определенного 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 разные операции, например, арифметические:

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

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

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine(upperMessage);   // HELLO WORLD

int x = 2021;
message = message + x.ToString();
Console.WriteLine(message);        // Hello world2021
```

Предопределенный тип `bool` поддерживает ровно два возможных значения: `true` и `false`. Тип `bool` обычно используется для разветвления потока выполнения по условию с помощью инструкции `if`. Например:

```
bool simpleVar = false;
if (simpleVar)
    Console.WriteLine("Этот текст не выводится");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
    Console.WriteLine("Этот текст выводится");
```

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

Примеры пользовательских типов

Точно так же, как из простых функций можно строить сложные функции, из элементарных типов можно создавать сложные

типы. В следующем примере мы определим пользовательский тип по имени `UnitConverter` — класс, который служит шаблоном для преобразования единиц:

```
UnitConverter feetToInches = new UnitConverter (12);
UnitConverter milesToFeet = new UnitConverter (5280);

Console.WriteLine(feetToInches.Convert(30));  //   360
Console.WriteLine(feetToInches.Convert(100)); //  1200
Console.WriteLine(feetToInches.Convert
                  (milesToFeet.Convert(1)));  // 63360

public class UnitConverter
{
    int ratio;                              // Поле
    public UnitConverter(int unitRatio) // Конструктор
    {
        ratio = unitRatio;
    }
    public int Convert (int unit)       // Метод
    {
        return unit * ratio;
    }
}
```

Члены типа

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

Симметрия предопределенных и пользовательских типов

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

Конструкторы и создание экземпляров

Данные создаются путем создания экземпляров (*инстанцирования*) типа. Мы можем создавать экземпляры предопределенных типов просто путем применения литерала, такого как `12` или `"Привет"`.

Оператор `new` создает экземпляры пользовательского типа. Мы начинаем нашу программу с создания двух экземпляров типа `UnitConverter`. Непосредственно после создания объекта оператором `new` вызывается его *конструктор* для выполнения инициализации. Конструктор определяется так же, как и метод, 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 *экземплярами* 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 имеют дело не с конкретным экземпляром типа, а с самим типом, должны помечаться как статические (`static`). Чтобы обратиться к статическому члену извне его типа, следует указывать имя его *типа*, а не экземпляр. Примером является метод `WriteLine` класса `Console`. Поскольку это статический метод, мы вызываем его как `Console.WriteLine()`, а не как `new Console().WriteLine()`.

В приведенном далее коде поле экземпляра `Name` относится к конкретному экземпляру `Panda`, в то время как поле `Population` принадлежит всему множеству экземпляров класса `Panda`. Мы создаем два экземпляра `Panda`, выводим их имена, а затем общую численность населения:

```
Panda p1 = new Panda("Pan Dee");
Panda p2 = new Panda("Pan Dah");

Console.WriteLine(p1.Name);          // Pan Dee
```

```
Console.WriteLine(p2.Name);           // Pan Dah
Console.WriteLine(Panda.Population); // 2

public class Panda
{
    public string Name;           // Поле экземпляра
    public static int Population; // Статическое поле
    public Panda (string n)       // Конструктор
    {
        Name = n;                 // Поле экземпляра
        Population = Population+1;// Статическое поле
    }
}
```

Попытки вычисления `p1.Population` или `Panda.Name` приводят к генерации ошибки времени компиляции.

Ключевое слово `public`

Ключевое слово `public` открывает доступ к членам для других классов. Если бы в рассматриваемом примере поле `Name` класса `Panda` не было помечено как `public`, то оно оказалось бы закрытым и класс `Console` не смог бы получить к нему доступ. Маркировка члена как открытого (`public`) означает, что тип разрешает его видеть всем 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.

Создание пространства имен

Особенно в случае больших программ имеет смысл организация типов в пространства имен. Вот как определяется класс `Panda` внутри пространства имен `Animals`:

```
namespace Animals
{
    public class Panda
    {
        ...
    }
}
```

Детально пространства имен будут рассмотрены позже, в разделе “Пространства имен”.

Определение метода Main

До сих пор во всех наших примерах использовались инструкции верхнего уровня (которые являются новой функциональной возможностью в C# 9). Без инструкций верхнего уровня простое консольное приложение или приложение Windows выглядит следующим образом:

```
using System;

class Program
{
    static void Main() // Входная точка программы
    {
        int x = 12 * 30;
        Console.WriteLine(x);
    }
}
```

В отсутствие инструкций верхнего уровня C# ищет статический метод с именем `Main`, который и становится точкой входа. Метод `Main` может быть определен внутри любого класса (при этом может существовать только один метод `Main`). Если вашему методу `Main` требуется доступ к закрытым членам некоторого класса, определите метод `Main` внутри этого класса — это может быть проще, чем использование инструкций верхнего уровня.

Метод `Main` может (необязательно) возвращать целое число (а не `void`), чтобы вернуть значение среде выполнения (где ненулевое значение обычно указывает на ошибку). Метод `Main` может также дополнительно принимать в качестве параметра массив строк (который будет заполнен аргументами, передаваемыми исполняемому файлу). Например:

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

ПРИМЕЧАНИЕ

Массив (такой, как `string[]`) представляет фиксированное количество элементов определенного типа. Массивы указываются с помощью квадратных скобок после типа элемента. Мы рассмотрим их позже, в разделе “Массивы”.

(Метод `Main` может также быть объявлен как `async` и возвращать `Task` или `Task<int>` для поддержки асинхронного программирования; см. раздел "Асинхронные функции".)

Инструкции верхнего уровня (C# 9)

Инструкции верхнего уровня C# 9 позволяют избежать необходимости статического метода `Main` и содержащего его класса. Файл с инструкциями верхнего уровня состоит из трех частей в таком порядке.

1. (Необязательная) директива `using`.
2. Ряд инструкций, возможно, перемешанных с объявлениями методов.
3. (Необязательные) объявления типов и пространств имен.

Все содержимое части 2 в конечном итоге оказывается внутри созданного компилятором метода `Main` класса, созданного компилятором. Это означает, что методы в ваших инструкциях верхнего уровня становятся *локальными методами* (об их тонкостях мы поговорим позже, в отдельном разделе, посвященном локальным методам). Инструкции верхнего уровня могут дополнительно возвращать целочисленное значение вызывающему коду и получать доступ к "магической" переменной `args` типа `string[]`, соответствующей переданным программе аргументам командной строки.

Поскольку программа может иметь только одну точку входа, в проекте C# может быть только один файл с инструкциями верхнего уровня.

Типы и 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 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` (имеющему в два раза меньше битов, чем `int`):

```
int   x = 12345;    // int — 32-битное целое
long  y = x;        // Неявное преобразование в 64 бита
short z = (short)x; // Явное преобразование в 16 бит
```

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

Типы-значения и ссылочные типы

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

Типы-значения включают большинство встроенных типов (в частности, все числовые типы, тип `char` и тип `bool`), а также пользовательские типы `struct` и `enum`. *Ссылочные типы* включают все классы, массивы, делегаты и интерфейсы.

Фундаментальное различие между типами-значениями и ссылочными типами связано с тем, как они поддерживаются в памяти.

Типы-значения

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

С помощью ключевого слова `struct` можно определить пользовательский тип-значение (рис. 1):

Рис. 1. *Экземпляр типа-значения в памяти*

Присваивание экземпляра типа-значения всегда *копирует* экземпляр. Например:

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

Point p2 = p1;   // Присваивание вызывает копирование
```

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

p1.X = 9;                 // Изменение p1.X

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

На рис. 2 продемонстрировано, что `p1` и `p2` имеют независимые хранилища.

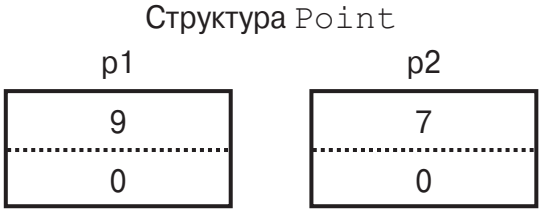


Рис. 2. *Присваивание копирует экземпляр типа-значения*

Ссылочные типы

Ссылочный тип сложнее типа-значения из-за наличия двух частей: самого *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; }
```

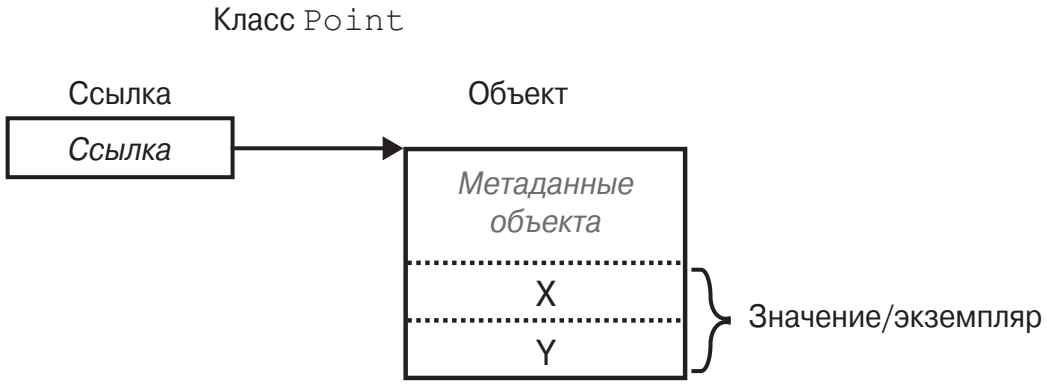


Рис. 3. *Экземпляр ссылочного типа в памяти*

Присваивание переменной ссылочного типа вызывает копирование ссылки, но не экземпляра объекта. Это позволяет множеству переменных ссылаться на один и тот же объект, что с типами-зна-

чениями обычно невозможно. Если повторить предыдущий пример при условии, что `Point` теперь представляет собой класс, то операции с `p1` будут воздействовать и на `p2`:

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

Point p2 = p1;   // Копирование ссылки p1

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

p1.X = 9;        // Изменение 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.

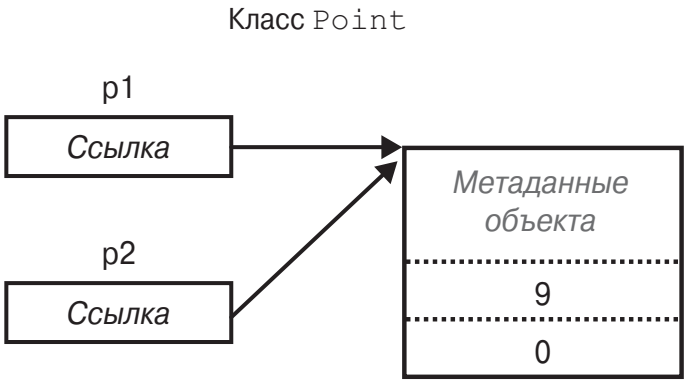


Рис. 4. *Экземпляр ссылочного типа в памяти*

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 нулевой ссылки приводит к ошибке времени выполнения:

```
Console.WriteLine(p.X); // Исключение
NullReferenceException
```

ПРИМЕЧАНИЕ

В разделе “Ссылочные типы, допускающие значение null” описывается функциональная возможность C#, которая уменьшает количество случайных ошибок `NullReferenceException`.

В противоположность этому тип-значение обычно не может иметь значение `null`:

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

Для обхода этого ограничения в C# есть специальная конструкция для представления значения `null`; см. раздел “Ссылочные типы, допускающие значение null”.

Классификация предопределенных типов

Предопределенные типы в C# классифицируются следующим образом.

Типы-значения

- ✓ Числовые
  - ♦ Целочисленные со знаком (`sbyte`, `short`, `int`, `long`)
  - ♦ Целочисленные без знака (`byte`, `ushort`, `uint`, `ulong`)
  - ♦ Действительные (`float`, `double`, `decimal`)
- ✓ Логический (`bool`)
- ✓ Символьный (`char`)

Ссылочные типы

- ✓ Строки (`string`)
- ✓ Объекты (`object`)

Предопределенные типы в C# являются псевдонимами типов .NET в пространстве имен `System`. Две показанные ниже инструкции различаются только синтаксисом:

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

Множество предопределенных типов-*значений*, исключая `decimal`, в общеязыковой исполняющей среде (Common Language Runtime — CLR) известно как *примитивные типы*. Примитивные типы называются так потому, что они поддерживаются непосредственно командами в скомпилированном коде, которые обычно транслируются в непосредственную поддержку процессором.

Числовые типы

В C# имеются перечисленные в табл. 1 предопределенные числовые типы.

Таблица 1. Предопределенные числовые типы C#

Тип	Тип System	Суффикс	Размер, битов	Диапазон
		Целочисленный знаковый		
`sbyte`	`SByte`		8	От -2^7 до 2^7-1
`short`	`Int16`		16	От -2^{15} до $2^{15}-1$
`int`	`Int32`		32	От -2^{31} до $2^{31}-1$
`long`	`Int64`	`L`	64	От -2^{63} до $2^{63}-1$
		Целочисленный беззнаковый		
`byte`	`Byte`		8	От 0 до 2^8-1
`ushort`	`UInt16`		16	От 0 до $2^{16}-1$
`uint`	`UInt32`	`U`	32	От 0 до $2^{32}-1$
`ulong`	`UInt64`	`UL`	64	От 0 до $2^{64}-1$
		Действительный		
`float`	`Single`	`F`	32	$\pm(\approx$ от 10^{-45} до $10^{38})$
`double`	`Double`	`D`	64	$\pm(\approx$ от 10^{-324} до $10^{308})$
`decimal`	`Decimal`	`M`	128	$\pm(\approx$ от 10^{-28} до $10^{28})$

Из всех *целочисленных* типов `int` и `long` являются *первоклассными* типами, которым обеспечиваются предпочтение и поддержка как языком C#, так и средой выполнения. Другие целочисленные типы обычно применяются для реализации взаимодействия или когда на первое место выходят эффективность хранения и экономия памяти.

ПРИМЕЧАНИЕ

В C# 9 имеются два новых целочисленных типа для представления *целых чисел платформы* (native): `nint` (со знаком) и `nuint` (без знака). Эти типы предназначены для взаимодействия в рамках платформы и отображаются во время выполнения в `System.IntPtr` и `System.UIntPtr` соответственно. Во время компиляции они обеспечивают дополнительную функциональность, в первую очередь поддержку стандартных числовых операций.

Среди *действительных* 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 требуются десятичная арифметика и высокая точность. (Технически `decimal` также является типом с плавающей точкой, хотя обычно о нем так не говорят.)

Числовые литералы

Целочисленные литералы могут использовать десятичную, шестнадцатеричную или бинарную форму записи; шестнадцатеричная форма записи предусматривает применение префикса `0x` (например, `0x7f` эквивалентно десятичному значению `127`), а бинарная — префикс `0b`. В *действительных литералах* может применяться десятичная или экспоненциальная форма записи, такая как `1E06`. Для улучшения читаемости в числовой литерал могут вставляться символы подчеркивания (например, `1_000_000`).

Вывод 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 табл. 1, 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 int -> long
```

Суффикс `D` технически избыточен, поскольку все литералы с десятичной точкой выводятся как `double` (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 приведенный далее код не скомпилируется, так как литерал `4.5` выводится как тип `double`, для которого не предусмотрено неявное преобразование в `float` или `decimal`:

```
float   f =   4.5F; // Без суффикса не компилируется
decimal d = -1.23M; // Без суффикса не компилируется
```

Числовые преобразования

Преобразования целых чисел в целые числа

Целочисленные преобразования являются *неявными*, когда целевой тип в состоянии представить любое возможное значение исходного типа; в противном случае требуется *явное* преобразование. Например:

```
int x = 12345;       // int — 32-битный целочисленный тип
long y = x;          // Неявное преобразование
                     // в 64-битный int
short z = (short)x;  // Явное преобразование в 16-битный int
```

Преобразования чисел с плавающей точкой в числа с плавающей точкой

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

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

Преобразования чисел с плавающей точкой в целые числа

Преобразования целочисленных типов в действительные 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 - нет
int i2 = (int)f; // 100000000
```

Арифметические операторы

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

Операторы инкремента и декремента

Операторы инкремента и декремента (соответственно `++` и `--`) увеличивают и уменьшают значения числовых типов на `1`. Такой оператор 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 равно 1
Console.WriteLine(++x); // Выводит 2; x теперь равно 2
Console.WriteLine(--x); // Выводит 1; x теперь равно 1
```

Специализированные целочисленные операции

Деление

Операции деления для целочисленных типов всегда усекают остатки (округление в направлении нуля). Деление на переменную, значение которой равно нулю, приводит к ошибке времени выполнения (исключению `DivideByZeroException`). Деление на литерал или константу, равную нулю, вызывает ошибку времени компиляции.

Переполнение

Выполнение арифметических операций с целочисленными типами может приводить к переполнению. По умолчанию это происходит без информирования о происшедшем — никакие исключения не генерируются, а результат демонстрирует поведение с циклическим возвратом, как если бы вычисление производилось над бо́льшим целочисленным типом с отбрасыванием дополнительных значащих битов. Например, декремент минимально возможного значения типа `int` дает в результате максимально возможное значение `int`:

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

Операторы `checked` и `unchecked`

Оператор `checked` сообщает среде выполнения о том, что вместо "молчаливого" переполнения она должна генерировать исключение `OverflowException`, когда целочисленное выражение или инструкция выходит за арифметические пределы данного типа. Оператор `checked` воздействует на выражения с операторами `++`, `--`, `-` (унарный), `+`, `-`, `*`, `/` и операторами явного преобразования между целочисленными типами. С проверкой переполнения связаны небольшие накладные расходы, влияющие на производительность.

Оператор `checked` можно использовать либо с выражением, либо с блоком инструкций. Например:

```
int a = 1000000, b = 1000000;
int c = checked(a*b); // Проверка только данного выражения
checked               // Проверка всех выражений
{                     // в блоке инструкций
```

```
    c = a * b;
    ...
}
```

Проверку на арифметическое переполнение можно сделать обязательной для всех выражений в программе, скомпилировав ее с аргументом 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, можно применить оператор `unchecked`.

Побитовые операторы

В C# поддерживаются указанные в табл. 2 побитовые операторы.

Таблица 2. Побитовые операторы C#

Оператор	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# при необходимости они неявно преобразуются в более крупные типы. Попытка присваивания результата переменной меньшего целочисленного типа может вызвать ошибку времени компиляции:

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

В данном случае для выполнения сложения переменные `x` и `y` неявно преобразуются в тип `int`. Это означает, что результат так-

же будет иметь тип `int`, который не может быть неявно приведен к типу `short` (из-за возможной потери информации). Чтобы приведенный выше код скомпилировался, необходимо добавить явное приведение:

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

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 нуль 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 нуль 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 оператор `==`, 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. В табл. 3 представлена сводка по отличиям между типами `double` и `decimal`.

Таблица 3. Типы `double` и `decimal`

Характеристика	`double`	`decimal`
Внутреннее представление	Двоичное	Десятичное
Точность	15–16 значащих цифр	28–29 значащих цифр
Диапазон	±(от $\approx 10^{-324}$ до $\approx 10^{308}$)	±(от $\approx 10^{-28}$ до $\approx 10^{28}$)
Специальные значения	+0, –0, +∞, –∞ и NaN	Нет
Скорость обработки	Внутрипроцессорная	Код; примерно в 10 раз медленнее `double`

Ошибки округления вещественных чисел

Типы `float` и `double` внутренне представляют собой числа в бинарном виде. По этой причине большинство литералов с дробной частью (которые являются десятичными) не могут быть представлены точно, что делает их непригодными для финансовых операций.

В противоположность им тип `decimal` работает в десятичной системе счисления и способен точно представлять дробные числа наподобие 0,1 (десятичное представление которого является конечным).

Логический тип и операторы

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

Хотя для хранения логического значения достаточно только одного бита, исполняющая среда будет использовать один байт памяти, поскольку это минимальное адресуемое количество памя-

ти, с которой могут эффективно работать исполняющая среда и процессор.

Во избежание непродуктивных затрат памяти в случае массивов инфраструктура .NET предлагает в пространстве имен `System.Collections` класс `BitArray`, который позволяет задействовать по одному биту для каждого булева значения в массиве.

Операторы эквивалентности и сравнения

Операторы `==` и `!=` проверяют эквивалентность и неэквивалентность значений любого типа и всегда возвращают значение типа `bool`. Типы-значения 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 основана на *ссылке*, а не на действительном *значении* объекта. Следовательно, два экземпляра объекта с идентичными данными не будут считаться равными, если только оператор `==` специально не был перегружен для достижения такого эффекта (см. разделы "Тип object" и "Перегрузка операторов").

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

Условные операторы

Операторы && и || реализуют условия *И* и *ИЛИ*. Они часто применяются в сочетании с оператором `!`, который выражает отрицание — *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 это возможно, операторы && и || используют *сокращенное вычисление*. В предыдущем примере, если дует ветер (windy), то выражение (rainy||sunny) даже не вычисляется (его значение не влияет на очевидный общий результат). Сокращенные вычисления играют важную роль, разрешая выражения, такие как показанное ниже, без генерации исключения NullReferenceException:

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

Операторы & и | также реализуют проверки условий *И* и *ИЛИ*:

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

Их отличие от рассмотренных выше операторов в том, что они не используют сокращенные вычисления. По этой причине операторы & и | редко используются вместо условных операторов.

Тернарный условный оператор (именуемый просто условным оператором) имеет вид q?a:b, где, если условие q истинно, вычисляется a, и b — в противном случае. Например:

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

Условный оператор особенно удобен 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 значением, например:

```
char newLine   = '\n';  // Символ новой строки
char backSlash = '\\';  // Обратная косая черта
```

Управляющие последовательности перечислены в табл. 4.

Таблица 4. Управляющие последовательности

Последовательность	Описание	Значение
\'	Одинарная кавычка	0x0027
\"	Двойные кавычки	0x0022
\\	Обратная косая черта	0x005C
\0	Нулевой символ	0x0000
\a	Звуковой сигнал	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 тех числовых типов, которые могут вместить беззнаковый `short`. 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";
```

ПРИМЕЧАНИЕ

`string` — тип ссылочный, а не тип-значение. Тем не менее его операторы эквивалентности следуют семантике типов-значений:

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

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

```
string a = "Это табуляция:\t";
```

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

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

Чтобы избежать этого, 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 требуется записать дважды.

Конкатенация строк

Оператор + 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 оператора + для построения строки может оказаться неэффективным; более удачное решение предусматривает использование типа `System.Text.StringBuilder`, который представляет изменяемую (редактируемую) строку и рас-

полагает методами эффективного добавления, вставки, удаления и замены подстрок — `Append`, `Insert`, `Remove` и `Replace` соответственно.

Интерполяция строк

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

```
int x = 4;
Console.Write($"У квадрата {x} стороны");
// Вывод: У квадрата 4 стороны
```

Внутри фигурных скобок может быть указано любое допустимое выражение C# произвольного типа; компилятор C# преобразует это выражение в строку, вызывая метод `ToString()` или эквивалентный метод типа выражения. Форматирование можно изменять путем добавления к выражению двоеточия и строки формата (строки формата рассматриваются в главе 6 книги *C# 9.0. Справочник. Полное описание языка*):

```
string s = $"{15:X2} - это 15 в шестнадцатеричной записи";
// Дает "0F - это 15 в шестнадцатеричной записи"
```

Интерполированные строки должны находиться в одной строке кода, если только вы не указываете оператор дословной строки. Обратите внимание, что оператор $ должен располагаться перед @:

```
int x = 2;
string s = $@"Это целых {
x} строки";
```

Чтобы включить в интерполированную строку литерал фигурной скобки, его следует удвоить.

Сравнения строк

Тип `string` не поддерживает операторы < и > для сравнения строк. Вместо них должен применяться метод `CompareTo()` типа `string`, который возвращает положительное или отрицательное число или нуль в зависимости от того, находится ли лексикографически первое значение после второго, до него или они совпадают:

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

Поиск в строках

Индексатор для `string` возвращает символ в указанной позиции:

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

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

Манипулирование строками

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

- ✓ 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 5 символов:

```
char[] vowels = new char[5];
```

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

```
vowels[0] = 'a'; vowels[1] = 'e'; vowels[2] = 'i';
vowels[3] = 'o'; vowels[4] = 'u';
Console.WriteLine(vowels[1]);   // e
```

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

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

Массивы также реализуют интерфейс `IEnumerable<T>` (см. раздел "Перечисление и итераторы"), так элементы массива можно обойти с помощью инструкции цикла `foreach`:

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

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

```
vowels[5] = 'y';  // 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 единой инструкцией:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

или проще:

```
char[] vowels = {'a','e','i','o','u'};
```

Все массивы унаследованы от класса `System.Array`, в котором определены общие методы и свойства для всех массивов. Сюда входят свойства экземпляра наподобие `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 отсортированном (`BinarySearch`) или неотсортированном (`IndexOf`, `LastIndexOf`, `Find`, `FindIndex`, `FindLastIndex`) массиве;
- ✓ сортировка массива (`Sort`);
- ✓ копирование массива (`Copy`).

Инициализация элементов по умолчанию

При создании массива всегда происходит инициализация его элементов значениями по умолчанию. Значение по умолчанию для типа представляет собой результат побитового обнуления памяти. Например, предположим, что создается массив целых чисел. Поскольку типом значения является `int`, выделится пространство под 1000 целочисленных значений в виде одного непрерывного блока памяти. Значением по умолчанию для каждого элемента будет 0:

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

Для элементов ссылочного типа значением по умолчанию является `null`.

Независимо от типа элементов *сам* массив всегда является объектом ссылочного типа. Например, следующая инструкция вполне допустима:

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

Индексы и диапазоны

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

ПРИМЕЧАНИЕ

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, путем определения индексатора типа `Index` или `Range` (см. раздел "Индексаторы").

Индексы

Индексы позволяют обращаться к элементам относительно конца массива с помощью оператора `^`. Конструкция `^1` обращается к последнему элементу, `^2` — ко второму с конца и т.д.:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char lastElement  = vowels[^1];    // 'u'
char secondToLast = vowels[^2];    // 'o'
```

(`^0` соответствует длине массива, поэтому `vowels[^0]` приводит к ошибке.)

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

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

Диапазоны

Диапазоны дают возможность "нарезать" массив с помощью оператора `..`:

```
char[] firstTwo  = vowels[..2];  // 'a', 'e'
char[] lastThree = vowels[2..];  // 'i', 'o', 'u'
char[] middleOne = vowels[2..3]; // 'i'
```

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]; // 'o', 'u'
```

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

Диапазоны реализованы в C# с помощью типа `Range`, поэтому можно также писать следующим образом:

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

Многомерные массивы

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

Прямоугольные массивы

Прямоугольные массивы объявляются с использованием запятых для отделения каждого измерения одно от другого. Ниже приведено объявление прямоугольного двумерного массива размером 3×3:

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

Метод `GetLength()` массива возвращает длину 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 массиву из предыдущего примера):

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

(Код, выделенный полужирным шрифтом, 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 равно 3:

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

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

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int[3]; // Создание внутреннего массива
  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 массиву из предыдущего примера, 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}
};
```

(Код, выделенный полужирным шрифтом, 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[] {'a','e','i','o','u'};
char[] vowels =            {'a','e','i','o','u'};
```

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":

```
Foo(new char[] {'a','e','i','o','u'} ); // Длинно
Foo(new[]      {'a','e','i','o','u'} ); // Коротко
```

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, ссылке, входным и выходным), *полем* (экземпляра или статическим) или *элементом массива*.

Стек и куча

Стек и куча — это места, где располагаются переменные. Стек и куча имеют существенно различающуюся 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);
}
```

Этот метод 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, память, выделенная этому экземпляру `int`, освобождается.

Куча

Куча — это память, в которой располагаются *объекты* (т.е. экземпляры ссылочного типа). Всякий раз, когда создается новый объект, он размещается в куче и на него возвращается ссылка. Во время выполнения программы по мере создания новых объектов

куча начинает заполняться. 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.

ПРИМЕЧАНИЕ

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

В куче также хранятся статические поля и константы. В отличие от объектов, память для которых выделена в куче (и 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` получить доступ к неинициализированной памяти невозможно. Определенное присваивание приводит к трем следствиям.

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

Например, следующий код вызывает ошибку времени компиляции:

```
int x;                // x - локальная переменная
Console.WriteLine(x); // Ошибка времени компиляции
```

Однако следующий код выводит 0, потому что поля неявно получают значения по умолчанию (будь то поле экземпляра или статическое):

```
Console.WriteLine(Test.X);          // 0
class Test { public static int X; } // Поле
```

Значения по умолчанию

Экземпляры всех типов имеют значения по умолчанию. Значения по умолчанию 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 значение по умолчанию для любого типа можно с использованием ключевого слова `default` (как вы увидите позже, на практике поступать так удобно при работе с обобщениями). Значение по умолчанию в пользовательском типе-значении (т.е. `struct`) — это то же самое, что и значения по умолчанию для всех полей, определенных данным пользовательским типом.

```
Console.WriteLine(default(decimal));   // 0
decimal d = default;
```

Параметры

Метод может принимать последовательность параметров. Параметры определяют набор аргументов, которые должны быть предоставлены этому методу. В следующем примере метод `Foo()` имеет единственный параметр с именем `p` типа `int`:

```
Foo (8);                        // 8 – аргумент
static void Foo (int p) {...} // p – параметр
```

Управлять способом передачи параметров можно с помощью модификаторов `ref`, `out` и `in` (табл. 5).

Таблица 5. Способы передачи параметров

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, несомненно, является самым распространенным случаем. Это означает, что при передаче методу создается копия значения:

```
int x = 8;
Foo(x);                  // Создание копии x
Console.WriteLine(x);    // x остается равным 8

static void Foo (int p)
{
    p = p + 1;           // Увеличение p на 1
    Console.WriteLine(p); // Вывод p на экран
}
```

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 инстанцирован (`sb`), 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 sb = new StringBuilder();
Foo(sb);
Console.WriteLine(sb.ToString());   // test

static void Foo(StringBuilder fooSB)
{
    fooSB.Append("test");
    fooSB = null;
}
```

Поскольку fooSB — *копия* ссылки, установка ее равной null не приводит к установке в null переменной sb. (Однако если параметр fooSB объявить и вызывать с модификатором ref, то sb станет равной 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:

```
int x = 8;
Foo(ref x);           // Foo работает непосредственно с x
Console.WriteLine(x); // Сейчас x равно 9

static void Foo(ref int p)
{
    p = p + 1;            // Увеличение p на 1
    Console.WriteLine(p); // Вывод p на экран
}
```

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.

ПРИМЕЧАНИЕ

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

Модификатор out

Аргумент out схож с аргументом ref, за исключением следующих аспектов:

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

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

Переменные `out` и их отбрасывание

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

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

Приведенный выше код эквивалентен следующему:

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

Когда вызываются методы с множеством параметров `out`, с помощью символа подчеркивания можно "отбрасывать" любые параметры, которые не интересуют вызывающий код. Предполагая, что метод `SomeBigMethod()` был определен с пятью параметрами `out`, вот как можно 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 предварять модификатором `in`, чтобы гарантировать его неизменность внутри метода. В результате у компилятора появляется возможность избежать накладных расходов по созданию копии аргумента для его передачи, которые могут оказаться существенными в случае крупных пользовательских типов значений (см. раздел "Структуры").

Модификатор `params`

Модификатор `params`, когда он применяется к последнему параметру метода, позволяет методу принимать любое количество аргументов определенного типа. Тип такого 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 значение по умолчанию:

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

При вызове метода необязательные параметры могут быть опущены:

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

В действительности в качестве необязательного параметра `x` передается аргумент со значением по умолчанию `23` — компилятор встраивает это значение в скомпилированный вызывающий код. Показанный выше вызов `Foo()` семантически эквивалентен вызову

```
Foo(23);
```

потому что компилятор просто подставляет значение по умолчанию необязательного параметра.

ПРИМЕЧАНИЕ

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

Значение по умолчанию необязательного параметра должно быть указано в виде константного выражения или конструктора без параметров для типа-значения. Необязательные параметры не могут быть помечены модификатором `ref` или `out`.

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

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

Чтобы сделать иначе (поменять местами значение по умолчанию для `x` и указанное явно значение для `y`), требуется скомбинировать необязательные параметры с *именованными аргументами*.

Именованные аргументы

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

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

void Foo(int x, int y)
{
    Console.WriteLine(x + ", " + y);
}
```

Именованные аргументы могут находиться 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 особенно удобно при работе с API COM.

Неявно типизированные локальные переменные

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

Это в точности эквивалентно следующему коду:

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

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

```
var x = 5;
x = "hello"; // Ошибка времени компиляции; x имеет тип int
```

В разделе "Анонимные типы" мы опишем сценарий, в котором использование ключевого слова `var` обязательно.

Контекстное определение типа выражения new

Еще одно средство уменьшить лексическое повторение — это появившаяся в C# 9 возможность контекстного определения типа выражения `new` (*target-typed new expressions*):

```
StringBuilder sb1 = new();
StringBuilder sb2 = new("Test");
```

Этот код в точности эквивалентен следующему:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("Test");
```

Принцип заключается в том, что вы можете вызвать `new` без указания имени типа, если компилятор может однозначно вывести этот тип. Такие выражения `new` особенно полезны, когда объявление переменной и инициализация находятся в разных частях

вашего кода. Типичный пример — когда вы хотите инициализировать поле в конструкторе:

```
class Foo
{
    System.Text.StringBuilder sb;
    public Foo (string initialValue)
    {
        sb = new (initialValue);
    }
}
```

Эта функциональная возможность полезна и в следующей ситуации:

```
MyMethod(new ("test"));
void MyMethod(System.Text.StringBuilder sb) { ... }
```

Выражения и операторы

Выражение, по сути, описывает значение. Простейшими разновидностями выражений являются константы (наподобие `123`) и переменные (такие, как `x`). Выражения могут видоизменяться и комбинироваться с помощью операторов. *Оператор* принимает один или несколько входных *операндов* и дает на выходе новое выражение:

```
12 * 30 // * - оператор; 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)
```

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

Операторы, которые являются неотъемлемой частью самого языка, называются *основными*, или *первичными* (primary); примером может служить оператор вызова метода. Выражение, не имеющее значения, называется *пустым выражением* (void expression):

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

Поскольку пустое выражение не имеет значения, оно не может использоваться в качестве операнда при построении более сложных выражений:

```
1 + Console.WriteLine(1) // Ошибка времени компиляции
```

Выражения присваивания

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

```
x = x * 5
```

Выражение присваивания не является пустым. На самом деле оно имеет значение, равное присваиваемому, а потому может встраиваться в другое выражение. В следующем примере выражение присваивает значение `2` переменной `x`, и значение `10` — переменной `y`:

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

Такой стиль выражения может использоваться для инициализации нескольких значений:

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

Составные операторы присваивания представляют собой синтаксическое сокращение, которое объединяет присваивание с другим оператором. Например:

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

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

Приоритеты и ассоциативность операторов

Когда выражение содержит несколько операторов, порядок их вычисления определяется *приоритетами* и *ассоциативностью*. Операторы с более высокими приоритетами выполняются перед

операторами, приоритеты которых ниже. Если операторы имеют одинаковые приоритеты, то порядок их выполнения определяется ассоциативностью.

Приоритеты операторов

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

Левоассоциативные операторы

Бинарные операторы (кроме оператора присваивания, лямбд и операторов объединения с `null`) являются *левоассоциативными*; другими словами, они вычисляются слева направо. Например, выражение `8/4/2` вычисляется как `(8/4)/2` по причине левоассоциативности оператора деления. Разумеется, порядок вычисления можно изменить, расставив скобки.

Правоассоциативные операторы

Оператор присваивания, лямбда, оператор объединения с `null` и условный оператор являются *правоассоциативными*; другими словами, они вычисляются справа налево. Правая ассоциативность обеспечивает возможность множественного присваивания, такого как `x=y=3`: сначала значение `3` присваивается переменной `y`, а затем результат этого выражения (`3`) присваивается переменной `x`.

Таблица операторов

В табл. 6 перечислены операторы C# в порядке их приоритета. О перегрузке операторов будет рассказано позже, в разделе "Перегрузка операторов".

Таблица 6. Операторы C#

Символ оператора	Название	Пример	Возможность перегрузки
Основные (наивысший приоритет)			
`.`	Доступ к члену	`x.y`	Нет
`?.`	`null`-условный	`x?.y`	Нет

Продолжение табл. 6

Символ оператора	Название	Пример	Возможность перегрузки
`->`	Указатель на структуру (небезопасный)	`x->y`	Нет
`()`	Вызов функции	`x()`	Нет
`[]`	Массив/индекс	`a[x]`	Через индексатор
`++`	Пост-инкремент	`x++`	Да
`--`	Пост-декремент	`x--`	Да
`new`	Создание экземпляра	`new Foo()`	Нет
`stackalloc`	Небезопасное выделение памяти в стеке	`stackalloc(10)`	Нет
`typeof`	Получение типа идентификатора	`typeof(x)`	Нет
`nameof`	Получение имени идентификатора	`nameof(x)`	Нет
`checked`	Проверка целочисленного переполнения	`checked(x)`	Нет
`unchecked`	Отказ от проверки целочисленного переполнения	`unchecked(x)`	Нет
`default`	Значение по умолчанию	`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 (отрицание)	`!x`	Да

Продолжение табл. 6

Символ оператора	Название	Пример	Возможность перегрузки
`~`	Побитовое дополнение	`~x`	Да
`++`	Префиксный инкремент	`++x`	Да
`--`	Префиксный декремент	`--x`	Да
`()`	Приведение типа	`(int)x`	Нет
`*`	Значение по адресу (разыменование) (небезопасный)	`*x`	Нет
`&`	Адрес значения (небезопасный)	`&x`	Нет
	Мультипликативные		
`*`	Умножение	`x*y`	Да
`/`	Деление	`x/y`	Да
`%`	Остаток от деления	`x%y`	Да
	Аддитивные		
`+`	Сложение	`x+y`	Да
`-`	Вычитание	`x-y`	Да
	Сдвиг		
`<<`	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 классу или его подклассу	`x is y`	Нет

Продолжение табл. 6

Символ оператора	Название	Пример	Возможность перегрузки
`as`	Преобразование типа	`x as y`	Нет
	Равенство		
`==`	Равно	`x==y`	Да
`!=`	Не равно	`x!=y`	Да
	Логическое и		
`&`	И	`x&y`	Да
	Логическое исключающее или		
`^`	Исключающее или	`x^y`	Да
	Логическое или		
`|`	Или	`x|y`	Да
	Условное и		
`&&`	Условное и	`x&&y`	Через `&`
	Условное или		
`||`	Условное или	`x||y`	Через `|`
	Объединение с null		
`??`	Объединение с `null`	`x??y`	Нет
	Условный (тернарный)		
`?:`	Условный оператор	`isTrue ? trueThis : falseThis`	Нет
	Присваивание и лямбда (наинизший приоритет)		
`=`	Присваивание	`x=y`	Нет
`*=`	Умножение с присваиванием	`x*=2`	Через `*`
`/=`	Деление с присваиванием	`x/=2`	Через `/`
`+=`	Сложение с присваиванием	`x+=2`	Через `+`

Окончание табл. 6

Символ оператора	Название	Пример	Возможность перегрузки
`-=`	Вычитание с присваиванием	`x-=2`	Через `-`
`<<=`	Сдвиг влево с присваиванием	`x<<=2`	Через `<<`
`>>=`	Сдвиг вправо с присваиванием	`x>>=2`	Через `>>`
`&=`	И с присваиванием	`x&=2`	Через `&`
`^=`	Исключающее или с присваиванием	`x^=2`	Через `^`
`|=`	Или с присваиванием	`x|=2`	Через `|`
`=>`	Лямбда	`x => x+1`	Нет

`null`-операторы

В языке C# определены три оператора, предназначенные для упрощения работы со значениями `null`: *оператор объединения с `null`* (null coalescing), *`null`-условный оператор* (null-conditional) и *оператор присваивания с объединением с `null`*.

Оператор объединения с `null`

Оператор объединения с `null` обозначается как `??`. Он заключается в следующем: если операнд слева не равен `null`, возвращается его значение, в противном случае возвращается другое значение. Например:

```
string s1 = null;
string s2 = s1 ?? "nothing"; // s2 равно "nothing"
```

Если левый операнд не равен `null`, то правый операнд не вычисляется. Оператор объединения с `null` работает также с типами, допускающими значения `null` (см. раздел “Типы-значения, допускающие null”).

Оператор присваивания с объединением с null

Оператор ??= выполняет присваивание переменной, только если ее значение равно null. Таким образом, код

```
myVariable ??= someDefault;
```

эквивалентен коду

```
if (myVariable == null) myVariable = someDefault;
```

null-условный оператор

null-условный оператор обозначается как ?.. Он позволяет вызывать методы и получать доступ к членам подобно стандартному оператору доступа — точке, но с тем отличием, 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(); // Ошибки нет; s равно null
```

Последняя строка кода эквивалентна строке

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

Столкнувшись со значением null, рассматриваемый оператор не вычисляет оставшуюся часть выражения. В следующем примере переменная s получает значение null, несмотря на наличие стандартного оператора точки между ToString() и ToUpper():

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper(); // Ошибки нет
```

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

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

Этот код эквивалентен следующему выражению (с тем отличием, что в нем x.y вычисляется только один 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, так как тип int не может принимать значение null:

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

Исправить ситуацию можно путем применения типа, допускающего значение `null` (см. раздел "Типы-значения, допускающие null"):

```
int? length = sb?.ToString().Length;
// OK : int? может быть null
```

`null`-условный оператор можно также использовать для вызова `void`-метода:

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

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

`null`-условный оператор может применяться с часто используемыми членами типов, которые описаны в разделе "Классы", в том числе с методами, полями, свойствами и индексаторами. Он также хорошо сочетается с оператором объединения с `null`:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "nothing";
// s вычисляется как "nothing"
```

Инструкции

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

Инструкции объявления

Инструкция объявления объявляет новую переменную с возможностью ее дополнительной инициализации посредством выражения. Инструкция 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 схоже с объявлением переменной, с тем отличием, что после объявления константа не может быть изменена, а объявление обязательно должно сопровождаться инициализацией (см. раздел "Константы"):

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

Область видимости локальной переменной

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

Инструкции выражений

Инструкции выражений — это выражения, которые также являются корректными инструкциями. На практике такие выражения что-то "делают"; другими словами, выражения:

- ✓ присваивают или изменяют значение переменной;
- ✓ создают экземпляр объекта;
- ✓ вызывают метод.

Выражения, которые не делают ничего из перечисленного выше, не являются корректными инструкциями:

```
string s = "foo";
s.Length; // Неверная инструкция: она 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 такая инструкция бесполезна:

```
new StringBuilder(); // Корректно, но бесполезно
x.Equals(y);         // Корректно, но бесполезно
```

Инструкции выбора

Инструкции выбора предназначены для управления потоком выполнения программы на основании выполнения некоторых условий.

Инструкция `if`

Инструкция `if` выполняет некоторую другую инструкцию, если результатом вычисления логического выражения является `true`. Например:

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

Инструкцией может быть блок кода:

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

Конструкция `else`

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

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

Внутрь конструкции `else` можно поместить вложенную инструкцию `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 предшествующей инструкции `if` в блоке инструкций. Например:

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

Семантически этот код идентичен следующему:

```
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:

```
if (age >= 35)
    Console.WriteLine("Вы можете стать президентом!");
else if (age >= 21)
    Console.WriteLine("Вы можете выпивать!");
else if (age >= 18)
    Console.WriteLine("Вы можете голосовать!");
else
    Console.WriteLine("Вы можете подождать!");
```

Инструкция `switch`

Инструкции `switch` позволяют организовать ветвление потока выполнения программы на основе выбора из возможных значений, которые переменная может принимать. Инструкции `switch` могут дать в результате более ясный код, чем множество инструкций `if`, потому что они требуют только однократного вычисления 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: // Все прочие 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 одной из инструкций перехода. Ниже перечислены возможные варианты:

- ✓ `break` (переход в конец инструкции `switch`);
- ✓ `goto case x` (переход к другой конструкции `case`);
- ✓ `goto default` (переход к конструкции `default`);
- ✓ любая другая инструкция перехода, например `return`, `throw`, `continue` или `goto` *метка*.

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

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

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

Инструкция switch для типов

Начиная с версии C# 7, инструкция switch может работать с *типами*:

```
static void TellMeTheType(object x)
{
  switch (x)
  {
    case int i:
      Console.WriteLine("Это int!");
      break;
    case string s:
      Console.WriteLine(s.Length); // Используем s
      break;
    case bool b when b == true:
      Console.WriteLine("True");  // Если b - true
      break;
    case null:                     // Работа с null
      Console.WriteLine("null");
      break;
  }
}
```

(Тип object допускает переменную любого типа; см. разделы "Наследование" и "Тип object".)

В каждой конструкции case указываются тип, с которым должно быть выполнено сопоставление, и переменная, которой необходимо присвоить типизированное значение в случае успешного совпадения. В отличие от констант, ограничения на используемые типы отсутствуют. В необязательной конструкции when указывается условие, которое должно быть удовлетворено, чтобы совпадение для case было успешным.

При использовании switch с типами порядок следования конструкций case важен (в отличие от случая констант). 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("f, m, d вне области видимости");
    break;
```

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

Выражения `switch`

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

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

Обратите внимание на то, что ключевое слово `switch` находится после имени переменной, а конструкции `case` являются выражениями (заканчивающими запятыми), а не инструкциями. Переключатель можно организовать и со множественным значением (*кортежем*):

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

Инструкции итераций

Язык C# позволяет многократно выполнять последовательности инструкций с помощью инструкций циклов `while`, `do-while`, `for` и `foreach`.

Циклы while и do-while

Циклы while многократно выполняют код своего тела до тех пор, пока результатом вычисления логического выражения является true. Выражение проверяется перед выполнением тела цикла. Например, следующий код выведет 012:

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

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

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

Циклы for

Циклы for схожи с циклами while, но имеют специальные конструкции для *инициализации* и *итерирования* переменной цикла. Цикл for содержит три части:

```
for( инициализация; условие; итерация)
    инструкция-или-блок-инструкций
```

Инициализация выполняется перед началом цикла и обычно инициализирует одну или несколько переменных *итерации*.

Условие представляет собой логическое выражение, которое проверяется *перед* каждой итерацией цикла. Тело цикла выполняется до тех пор, пока вычисление условия дает true.

Итерация 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 (каждое число Фибоначчи является суммой двух предшествующих чисел Фибоначчи):

```
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 инструкции `for` может быть опущена. Бесконечный цикл можно реализовать так (можно также использовать `while(true)`):

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

Циклы `foreach`

Инструкция `foreach` обеспечивает проход по всем элементам в перечислимом объекте. Большинство типов в C# и .NET, 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 "Вода")
    Console.WriteLine(c + " "); // В о д а
```

Перечислимые объекты рассматриваются в разделе "Перечисление и итераторы".

Инструкции перехода

К инструкциям перехода в C# относятся `break`, `continue`, `goto`, `return` и `throw`. Ключевое слово `throw` рассмотрено в разделе "Инструкции try и исключения".

Инструкция `break`

Инструкция `break` завершает выполнение тела итерации или инструкции `switch`:

```
int x = 0;
while (true)
{
    if (x++ > 5) break; // Прерывание цикла
```

```
}
// После break выполнение продолжается здесь
...
```

Инструкция continue

Инструкция `continue` пропускает оставшиеся инструкции в теле цикла 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
}
```

Инструкция goto

Инструкция `goto` передает выполнение метке (определяемой с использованием двоеточия после идентификатора) в пределах блока инструкций. 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;
}
```

Инструкция return

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

```
static decimal AsPercentage (decimal d)
{
    decimal p = d * 100m;
    return p; // Возврат значения вызывающему коду
}
```

Инструкция `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 с открытым ключом, определен в пространстве имен `System.Security.Cryptography`.

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

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

ПРИМЕЧАНИЕ

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

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

Директива `using`

Директива `using` *импортирует* пространство имен и является удобным средством для обращения к типам без указания их полностью квалифицированных имен. Мы можем ссылаться на `Class1` из предыдущего примера следующим образом:

```
using Outer.Middle.Inner;
Class1 c; // Полностью квалифицированное имя не требуется
```

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

Директива `using static`

Директива `using static` импортирует *тип*, а не пространство имен. После нее все статические члены этого типа можно использовать без полностью квалифицированного имени типа. В следующем примере мы вызываем статический метод `WriteLine` класса `Console`:

```
using static System.Console;
WriteLine("Hello");
```

Директива `using static` импортирует все доступные статические члены типа, включая поля, свойства и вложенные типы. Эту директиву можно также применять к перечислениям (см. раздел “Перечисления”); в этом случае импортируются их члены.

Если между несколькими директивами статического импорта возникает неоднозначность, компилятор C# не может вывести корректный тип из контекста и сообщает об ошибке.

Правила пространств имен

Область видимости имен

Имена, объявленные во внешних пространствах имен, могут использоваться во внутренних пространствах имен без дополнительного указания пространства имен. 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, можно применять частично квалифицированное 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 пространствах имен, то преимущество получает тип из

внутреннего пространства имен. Чтобы сослаться на тип во внешнем пространстве имен, имя требуется квалифицировать.

ПРИМЕЧАНИЕ

Все имена типов во время компиляции преобразуются в полностью квалифицированные имена. В коде на промежуточном языке (Intermediate Language — IL) неполные или частично квалифицированные 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 исходные файлы и сборки.

Квалификатор `global::`

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

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

Псевдонимы типов и пространств имен

Импорт пространства имен может привести к конфликту имен типов. Вместо полного пространства имен можно импортировать только конкретные типы 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* и *ограничения*, *базовый класс* и *интерфейсы*
Внутри фигурных скобок	*Члены класса* (к ним относятся *методы*, *свойства*, *индексаторы*, *события*, *поля*, *конструкторы*, *перегруженные операторы*, *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 значение типа по умолчанию (`0`, `\0`, `null`, `false`). Инициализаторы полей выполняются перед конструкторами в порядке, в котором они указаны.

Для удобства множество полей одного типа можно объявлять списком, 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 очень похоже на макрос в языке программирования C). Константа может иметь любой встроенный числовой тип, быть `bool`, `char`, `string` или перечислением.

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

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

Константа является гораздо более ограничивающей, чем поле `static readonly`, как в плане типов, которые можно применять, так и в плане 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 инструкций. Он может получать *входные* данные из вызывающего кода посредством *параметров* и возвращать *выходные* данные обратно вызывающему коду с помощью *возвращаемого типа*. Метод может иметь возвращаемый тип `void`, который указывает на то,

что метод ничего не возвращает вызывающему коду. Метод также может возвращать выходные данные вызывающему коду через параметры, объявленные как `ref` и `out`.

Сигнатура метода должна быть уникальной в рамках типа. Она включает в себя имя метода и типы параметров в указанном в объявлении порядке (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; }
```

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);
```

Локальные методы

Метод может быть определен внутри другого метода:

```
void WriteCubes()
{
    Console.WriteLine(Cube (3));

    int Cube(int value) => value*value*value;
}
```

Локальный метод (в данном случае — `Cube()`) будет видимым только для охватывающего метода (`WriteCubes()`). Это упрощает содержащий метод тип и сигнализирует любому, кто просматривает код, что `Cube()` больше нигде не применяется. Локальные методы могут обращаться к локальным переменным и параметрам охватывающего метода, что имеет несколько последствий, описанных в разделе “Захват внешних переменных”.

Локальные методы могут появляться внутри функций других видов, таких как средства доступа к свойствам, конструкторы и

так далее, и даже внутри других локальных методов. Локальные методы могут быть итераторными или асинхронными.

Методы, объявленные в инструкциях верхнего уровня, неявно являются локальными; мы можем продемонстрировать это следующим образом:

```
int x = 3; Foo();
void Foo() => Console.WriteLine(x); // Имеется доступ к x
```

Статические локальные методы

Добавление модификатора `static` к локальному методу (начиная с C# 8) предотвращает его обращение к локальным переменным и параметрам охватывающего метода. Это помогает уменьшить связность и предотвратить локальный метод от случайного обращения к переменным в охватывающем методе.

Перегрузка методов

ПРИМЕЧАНИЕ

Локальные методы не могут быть перегружены. Это значит, что методы, объявленные в инструкциях верхнего уровня (которые рассматриваются как локальные методы), перегружены быть не могут.

Тип может перегружать методы (иметь несколько методов с одним и тем же именем) при условии, что типы параметров различаются. 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);
```

Конструкторы экземпляров

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

```
Panda p = new Panda("Petey"); // Вызов конструктора

public class Panda
{
    string name;              // Определение поля
    public Panda (string n)   // Определение конструктора
    {
        name = n;             // Код инициализации
    }
}
```

Конструкторы с единственной инструкцией 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, первым выполняется *вызванный конструктор*.

Другому конструктору можно передавать *выражение* следующим образом:

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

В самом выражении не допускается применять ссылку `this`, например, для вызова метода экземпляра. Однако 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 был определен ни один конструктор. Однако после определения хотя бы одного конструктора конструктор без параметров 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+) выполняет противоположное и присваивает поля набору переменных. Имя метода деконструктора должно быть `Deconstruct()`, а сам метод должен иметь один или более 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
```

Вторая строка представляет собой вызов деконструктора. Она создает две локальные переменные, а затем обращается к методу `Deconstruct()`. Такой вызов деконструктора эквивалентен следующему коду:

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

Вызовы деконструктора допускают неявную типизацию, так что наш вызов можно было бы сократить следующим образом:

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

Или так:

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

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

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

Перегружая метод `Deconstruct()`, вызывающему коду можно предложить целый набор вариантов деконструкции.

ПРИМЕЧАНИЕ

Метод `Deconstruct()` может быть расширяющим методом (см. раздел "Расширяющие методы"). 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 инициализированы 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 следующим образом:

```
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

Внешне свойства выглядят схожими с полями, но внутренне они содержат логику подобно методам. Например, взглянув на следующий код, невозможно сказать, 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 "закулисное"
поле

  public decimal CurrentPrice // Открытое свойство
  {
    get { return currentPrice; }
    set { currentPrice = value; }
  }
}
```

`get` и `set` обозначают средства доступа (*accessors*) к свойству. Средство доступа `get` вызывается при чтении свойства. Оно должно возвращать значение, имеющее тип самого свойства. Средство доступа `set` выполняется при присваивании свойству значения. Оно принимает неявный параметр по имени `value` с типом свойства, который обычно присваивается закрытому полю (в данном случае полю `currentPrice`).

Хотя доступ к свойствам осуществляется таким же способом, как и 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 свойства пользователю. В приведенном примере метод `set` мог бы, например, генерировать исключение, если значение `value` выходит за пределы допустимого диапазона.

ПРИМЕЧАНИЕ

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

Свойство 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

Свойство только для чтения, подобное показанному 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 оператором `=>`:

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

Начиная с версии C# 7, также допускается объявлять сжатыми до выражения 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 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, к которому невозможно обратиться. Средство доступа `set` может быть помечено как `private` или `protected`, если вы хотите сделать данное свойство доступным другим типам только для чтения.

Инициализаторы свойств

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 присваиваться также в конструкторе типа. Это удобно при создании *неизменяемых* (доступных только для чтения) типов.

Доступность `get` и `set`

Средства доступа `get` и `set` могут иметь разные уровни доступа. В типичном сценарии может быть 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 (в данном случае — `public`), а к средству доступа, которое должно быть *менее* доступным, добавлен соответствующий модификатор.

Инициализирующие установщики (C# 9)

Начиная с C# 9, вы можете объявить средство доступа к свойству с использованием `init` вместо `set`:

```
public class Note
{
    public int Pitch    { get; init; } = 20;
    public int Duration { get; init; } = 100;
}
```

Такие *только инициализируемые* свойства действуют подобно свойствам только для чтения, с тем отличием, что их значения могут быть установлены с использованием инициализатора объекта:

```
var note = new Note { Pitch = 50 };
```

После этого свойство не может быть изменено:

```
note.Pitch = 200; // Ошибка - свойство init!
```

Только инициализируемые свойства не могут быть установлены даже внутри своего класса, кроме как через инициализатор свойства, конструктор или другое только инициализируемое свойство.

Альтернативой только инициализируемым свойствам являются свойства, доступные только для чтения, которые вы устанавливаете в конструкторе:

```
public Note (int pitch = 20, int duration = 100)
{
    Pitch = pitch; Duration = duration;
}
```

Если класс является частью открытой библиотеки, такой подход затрудняет управление версиями, так как добавление необязательного параметра в конструктор позже нарушает бинарную совместимость с потребителями (тогда как добавление нового только инициализируемого свойства ничего не нарушает).

ПРИМЕЧАНИЕ

Только инициализируемые свойства имеют еще одно существенное преимущество: они допускают недеструктивное изменение при использовании вместе с записями (см. раздел “Записи (C# 9)”).

Так же, как и обычные средства доступа `set`, средства доступа `init` могут иметь реализации:

```
public class Point
{
    readonly int _x;
    public int X { get => _x; init => _x = value; }
    ...
```

Обратите внимание, что поле `_x` доступно только для чтения: `init` может изменять поле, объявленное как `readonly`, только в собственном классе. (Без этой возможности поле `_x` должно было бы быть доступным и класс не был бы внутренне неизменным.)

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 индексный аргумент, а не через имя свойства. Класс `string` имеет индексатор, который позволяет получать доступ к каждому значению `char` в нем с использованием индекса типа `int`:

```
string s = "hello";
Console.WriteLine(s[0]); // 'h'
Console.WriteLine(s[3]); // 'l'
```

Синтаксис использования индексаторов подобен синтаксису работы с массивами, с тем отличием, что аргумент (или аргументы) индекса может быть любого типа. Индексаторы могут вызываться `null`-условно с использованием вопросительного знака перед открывающей квадратной скобкой (см. раздел “null-операторы”):

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

Реализация индексатора

Для реализации индексатора необходимо определить свойство с именем `this`, указав аргументы в квадратных скобках. Например:

```
class Sentence
{
```

```
    string[] words = "Быстрая рыжая лиса".Split();

    public string this [int wordNum] // Индексатор
    {
        get { return words [wordNum]; }
        set { words [wordNum] = value; }
    }
}
```

Ниже показано, как можно применять такой индексатор:

```
Sentence s = new Sentence();
Console.WriteLine(s[2]); // лиса
s[3] = "собака";
Console.WriteLine(s[2]); // собака
```

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:

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

Использование индексов и диапазонов с помощью индексаторов

Поддерживать в своих классах индексы и диапазоны (см. раздел “Индексы и диапазоны”) можно с помощью 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];
```

Это позволит написать следующий код:

```
Sentence s = new Sentence();
Console.WriteLine(s[^1]);        // лиса
string[] firstTwoWords = s[..2]; // (Быстрая, рыжая)
```

Статические конструкторы

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

```
class Test
{
    static Test(){ Console.Write("Инициализация типа"); }
}
```

Исполняющая среда автоматически вызывает статический конструктор непосредственно перед началом использования типа. Этот вызов инициируется двумя действиями: созданием экземпляра типа и доступом к статическому члену типа.

ПРИМЕЧАНИЕ

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

ПРИМЕЧАНИЕ

Начиная с C# 9, вы также можете определять *инициализаторы модулей*, которые выполняются один раз для каждой сборки (когда сборка впервые загружается). Чтобы определить инициализатор модуля, напишите статический `void`-метод и примените к нему атрибут `[Module Initializer]`:

```
[System.Runtime.CompilerServices.
ModuleInitializer]
internal static void InitAssembly()
{
...
}
```

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

имеет статического конструктора, то инициализаторы статических полей будут выполняться непосредственно перед началом использования типа — или *в любой момент раньше* по прихоти исполняющей среды.

Статические классы

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

Финализаторы

Финализаторы — это методы, предназначенные только для классов, которые выполняются перед тем, 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 перекрывает метод `Finalize()` класса `object`. Сборка мусора и финализаторы подробно обсуждаются в главе 12 книги *C# 9.0. Справочник. Полное описание языка*.

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 – автоматически сгенерированный
partial class PaymentForm { ... }

// PaymentForm.cs – написанный программистом
partial class PaymentForm { ... }
```

Каждый участник должен иметь объявление `partial`.

Участники не могут содержать конфликтующие члены. Например, нельзя повторять конструктор с одними и теми же параметрами. Частичные типы полностью разрешаются компилятором, а это значит, что каждый участник должен быть доступен на этапе компиляции и располагаться в одной и той же сборке.

Базовый класс может быть указан как для единственного участника, так и для множества участников (при условии, что для каждого из них базовый класс будет одним и тем же). Кроме того, для каждого участника можно независимо указывать интерфейсы, подлежащие реализации. Базовые классы и интерфейсы рассматриваются в разделах “Наследование” и “Интерфейсы”.

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 автоматически
{                           // сгенерированном файле
    partial void ValidatePayment (decimal amount);
}

partial class PaymentForm // в файле, созданном
{                         // программистом
    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 определение частичного метода при компиляции удаляется (вместе с кодом его вызова). Это дает автоматически сгенерированно-

му коду бо́льшую свободу в предоставлении точек привязки, не заставляя беспокоиться по поводу разбухания кода. Частичные методы должны быть `void`-методами, и неявно они являются `private`.

Расширенные частичные методы (C# 9)

Расширенные частичные методы предназначены для обратного сценария генерации кода, в котором программист определяет точки привязки, которые реализует генератор кода. Это может происходить, например, в *генераторах исходных текстов* (функциональная возможность Roslyn, позволяющая вам передавать компилятору сборку, которая автоматически генерирует части вашего кода).

Объявление частичного метода *расширенное*, если оно начинается с модификатора доступности:

```
public partial class Test
{
  public  partial void M1(); // Расширенный частичный метод
  private partial void M2(); // Расширенный частичный метод
}
```

Наличие модификатора доступности влияет не только на доступность: он также сообщает компилятору о том, что данное объявление должно рассматриваться особым образом.

Расширенные частичные методы *обязаны* иметь реализации. В приведенном примере и метод `M1`, и метод `M2` должны иметь реализации, потому что каждый из них имеет модификатор доступа (`public` и `private` соответственно).

Поскольку они не могут быть удалены, расширенные частичные методы могут возвращать любой тип и включать параметры `out`.

Оператор `nameof`

Оператор `nameof` возвращает имя любого символа (типа, члена, переменной и т.д.) в виде строки:

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

Преимущество применения данного оператора по сравнению с простым указанием строки связано со статической проверкой типов. Инструменты, подобные Visual Studio, способны воспринимать символические ссылки, так что переименование любого символа приводит к переименованию его ссылок.

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

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

Результатом будет `"Length"`. Чтобы получить `"StringBuilder.Length"`, понадобится 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 он может быть унаследован множеством классов, формируя иерархию классов. В этом примере мы начнем с определения класса `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 имеется в `Asset`, плюс любые дополнительные члены, которые в них будут определены:

```
public class Stock : Asset // Производный от Asset
{
    public long SharesOwned;
}
public class House : Asset // Производный 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`, так как 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 объект типа `House`, то передача ему `Asset` станет невозможной.

Приведение и преобразования ссылок

Ссылка на объект может быть:

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

Приведения вверх и вниз между совместимыми ссылочными типами выполняют *преобразования ссылок*: создается новая ссыл-

ка, которая указывает на *тот же 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);        // OK
Console.WriteLine(a.SharesOwned); // Ошибка времени
                                  // компиляции
```

Последняя строка кода вызывает ошибку времени компиляции, поскольку переменная `a` имеет тип `Asset`, несмотря на то что она ссылается на объект типа `Stock`. Чтобы получить доступ к полю `SharesOwned`, требуется приведение `Asset` 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); // OK
Console.WriteLine(s == a);        // True
Console.WriteLine(s == msft);     // True
```

Как и в случае приведения вверх, затрагиваются только ссылки, но не лежащий в их 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; // Ошибка: a не является Stock
```

Когда приведение вниз терпит неудачу, генерируется исключение `InvalidCastException`. Это пример *проверки типов времени выполнения* (которая более подробно рассматривается в разделе "Проверка типов — статическая и времени выполнения").

Оператор as

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

```
Asset a = new Asset();
Stock s = a as Stock; // s == null; исключения нет
```

Этот оператор удобен, когда нужно организовать последующую проверку результата на предмет равенства `null`:

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

Оператор `as` не может выполнять специальные преобразования (см. раздел "Перегрузка операторов"), равно как и числовые преобразования.

Оператор is

Оператор `is` проверяет, будет ли преобразование ссылки успешным; другими словами, является ли объект производным от указанного класса (или реализует ли он какой-либо интерфейс). Он часто применяется при проверке перед приведением вниз:

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

Оператор `is` возвращает `true`, также если может успешно выполниться распаковывающее преобразование (см. раздел "Тип object"). Однако он не принимает во внимание специальные или числовые преобразования.

Начиная с версии C# 7, появилась возможность при использовании оператора is вводить переменную:

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

Введенная переменная доступна для "немедленного" употребления и остается в области видимости за пределами выражения `is`:

```
if (a is Stock s && s.SharesOwned > 100000)
    Console.WriteLine("Wealthy");
else
    s = new Stock();              // s в области видимости
Console.WriteLine(s.SharesOwned); // s все еще 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, свойства, 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 о таком синтаксисе обращайтесь к разделу "Свойства, сжатые до выражений".) Подкласс перекрывает виртуальный метод с помощью 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. Однако класс `House` специализирует свойство `Liability` так, чтобы оно возвращало 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 перекрытого методов должны быть идентичны. Внутри пере-

крытого метода можно вызвать реализацию метода из базового класса с помощью ключевого слова `base` (см. раздел "Ключевое слово base").

Ковариантные возвраты (C# 9)

Начиная с C# 9, вы можете перекрыть метод (или средство доступа `get` свойства) так, чтобы он возвращал *более производный* тип (подкласса). Например, вы можете написать метод `Clone()` в классе `Asset`, который возвращает `Asset`, и перекрыть этот метод в классе `House` так, что он возвратит `House`.

Это разрешено, потому что это не нарушает контракт, по которому метод `Clone()` должен вернуть `Asset`: он возвращает `House`, который *является* `Asset`.

Абстрактные классы и абстрактные члены

Класс, объявленный как *абстрактный* (`abstract`), не может быть инстанцирован. Можно инстанцировать только его конкретные *подклассы*.

В абстрактных классах есть возможность определять *абстрактные члены*. Абстрактные члены схожи с виртуальными членами, с тем отличием, что они не предоставляют реализацию по умолчанию. Такая реализация должна обеспечиваться подклассом, если только подкласс также не объявлен как абстрактный:

```
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 к базовому типу добавляется член *после* того, как идентичный член уже был добавлен к подтипу. В таком случае компилятор генерирует предупреждение, а затем разрешает неоднозначность следующим образом:

- ✓ ссылки на `A` (во время компиляции) связываются с `A.Counter`;
- ✓ ссылки на `B` (во время компиляции) связываются с `B.Counter`.

Иногда необходимо преднамеренно скрыть какой-либо 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 программистам — о том, что дублирование члена произошло неслучайно.

Запечатывание функций и классов

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

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

Можно также применить модификатор `sealed` к самому классу, запрещая тем самым наследование этого класса подклассами.

Ключевое слово `base`

Ключевое слово `base` схоже с ключевым словом `this`. Оно служит двум важным целям: для доступа к перекрытой функции-члену из подкласса и для вызова 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 *невиртуально*. Это значит, что мы всегда обращаемся к версии данного свойства из Asset, независимо от фактического типа экземпляра во время выполнения.

Тот же подход работает и в ситуации, когда свойство Liability *сокрыто*, а не *перекрыто*. (Доступ к скрытым членам можно получить путем приведения к базовому классу перед вызовом функции.)

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 далее код будет некорректным:

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

В классе Subclass должны быть "повторно определены" любые необходимые конструкторы. 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; это гарантирует выполнение *базовой* инициализации перед *специализированной* инициализацией.

Если в конструкторе подкласса опустить ключевое слово `base`, то будет неявно вызываться конструктор базового класса *без параметров* (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 следующем 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 разрешение

Наследование оказывает интересное влияние на перегрузку методов. Предположим, что есть следующие две перегруженные 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)
```

ПРИМЕЧАНИЕ

Если привести `Asset` к `dynamic` (см. раздел “Динамическое связывание”), то решение о том, какая перегружен-

ная версия должна вызываться, откладывается до времени выполнения, и выбор основывается на фактическом типе объекта.

Тип `object`

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

Чтобы проиллюстрировать, насколько это полезно, рассмотрим *стек* общего назначения. Стек является структурой данных, работа которой основана на принципе LIFO ("Last-In First-Out" — "последним пришел — первым вышел"). Стек поддерживает две операции: *занесение* объекта в стек и *снятие* объекта со 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("Стек");
string s = (string) stack.Pop(); // Приведение вниз
Console.WriteLine(s);            // Стек
```

`object` относится к ссылочным типам в силу того, что представляет собой класс. Несмотря на это типы-значения, 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 различий между типами значений и ссылочными типами. Данный процесс называется *упаковкой* (boxing) и *распаковкой* (unboxing).

ПРИМЕЧАНИЕ

В разделе “Обобщения” будет показано, как усовершенствовать класс `Stack`, чтобы улучшить поддержку стеков однотипных элементов.

Упаковка и распаковка

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

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

Распаковка (unboxing) представляет собой обратную операцию, которая предусматривает приведение объекта к исходному типу-значению:

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

Распаковка требует явного приведения. Исполняющая среда проверяет, соответствует ли указанный тип значения фактическому объектному типу, и генерирует исключение `InvalidCast Exception`, если это не так. Например, показанный ниже код ведет к генерации исключения, поскольку тип `long` не соответствует типу `int`:

```
object obj = 9;       // 9 выводится как тип int
long x = (long) obj; // Генерация InvalidCastException
```

Однако следующий код выполняется успешно:

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

И этот код также не вызывает ошибки:

```
object obj = 3.5;          // 3.5 выводится как тип double
int x = (int)(double)obj; // x 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
```

Проверка типов — статическая и времени выполнения

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

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

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

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

```
object y = "5";
int z = (int)y; // Ошибка 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 с помощью метода `GetType()` класса `object`.

Метод `GetType()` и оператор `typeof`

Все типы в C# во время выполнения представлены с помощью экземпляра `System.Type`. Получить объект `System.Type` можно двумя основными способами: вызвать метод `GetType()` экземпляра или воспользоваться оператором `typeof` с именем типа. Результат `GetType()` вычисляется во время выполнения, а `typeof` — статически во время компиляции.

В классе System.Type предусмотрены свойства для таких вещей, как имена 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 имеются также методы, которые действуют в качестве шлюза для модели рефлексии времени выполнения. За подробной информацией обращайтесь к главе 19 книги *C# 9.0. Справочник. Полное описание языка*.

Список членов object

Вот список всех членов object:

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

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

Метод Equals() класса object схож с оператором ==, с тем отличием, что Equals() является виртуальным методом, а оператор == — статическим. 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 выполняет статическую привязку к оператору == класса object, которая для сравнения двух экземпляров применяет семантику *ссылочного типа*. (Из-за того, что x и y упакованы,

они находятся в разных ячейках памяти, и 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()` первого аргумента (после проверки, не равны ли аргументы `null`):

```
object x = null, y = 3;
bool error = x.Equals(y);     // Ошибка времени выполнения
bool ok = object.Equals(x,y); // OK (false)
```

Метод `ReferenceEquals()` принудительно применяет сравнение эквивалентности ссылочных типов (что иногда удобно для ссылочных типов, в которых оператор `==` был перегружен для выполнения другого действия).

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

Чтобы настроить семантику эквивалентности типов, требуется как минимум переопределить методы `Equals()` и `GetHashCode()`. Обычно также перегружаются операторы `==` и `!=`. Пример такой настройки приведен в разделе “Перегрузка операторов”.

Метод `ToString()`

Метод `ToString()` возвращает текстовое представление экземпляра типа по умолчанию. 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 пользовательских типах можно следующим образом:

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

Структуры

Структура схожа с классом, но обладает следующими ключевыми отличиями.

- ✓ Структура является типом-значением, 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 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 инстанцирования в куче (с последующей сборкой мусора); это дает ощутимую экономию при создании большого количества экземпляров типа.

Как и любой тип-значение, структура может косвенно оказаться в куче либо из-за упаковки, либо из-за того, что она является полем класса. Если создать экземпляр показанного ниже класса `SomeClass`, то поле `Y` ссылалось бы на структуру в куче:

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

Аналогично, если бы мы объявили массив из элементов типа `SomeStruct`, то экземпляр был бы сохранен в куче (так как массивы являются ссылочными типами), хотя весь массив потребовал бы только одного выделения памяти.

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

Семантика конструирования структуры

Семантика конструирования структуры выглядит следующим образом.

- ✓ Имеется 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 только для чтения

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

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

Если модификатор `readonly` требует применения с большей степенью детализации, то вы можете, начиная с C# 8, применять модификатор `readonly` к *функциям* структуры. Если такая функция попытается модифицировать любое поле, сгенерируется ошибка времени компиляции:

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

Если `readonly`-функция вызывает функцию, не являющуюся `readonly`, компилятор выдаст предупреждение (и в качестве защиты скопирует структуру — во избежание возможности ее изменения).

Модификаторы доступа

Для содействия инкапсуляции тип или член типа может ограничивать свою доступность для других типов и сборок с помощью добавления к объявлению одного из шести модификаторов доступа, описанных ниже.

`public`

Полная доступность. Это неявная доступность для членов перечислений и интерфейсов.

`internal`

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

`private`

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

`protected`

Доступность только внутри содержащего типа или в его подклассах.

`protected internal`

Объединение доступностей `protected` и `internal` (*менее* ограничивающая доступность, чем `protected` или `internal` по отдельности, так как 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 вне его сборки, а класс `Class1` — недоступен:

```
class Class1 {}  // Class1 по умолчанию internal
public class Class2 {}
```

Класс `ClassB` открывает поле `x` другим типам в той же 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. Ком-

пилятор препятствует любому несогласованному использованию модификаторов доступа — например, подкласс может иметь меньшую доступность, чем базовый класс, но не бо́льшую.

Дружественные сборки

Члены с модификатором `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 подписана строгим именем, требуется указывать ее *полный* 160-байтный открытый ключ. Извлечь этот ключ можно с помощью запроса LINQ — вы можете найти интерактивный пример в бесплатной библиотеке примеров LINQPad для главы 3 книги *C# 9.0. Справочник. Полное описание языка*.

Установка верхнего предела доступности

Тип устанавливает верхний предел доступности объявленных в нем членов. Наиболее распространенным примером такой установки является ситуация, когда есть тип `internal` с членами `public`. Например:

```
class C { public void Foo() {} }
```

Доступность по умолчанию `internal` класса `C` устанавливает верхний предел доступности метода `Foo()`, по сути, делая `Foo()` объявленным как `internal`. Распространенная причина пометки `Foo()` как `public` связана с облегчением рефакторинга, если позже будет решено изменить доступность класса `C` на `public`.

Интерфейсы

Интерфейс схож с классом, но обеспечивает для своих членов только спецификацию, а не реализацию (хотя, начиная с версии C# 8.0, интерфейс способен предоставлять реализацию *по умол-*

чанию; см. раздел “Методы интерфейсов по умолчанию”). Интерфейс обладает следующими особенностями.

- ✓ Все члены интерфейса *неявно являются 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 *единственного* класса, а структура вообще не поддерживает наследование (за исключением порождения от `System.ValueType`).

Объявление интерфейса схоже с объявлением 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 `public`-реализации 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 с помощью 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() в Widget");
    }
    int I2.Foo()      // Явная реализация I2.Foo
    {
        Console.Write("Реализация I2.Foo() в Widget");
        return 42;
    }
}
```

Поскольку интерфейсы `I1` и `I2` имеют конфликтующие сигнатуры `Foo()`, класс `Widget` явно реализует метод `Foo()` интерфейса `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() в Widget
((I1)w).Foo(); // Реализация I1.Foo() в Widget
((I2)w).Foo(); // Реализация I2.Foo() в 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 избегает 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 перекрыть, он должен быть помечен в базовом классе как `virtual` или `abstract`: вызов этого члена интерфейса через базовый класс либо интерфейс приводит к вызову его реализации из подкласса.

Явно реализованный член интерфейса не может быть помечен как `virtual`, как и не может быть перекрыт 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 "перекрыть" 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, так как обращение к члену через базовый класс приводит к вызову базовой реализации.

Методы интерфейсов по умолчанию

Начиная с версии C# 8, к члену интерфейса можно добавлять реализацию по умолчанию, делая этот член необязательным для реализации:

```
interface ILogger
{
    void Log(string text) => Console.WriteLine(text);
}
```

Такая возможность полезна, когда требуется добавить член к интерфейсу, определенному в некоторой популярной библиотеке, но при этом не нарушить работу (потенциально тысяч) реализаций.

Реализации по умолчанию всегда являются явными, так что если в классе, реализующем `ILogger`, отсутствует определение метода `Log()`, то вызвать его получится только единственным способом — через интерфейс:

```
class Logger : ILogger { }
...
((ILogger)new Logger()).Log ("Сообщение");
```

Тем самым решается проблема наследования множества реализаций: если один и тот же член по умолчанию был добавлен в два интерфейса, которые реализует класс, то неоднозначность относительно того, какой член вызывается, никогда не возникнет.

Теперь в интерфейсах можно определять и статические члены (включая поля), доступ к которым осуществляется из кода внутри реализаций по умолчанию:

```
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 извне:

```
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 значение целочисленного типа. Лежащие в основе значения по умолчанию относятся к типу `int`, а членам перечисления присваиваются

константные значения 0, 1, 2… (в порядке их объявления). Можно указать и иной целочисленный тип:

```
public enum BorderSide : byte { Left,Right,Top,Bottom }
```

Для каждого члена перечисления можно также указывать явные значения:

```
public enum BorderSide : byte
    { Left=1, Right=2, Top=10, Bottom=11 }
```

Кроме того, компилятор позволяет явно присваивать значения некоторым членам перечисления. Члены, значения которым присвоены не были, получают значения на основе последовательного увеличения 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 лежащее в его основе целочисленное значение и из него 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 конкретном примере BorderSide не имеет членов с целочисленным значением 0. Но это не приводит к ошибке: ограниченность перечислений в том, 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-флаги

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 целочисленные значения элементов перечисления:

```
BorderSides leftRight =
    BorderSides.Left | BorderSides.Right;

if ((leftRight & BorderSides.Left) != 0)
    Console.WriteLine ("Включает Left"); // В leftRight бит
                                         // Lef установлен
string formatted = leftRight.ToString(); //"Left, Right"

BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine(s == leftRight);       // True
```

К таким типам комбинируемых перечислений должен быть применен атрибут `Flags`, иначе вызов `ToString()` для экземпляра перечисления возвратит число, а не последовательность имен.

Для удобства комбинации-члены могут быть помещены в 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
}
```

Операторы для работы с перечислениями

Ниже указаны операторы, которые могут работать с перечислениями:

```
=    ==   !=   <    >    <=   >=   +    -    ^
&    |    ~    +=   -=   ++   -    sizeof
```

Операторы побитовые, арифметические и сравнения возвращают результат обработки целочисленных значений элементов перечислений. Сложение разрешено для перечисления и целочисленного типа, но не для двух перечислений.

Вложенные типы

Вложенный тип объявляется внутри области видимости некоторого другого типа. Например:

```
public class TopLevel
{
  public class Nested { }              // Вложенный класс
  public enum Color { Red, Blue, Tan } // Вложенное
}                                      // перечисление
```

Вложенный тип обладает следующими характеристиками.

- ✓ Может получать доступ к закрытым членам охватывающего типа и ко всему остальному, к чему имеет доступ охватывающий тип.
- ✓ Может быть объявлен с полным диапазоном модификаторов доступа, а не только `public` и `internal`.
- ✓ Доступностью вложенного типа по умолчанию является `private`, а не `internal`.
- ✓ Доступ к вложенному типу извне требует указания имени охватывающего типа (как при обращении к статическим членам).

Например, получить доступ к члену `Color.Red` извне класса `TopLevel` можно 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, повторно используемого различными типами: *наследование* и *обобщения* (generics). В то время как наследование выражает повторное использование с помощью базового типа, обобщения делают это посредством "шаблона", который содержит "типы-заполнители". Обобщения, по сравнению 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 пользователем обобщенного типа, которые передаются как *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];
}
```

Использовать `Stack<T>` можно следующим образом:

```
var stack = new Stack<int>();
stack.Push(5);
stack.Push(10);
int x = stack.Pop(); // x равно 10
int y = stack.Pop(); // y равно  5
```

ПРИМЕЧАНИЕ

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 вместо T используется тип object (см. пример в разделе “Тип object”).

Класс Stack<int> заменяет 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 являются закрытыми — с соответствующей заменой 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 использовать следующим образом:

```
int x = 5, y = 10;
Swap(ref x, ref y);
```

Как правило, предоставлять аргументы типа обобщенному методу нет нужды, поскольку компилятор способен вывести их самостоятельно. Если же 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 *вводит* параметры типа (посредством синтаксиса с угловыми скобками). Метод `Pop()` в нашем обобщенном стеке просто использует существующий параметр типа `T`, а потому и не классифицируется как обобщенный.

Методы и типы — единственные конструкции, в которых могут вводиться параметры типа. Свойства, индексаторы, события, поля, конструкторы, операции и так далее 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 могут их *вводить*.

Объявление параметров типа

Параметры типа могут быть введены в объявлениях классов, структур, интерфейсов, делегатов (см. раздел “Делегаты”) и методов. Можно указывать несколько параметров типа, разделяя их запятыми:

```
class Dictionary<TKey, TValue> {...}
```

Вот как он инстанцируется:

```
var myDict = new Dictionary<int,string>();
```

Имена обобщенных типов и методов могут быть перегружены при условии, что количество параметров типа у них различается. Например, показанные ниже три имени типов не конфликтуют друг с другом:

```
class A {}
class A<T> {}
class A<T1,T2> {}
```

ПРИМЕЧАНИЕ

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).

Оператор typeof и несвязанные обобщенные типы

Во время выполнения открытых обобщенных типов не существует: они закрываются как часть 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 оператора typeof:

```
class A<T> {}
class A<T1,T2> {}
...
Type a1 = typeof(A<>);  // Несвязанный тип
Type a2 = typeof(A<,>); // Указывает на 2 аргумента типа
Console.Write(a2.GetGenericArguments().Count());    // 2
```

Оператор typeof можно использовать также для указания закрытого типа:

```
Type a3 = typeof (A<int,int>);
```

или открытого типа (который закрыт во время выполнения):

```
class B<T> { void X() { Type t = typeof (T); } }
```

Обобщенное значение по умолчанию

Ключевое слово default может применяться для получения значения параметра типа обобщения по умолчанию. Значением по умолчанию 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 : base-class // Ограничение базового класса
where T : interface  // Ограничение batqcf
where T : class      // Ограничение ссылочного типа
where T : class?     // (См. раздел
    // "Ссылочные типы, допускающие значение null")
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
where T : notnull    // Тип-значение или ссылочный тип,
                     // не могущий быть null
```

В следующем примере `GenericClass<T,U>` требует, чтобы тип `T` был производным от класса `SomeClass` (или идентичен ему) и реализовывал интерфейс `Interface1`, а тип `U` имел 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-значением (не допускающим `null`) соответственно. *Ограничение unmanaged* является более сильной версией *ограничения struct*: тип `T` должен быть простым типом-значением или структурой, которая (рекурсивно) не имеет ссылочных типов. *Ограничение конструктора без параметров* требует, чтобы тип `T` имел открытый конструктор без параметров и позволял вызывать операцию `new()` для `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).

Подклассы 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 с помощью конкретного типа:

```
class IntStack : Stack<int> {...}
```

Подкласс может также вводить новые аргументы типа:

```
class List<T> {...}
class KeyedList<T,TKey> : List<T> {...}
```

Самоссылающиеся обобщенные объявления

При закрытии аргумента типа тип может указывать в качестве конкретного типа *самого себя*:

```
public interface IEquatable<T> { bool Equals (T obj); }

public class Balloon : IEquatable<Balloon>
{
public bool Equals (Balloon b) { ... }
}
```

Следующий код также корректен:

```
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

ПРИМЕЧАНИЕ

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>` преобразуется в `X<B>`.

(Согласно понятию вариантности в C#, “преобразуется” означает возможность преобразования через *неявное ссылочное*

преобразование — такое, как когда A *является подклассом* B или A *реализует* B. Сюда не входят числовые преобразования, упаковывающие преобразования и пользовательские преобразования.)

Например, тип IFoo<T> имеет ковариантный тип T, если корректен следующий код:

```
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 методов) и помечает параметр типа как *ковариантный*, разрешая написание такого кода:

```
// Полагая, что Bear является подклассом Animal:
var bears = new Stack<Bear>();
bears.Push(new Bear());
// Так как bears реализует IPoppable<Bear>,
// его можно конвертировать в IPoppable<Animal>:
IPoppable<Animal> animals = bears;   // Корректно
Animal a = animals.Pop();
```

Приведение bears к animals разрешено компилятором — в силу того, что параметр типа в интерфейсе является ковариантным.

ПРИМЕЧАНИЕ

Интерфейсы IEnumerator<T> и IEnumerable<T> (см. раздел "Перечисление и итераторы") 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. Напри-

мер, оно предотвращает добавление к этому интерфейсу метода Push(T), который пользователи могли бы неправильно применять для внешне безобидной операции внесения в стек объекта, представляющего верблюда, в реализацию IPoppable<Animal> (вспомните, что базовым типом в нашем примере является стек медведей). Чтобы можно было определить метод Push(T), параметр типа T в действительности должен быть *контравариантным*.

ПРИМЕЧАНИЕ

В языке C# ковариантность (и контравариантность) поддерживается только для элементов со *ссылочными*, 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 мы видели 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>. Контравариантность поддерживается интерфейсами и делегатами, когда параметр типа встречается только во *входных* позициях, обозначаемых с помощью модификатора in. Продолжая предыдущий пример, если класс Stack<T> реализует интерфейс

```
public interface IPushable<in T> { void Push (T obj); }
```

то вполне корректно поступать так:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals; // Корректно
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. У делегатов имеется 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 вызывает делегата, после чего делегат вызывает целевой метод. Такая косвенность развязывает вызывающий компонент и целевой метод.

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:

```
Transformer t = Square;  // Создание экземпляра делегата
int result = t(3);       // Вызов делегата
Console.Write (result);  // 9

int Square (int x) => x * x;
```

Вызов делегата очень схож с вызовом метода (так как целью делегата является всего лишь обеспечение определенного уровня косвенности):

```
t(3);
```

Инструкция `Transformer t = Square;` представляет собой сокращение следующей инструкции:

```
Transformer t = new Transformer (Square);
```

Точно так же `t(3)` — сокращение вызова

```
t.Invoke (3);
```

Делегат схож с *обратным вызовом* — общий термин, который охватывает конструкции, такие как указатели на функции языка программирования C.

Написание подключаемых методов с помощью делегатов

Метод присваивается переменной делегата во время выполнения. Это удобно, когда нужно писать 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.

```
int[] values = { 1, 2, 3 };
Transform(values, Square);   // Подключение метода Square

foreach (int i in values)
    Console.Write (i + " "); // 1 4 9

void Transform(int[] values, Transformer t)
{
    for (int i = 0; i < values.Length; i++)
        values[i] = t (values[i]);
}

int Square (int x) => x * x;

delegate int Transformer(int x);
```

Целевые методы экземпляра и целевые статические методы

Целевой метод делегата может быть локальным, статическим или методом экземпляра.

Когда объекту делегата назначается *метод экземпляра*, последний должен поддерживать ссылку не только на метод, но и на экземпляр, которому принадлежит метод. Свойство `System.Delegate` класса `Target` представляет этот экземпляр (и является `null` для делегата, ссылающегося на статический метод).

Групповые делегаты

Все экземпляры делегатов обладают возможностью *группового вызова* (multicast). Это означает, что экземпляр делегата может ссылаться не только на одиночный целевой метод, но и на список целевых методов. Экземпляры делегатов объединяются с помощью операторов + 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.

Операторы `-` и `-=` удаляют правый операнд делегата из левого операнда делегата. Например:

```
d -= SomeMethod1;
```

Обращение к `d` теперь приведет к вызову только одного метода — `SomeMethod2()`.

Применение оператора + или += к переменной делегата со значением `null` законно, как и применение оператора `-=` к переменной делегата с единственным целевым методом (в результате чего экземпляр делегата получает значение `null`).

ПРИМЕЧАНИЕ

Делегаты являются *неизменяемыми*, так что при использовании оператора += или `-=` фактически создается *новый* экземпляр делегата, который и присваивается существующей переменной.

Если групповой делегат имеет возвращаемый тип, отличающийся от `void`, то вызывающий компонент получает возвращаемое значение от последнего вызванного метода. Предшествующие методы по-прежнему вызываются, но их возвращаемые значения игнорируются. В большинстве сценариев групповые делегаты

имеют возвращаемые типы `void`, поэтому в них такая ситуация не возникает.

Все типы делегатов неявно порождены от класса `System.MulticastDelegate`, который унаследован от `System.Delegate`. Операторы `+`, `-`, `+=` и `-=`, применяемые к делегату, транслируются в вызовы статических методов `Combine()` и `Remove()` класса `System.Delegate`.

Обобщенные типы делегатов

Тип делегата может содержать параметры обобщенного типа. Например:

```
public delegate T Transformer<T> (T arg);
```

Ниже показано, как можно использовать этот тип делегата:

```
Transformer<double> s = Square;
Console.WriteLine(s(3.3));    // 10.89
double Square(double x) => x * x;
```

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 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 до T16
```

Представленные делегаты исключительно универсальны. Делегат `Transformer` из предыдущего примера 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 охватывают лишь те практические сценарии, которые связаны с параметрами `ref`/`out` и параметрами-указателями.

Совместимость делегатов

Все типы делегатов несовместимы друг с другом, даже если имеют одинаковые сигнатуры:

```
delegate void D1(); delegate void D2();
...
D1 d1 = Method1;
D2 d2 = d1;   // Ошибка времени компиляции
```

Тем не менее следующий код разрешен:

```
D2 d2 = new D2(d1);
```

Экземпляры делегатов считаются равными, если они имеют одинаковые типы и целевые методы. Для групповых делегатов важен порядок следования целевых методов.

Вариантность возвращаемых типов

В результате вызова метода можно получить обратно тип, который является более конкретным, чем запрошенный. Это обычное полиморфное поведение. В соответствии с таким поведением целевой метод делегата может возвращать более конкретный тип, чем описанный делегатом. Это *ковариантность*:

```
ObjectRetriever o = new ObjectRetriever (RetriveString);
object result = o();
Console.WriteLine(result); // hello
```

```
string RetriveString() => "hello";
delegate object ObjectRetriever();
```

Делегат `ObjectRetriever` ожидает возврата `object`, но годится также *подкласс* `object`, потому что возвращаемые типы делегатов являются *ковариантными*.

Вариантность параметров

При вызове метода можно предоставлять аргументы, которые имеют более конкретный тип, чем параметры данного метода. Это обычное полиморфное поведение. В соответствии с таким поведением у целевого метода делегата могут быть менее конкретные типы параметров, чем описанные самим делегатом. Это *контравариантность*:

```
StringAction sa = new StringAction(ActOnObject);
sa ("hello");

void ActOnObject (object o) => Console.WriteLine(o);
delegate void StringAction(string s);
```

ПРИМЕЧАНИЕ

Стандартный шаблон событий спроектирован так, чтобы помочь 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” было указано, что параметры типа для обобщенных интерфейсов могут быть ковариантными и контравариантными. Аналогичная возможность существует и для обобщенных делегатов. При определении обобщенного типа делегата рекомендуется поступать следующим образом:

- ✓ маркировать параметр типа, используемый только в качестве возвращаемого значения, как ковариантный (`out`);

- ✓ маркировать любой параметр типа, используемый только в параметрах, как контравариантный (`in`).

Это позволяет преобразованиям работать естественным образом, соблюдая отношения наследования между типами. Делегат

```
delegate TResult Func<out TResult>();
```

(определенный в пространстве имен `System`) является ковариантным для `TResult`, разрешая следующую запись:

```
Func<string> x = ...;
Func<object> y = x;
```

Делегат

```
delegate void Action<in T> (T arg);
```

(определенный в пространстве имен `System`) является контравариантным для `T`, разрешая следующую запись:

```
Action<object> x = ...;
Action<string> y = x;
```

События

Когда используются делегаты, обычно возникают две независимые роли: *ретранслятор* (broadcaster) и *подписчик* (subscriber). 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, применяя операторы `+=` и `-=` для делегата 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 одного на другой*.

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 рассматривать его как делегат. Код за пределами `Broadcaster` может только выполнять операторы `+=` и `-=` для события `PriceChanged`.

В следующем примере класс `Stock` запускает свое событие `PriceChanged` всякий 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;
            // Запуск события, если
            // список вызовов не пуст:
            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 такими же. Однако класс `Stock` станет менее надежным в том плане, что подписчики смогут предпринимать следующие действия, влияя один на другого:

- ✓ заменять других подписчиков, переустанавливая `PriceChanged` (вместо применения оператора `+=`);
- ✓ очищать всех подписчиков (путем установки `PriceChanged` в `null`);
- ✓ выполнять групповую рассылку другим подписчикам, вызывая делегат.

События могут быть виртуальными, перекрытыми, абстрактными или запечатанными. Они также могут быть статическими.

Стандартный шаблон событий

Почти во всех случаях, когда события определяются в библиотеке .NET, 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 null:
  PriceChanged?.Invoke(this, e);

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
        OnPriceChanged(new PriceChangedEventArgs(price,
                                                 value));
      price = value;
    }
  }
}
```

В ядре стандартного шаблона событий находится предопределенный класс .NET `System.EventArgs` без членов, не считая статического свойства `Empty`. `EventArgs` является базовым классом для передачи информации событию. В данном примере мы создаем подкласс `EventArgs` для передачи старых и новых цен при генерации события `PriceChanged`.

Обобщенный делегат `System.EventHandler` также является частью .NET и определен следующим образом:

```
public delegate void EventHandler<TEventArgs>
        (object source, TEventArgs e)
        where TEventArgs : EventArgs;
```

ПРИМЕЧАНИЕ

До выхода версии C# 2.0 (в которой в язык были добавлены обобщения) решение предусматривало написание специального делегата обработки событий для каждого типа `EventArgs` следующим образом:

```
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), а 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, рост цены 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 инстанцировать `EventArgs`.

Средства доступа к событию

Средства доступа к событию представляют собой реализации его операторов `+=` и `-=`. По умолчанию средства доступа неявно реализуются компилятором. Взгляните на следующее объявление события:

```
public event EventHandler PriceChanged;
```

Компилятор преобразует его в перечисленные ниже компоненты:

- ✓ закрытое поле делегата;
- ✓ пара открытых функций доступа, реализации которых перенаправляют операторы `+=` и `-=` к закрытому полю делегата.

Контроль над этим процессом можно взять на себя, определив *явные* средства доступа. Вот как выглядит реализация события `PriceChanged` из предыдущего примера вручную:

```
EventHandler priceChanged; // Закрытый делегат
public event EventHandler PriceChanged
{
    add    { priceChanged += value; }
    remove { priceChanged -= value; }
}
```

Приведенный пример функционально идентичен реализации средств доступа C# по умолчанию (с тем отличием, что компилятор C# гарантирует также безопасность в отношении потоков во время обновления делегата). Определяя средства доступа к событию самостоятельно, мы указываем, что генерировать поле по умолчанию 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 безымянный 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. Это делает возможной интерпретацию лямбда-выражения позже, во время выполнения (весь процесс подробно описан в главе 8 книги *C# 9.0. Справочник. Полное описание языка*).

В следующем примере `x => x*x` представляет собой лямбда-выражение:

```
Transformer sqr = x => x * x;
Console.WriteLine(sqr(3));       // 9

delegate int Transformer(int i);
```

ПРИМЕЧАНИЕ

Внутренне компилятор преобразует лямбда-выражение такого типа в закрытый метод и помещает в его тело код выражения.

Лямбда-выражение имеет следующий вид:

`(`*параметры*`) => `*выражение-или-блок-инструкций*

Для удобства круглые скобки можно опускать тогда и только тогда, когда имеется в точности один параметр выводимого типа.

В нашем примере единственным параметром является `x`, а выражением — `x*x`:

```
x => x * x;
```

Каждый параметр лямбда-выражения соответствует параметру делегата, а тип выражения (который 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 инструкций, 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;
```

Обычно компилятор способен *вывести* тип параметров 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 ("hello", "world"); // total = 10
```

Предполагая, что `Clicked` — это событие типа `EventHandler`, следующий код присоединяет обработчик события через лямбда-выражение:

```
obj.Clicked += (sender,args) => Console.Write("Click");
```

Захват внешних переменных

Лямбда-выражение может обращаться к любым переменным, доступным в момент определения лямбда-выражения. Такие переменные называются *внешними переменными* и включают локальные переменные, параметры и поля. Например:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine(multiplier(3));        // 6
```

Внешние переменные, к которым обращается лямбда-выражение, называются *захваченными переменными* (captured). Лямбда-выражение, которое захватывает переменные, называется *замыканием* (closure). Захваченные переменные вычисляются при фактическом *вызове* делегата, а не в момент *захвата*:

```
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 собственное время жизни, расширенное до времени жизни делегата. В следующем примере локальная переменная `seed` должна бы покидать область видимости после того, как выполнение `Natural` завершено. Но поскольку переменная `seed` была *захвачена*, ее время жизни расширяется до времени жизни захватившего ее делегата `natural`:

```
Func<int> natural = Natural();
Console.WriteLine(natural());  // 0
Console.WriteLine(natural());  // 1

static Func<int> Natural()
{
    int seed = 0;
    return () => seed++; // Возвращает замыкание
}
```

ПРИМЕЧАНИЕ

Переменные могут также быть захвачены анонимными и локальными методами. В этих ситуациях правила для захваченных переменных такие же.

Статические лямбда-выражения (C# 9)

Начиная с C# 9, можно гарантировать, что лямбда-выражение, локальная функция или анонимный метод не захватывает состояние, применяя ключевое слово `static`. Это может быть полезно в сценариях микрооптимизации для предотвращения (потенциально непреднамеренного) выделения памяти и очистки замыкания. Например, мы можем применить модификатор `static` к следующему лямбда-выражению:

```
Func<int,int> multiplier = static n => n * 2;
```

Если позже мы попытаемся изменить лямбда-выражение так, чтобы оно захватывало локальную переменную, компилятор сгенерирует ошибку. Эта функция более полезна в локальных методах (поскольку лямбда-выражение само требует выделения памяти). В следующем примере метод `Multiply` не может получить доступ к переменной `factor`:

```
void Foo ()
{
    int factor = 123;
    static int Multiply (int x) => x * 2;
}
```

Применение `static` здесь, возможно, полезно также в качестве инструмента документации, указывающего на пониженный уровень связывания. Статические лямбда-выражения все еще могут обращаться к статическим переменным и константам (потому что они не требуют замыкания).

ПРИМЕЧАНИЕ

Ключевое слово `static` действует просто как *проверка*; оно никак не влияет на код на промежуточном языке, который генерирует компилятор. Без ключевого слова `static` компилятор не генерирует замыкание, если только оно не является необходимым (но даже в этом случае у него есть уловки, позволяющие упростить код и сделать его менее дорогим).

Захват 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 *одна и та же* 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 (выделено полужирным) захватывает одну и ту же переменную `i`. (Это действительно имеет смысл, если рассматривать переменную `i` как переменную, значение которой сохраняется между итерациями цикла; при желании `i` можно даже явно изменять внутри тела цикла.) В результате, когда позже вызываются делегаты, они видят значение `i` в момент *вызова*, т.е. `3`. Если же требуется вывести на экран `012`, то решение состоит в присваивании значения итерационной переменной некоторой 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*.

Обратите внимание, что (начиная с C# 5) переменная итерации в цикле `foreach` является неявно локальной, поэтому вы можете безопасно захватывать ее без необходимости прибегать ко временной переменной.

Сравнение лямбда-выражений и локальных методов

Функциональность локальных методов (см. раздел “Локальные методы”) перекрывается с функциональностью лямбда-выраже-

ний. 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.

Однако 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 схож с лямбда-выражением, с тем отличием, что он лишен неявно типизированных параметров, синтаксиса выражений (анонимный метод всегда должен быть блоком инструкций) и возможности компиляции в дерево выражения. Чтобы написать анонимный метод, требуется указать ключевое слово `delegate`, затем — (необязательное) объявление параметра и наконец — тело метода. Например:

```
Transformer sqr = delegate(int x) {return x * x;};
Console.WriteLine(sqr(3));      // 9

delegate int Transformer(int i);
```

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 с пустым обработчиком по умолчанию:

```
public event EventHandler Clicked = delegate { };
```

В итоге устраняется необходимость проверки на равенство `null` перед запуском события. Приведенный далее код также допустим (обратите внимание на отсутствие параметров):

```
Clicked += delegate { Console.Write("clicked"); };
```

Анонимные методы захватывают внешние переменные тем же способом, что и лямбда-выражения.

Инструкции `try` и исключения

Инструкция `try` определяет 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 в блоке `try` генерируется исключение. Блок `finally` выполняется после того, как поток управления покидает блок `try` (или блок `catch`, если таковой присутствует), обеспечивая очистку независимо от того, было сгенерировано исключение или нет.

Блок `catch` имеет доступ к объекту `Exception`, который содержит информацию о происшедшей ошибке. Блок `catch` применяется либо для обработки ошибки, либо для *повторной генерации* исключения. Исключение генерируется повторно, если, например, нужно просто зарегистрировать факт возникновения проблемы в журнале или если необходимо сгенерировать исключение нового типа более высокого уровня.

Блок `finally` увеличивает степень детерминизма программы тем, что выполняется несмотря ни на что. Он полезен для выполнения задач очистки наподобие закрытия файлов или сетевых подключений.

Инструкция `try` выглядит следующим образом:

```
try
{
    ... // Выполнение кода в этом блоке может
```

```
        // генерировать исключение
}
catch (ExceptionA ex)
{
    ... // Обработка исключений типа ExceptionA
}
catch (ExceptionB ex)
{
    ... // Обработка исключений типа ExceptionB
}
finally
{
    ... // Код очистки
}
```

Рассмотрим следующий 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 следующим образом:

```
try
{
    int x = 3, y = 0;
    Console.WriteLine(x / y);
}
catch (DivideByZeroException ex)
{
    Console.Write("y не может быть нулем.");
}
// После исключения выполнение продолжается...
```

ПРИМЕЧАНИЕ

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 внутри инструкции `try` сгенерировано исключение, среда CLR выполняет следующую проверку.

Имеет ли инструкция `try` совместимые с исключением блоки `catch`?

- ✓ Если имеет, то управление переходит соответствующему блоку `catch`, затем — блоку `finally` (при наличии такового) и далее выполнение продолжается обычным образом.
- ✓ Если не имеет, управление переходит прямо блоку `finally` (при наличии такового), а затем 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 обработку исключения, то пользователю отображается диалоговое окно с сообщением об ошибке и программа прекращает работу.

Конструкция `catch`

Конструкция `catch` указывает тип исключения, подлежащего перехвату. Этим 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 конкретных 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 конкретного исключения выполняется только одна конструкция `catch`. Если вы хотите предусмотреть сетку безопасности для перехвата общих исключений (типа `System.Exception`), размещайте более конкретные обработчики первыми.

Исключение можно перехватывать без указания переменной, если доступ к свойствам исключения не нужен:

```
catch (OverflowException) // Без переменной
{ ... }
```

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`, то вычисляется логическое выражение, находящееся после ключевого слова `when`. Если результатом вычисления оказывается `false`, то данный блок `catch` игнорируется и просматриваются (при их наличии) последующие конструкции `catch`. Благодаря фильтрам исключений может появиться смысл в повторном перехвате исключения одного и того же типа:

```
catch (WebException ex)
    when (ex.Status == некоторое-значение)
{ ... }
catch (WebException ex)
```

```
    when (ex.Status == некоторое-иное-значение)
{ ... }
```

Логическое выражение в конструкции `when` может иметь побочные действия, например вызывать метод, который регистрирует в журнале сведения об исключении для целей диагностики.

Блок `finally`

Блок `finally` выполняется всегда — независимо от того, было ли сгенерировано 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` повышает детерминизм 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 генерируется исключение `IOException`.

Вот пример:

```
static void ReadFile()
{
    StreamReader reader = null; // В пространстве
                                // имен System.IO
    try
    {
        reader = File.OpenText ("file.txt");
        if (reader.EndOfStream) return;
        Console.WriteLine(reader.ReadToEnd());
    }
```

```
    finally
    {
        if (reader != null) reader.Dispose();
    }
}
```

В этом примере мы закрываем файл с помощью вызова `Dispose()` для `StreamReader`. Вызов `Dispose()` для объекта внутри блока `finally` представляет собой стандартное соглашение, соблюдаемое повсеместно в .NET, и оно явно поддерживается в языке C# с использованием инструкции `using`.

Инструкция `using`

Многие классы инкапсулируют неуправляемые ресурсы, такие как дескрипторы файлов или графические дескрипторы, или подключения к базам данных. Классы подобного рода реализуют интерфейс `System.IDisposable`, в котором определен единственный метод без параметров с именем `Dispose()`, предназначенный для очистки этих ресурсов. Инструкция `using` предлагает элегантный синтаксис для вызова `Dispose()` для объекта `IDisposable` внутри блока `finally`.

Инструкция

```
using (StreamReader reader = File.OpenText("file.txt"))
{
    ...
}
```

в точности эквивалентна следующему коду:

```
{
  StreamReader reader = File.OpenText("file.txt");
  try
  {
    ...
  }
  finally
  {
    if (reader != null) ((IDisposable)reader).Dispose();
  }
}
```

Объявления `using`

Если опустить круглые скобки и блок инструкций, следующий за инструкцией `using`, то можно получить *объявление `using`*. Такой ресурс освобождается, когда поток управления выходит за пределы *охватывающего* блока инструкций:

```
if (File.Exists ("file.txt"))
{
    using var reader = File.OpenText("file.txt");
    Console.WriteLine(reader.ReadLine());
    ...
}
```

В данном случае `reader` будет освобожден после того, как поток управления выйдет за пределы блока инструкций `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, `throw` может появляться как выражение в функциях, сжатых до выражения

```
public string Foo() =>
    throw new NotImplementedException();
```

Выражение `throw` может также находиться внутри тернарного условного оператора:

```
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)
{
    // Запись в журнал...
    ...
    throw; // Генерация того же исключения
}
```

Повторная генерация в подобной манере дает возможность зарегистрировать в журнале информацию об ошибке, не подавляя ее. Она позволяет также отказаться от обработки исключения, если обстоятельства сложились не так, как ожидалось.

ПРИМЕЧАНИЕ

Если `throw` заменить выражением `throw ex`, то пример сохранит работоспособность, но свойство `StackTrace` исключения больше не будет отражать информацию об исходной ошибке.

Еще один распространенный сценарий предусматривает повторную генерацию исключения более конкретного или значащего типа:

```
try
{
    ... // Получения даты из элемента данных XML
}
catch(FormatException ex)
{
    throw new XmlException ("Некорректная дата", ex);
}
```

При повторной генерации другого исключения в свойстве `InnerException` можно указать исходное исключение, чтобы помочь в отладке. Почти все типы исключений предоставляют конструктор для данной 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 таковое имеется), которое привело к генерации внешнего исключения. Само это свойство, в свою очередь, может иметь свойство InnerException, отличающееся от данного.

Перечисление и итераторы

Перечисление

Перечислитель — это однонаправленный, предназначенный только для чтения курсор по *последовательности значений*. C# рассматривает тип как перечислитель, если он соответствует одному пункту из следующего списка.

- ✓ Имеет открытый метод без параметров с именем MoveNext и свойство с именем Current.
- ✓ Реализует интерфейс System.Collections.IEnumerator.
- ✓ Реализует интерфейс System.Collections.Generic.IEnumerator<T>.

Инструкция foreach выполняет итерацию по *перечислимому* объекту. Перечислимый объект является логическим представлением последовательности. Это не собственно курсор, а объект, который производит курсор для себя самого. C# рассматривает тип как перечислимый, если он соответствует любому пункту из

следующего списка (проверка соответствия выполняется в указанном порядке).

- ✓ Имеет открытый метод без параметров с именем `GetEnumerator`, который возвращает перечислитель.
- ✓ Реализует интерфейс `System.Collections.Generic.IEnumerable<T>`.
- ✓ Реализует интерфейс `System.Collections.IEnumerable`.
- ✓ (Начиная с C# 9) может быть связан с *расширяющим методом*, названным `GetEnumerator`, который возвращает перечислитель (см. раздел "Расширяющие методы").

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() {...}
}
```

Вот как выглядит высокоуровневый способ перебора символов в слове *beer* с использованием инструкции `foreach`:

```
foreach(char c in "beer") Console.WriteLine(c);
```

А вот низкоуровневый способ перебора символов в том же слове без использования инструкции `foreach`:

```
using (var enumerator = "beer".GetEnumerator())
while (enumerator.MoveNext())
{
    var element = enumerator.Current;
    Console.WriteLine (element);
}
```

Если перечислитель реализует интерфейс `IDisposable`, инструкция `foreach` действует также как оператор `using`, неявно удаляя 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 последнюю строку в следующий код:

```
List<int> list = new List<int>();
list.Add (1); list.Add (2); list.Add (3);
```

Для этого 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,  "five" },
    { 10, "ten" }
};
```

Или более кратко:

```
var dict = new Dictionary<int, string>()
{
    [5]  = "five",
    [10] = "ten"
};
```

Второй вариант записи допустим не только со словарями, но и с любым типом, для которого существует индексатор.

Итераторы

В то время как инструкцию `foreach` можно рассматривать как *потребитель* перечислителя, итератор является *производителем* перечислителя. В приведенном ниже примере итератор используется для возврата последовательности чисел Фибоначчи (в которой каждое число является суммой двух предыдущих чисел):

```
foreach (int fib in Fibs (6))
    Console.Write (fib + " ");

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` означает "Вот значение, которое должно быть возвращено из этого метода", оператор `yield return` означает "Вот следующий элемент, который должен быть выдан этим перечислителем". Когда встречается инструкция `yield`, управление возвращается вызывающему коду, но состояние вызываемого метода сохраняется, так что этот метод может продолжить свое выполнение, как только вызывающий код переходит к следующему элементу. Жизненный цикл состояния ограничен перечислителем, поэтому состояние может быть освобождено, когда вызывающий код завершит перечисление.

ПРИМЕЧАНИЕ

Компилятор преобразует методы итератора в закрытые классы, которые реализуют интерфейсы `IEnumerable<T>` и/или `IEnumerator<T>`. Логика внутри блока итератора "инвертируется", после чего соединяется с методом `MoveNext()` и свойством `Current` класса перечислителя, сгенерированного компилятором. Это означает, что при вызове метода все, что происходит, — это инстанцирование сгенерированного компилятором класса; никакой написанный вами код в действительности не выполняется! Ваш код выполняется, только когда начинается перечисление результирующей последовательности, обычно с помощью инструкции `foreach`.

Семантика итератора

Итератор представляет собой метод, свойство или индексатор, который содержит один или большее количество инструкций `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 интерфейс *перечислителя*, как правило, используются реже. Они удобны при написании пользовательского класса коллекции: обычно вы назначаете итератору имя `GetEnumerator` и обеспечиваете реализацию классом интерфейса `IEnumerable<T>`.

Итераторы, возвращающие интерфейс *перечислимого* типа, являются более распространенными и более простыми в использовании, так как вам не приходится писать класс коллекции. За кулисами компилятор генерирует закрытый класс, реализующий `IEnumerable<T>` (а также `IEnumerator<T>`).

Несколько инструкций `yield`

Итератор может включать несколько инструкций `yield`:

```
foreach (string s in Foo())
    Console.Write (s + " ");  // One Two Three

IEnumerable<string> Foo()
{
    yield return "One";
    yield return "Two";
    yield return "Three";
}
```

Инструкция `yield break`

В блоке итератора `return` не допускается; взамен вы обязаны использовать конструкцию `yield break`, указывающую, что блок итератора должен быть завершен преждевременно, и больше эле-

ментов не возвращать. Чтобы продемонстрировать работу, изменим метод `Foo()`, как показано ниже:

```
IEnumerable<string> Foo (bool breakEarly)
{
    yield return "One";
    yield return "Two";
    if (breakEarly) yield break;
    yield return "Three";
}
```

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 будут выводиться 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.

Возможность компоновки, поддерживаемая шаблоном итератора, крайне важна при построении запросов LINQ.

Типы-значения, допускающие `null`

Ссылочные типы могут представлять несуществующее значение с помощью ссылки `null`. Однако типы-значения не способны представлять значения `null` обычным образом. Например:

```
string s = null; // OK, ссылочный тип
int i = null;    // Ошибка времени компиляции -
                 // int не может быть null
```

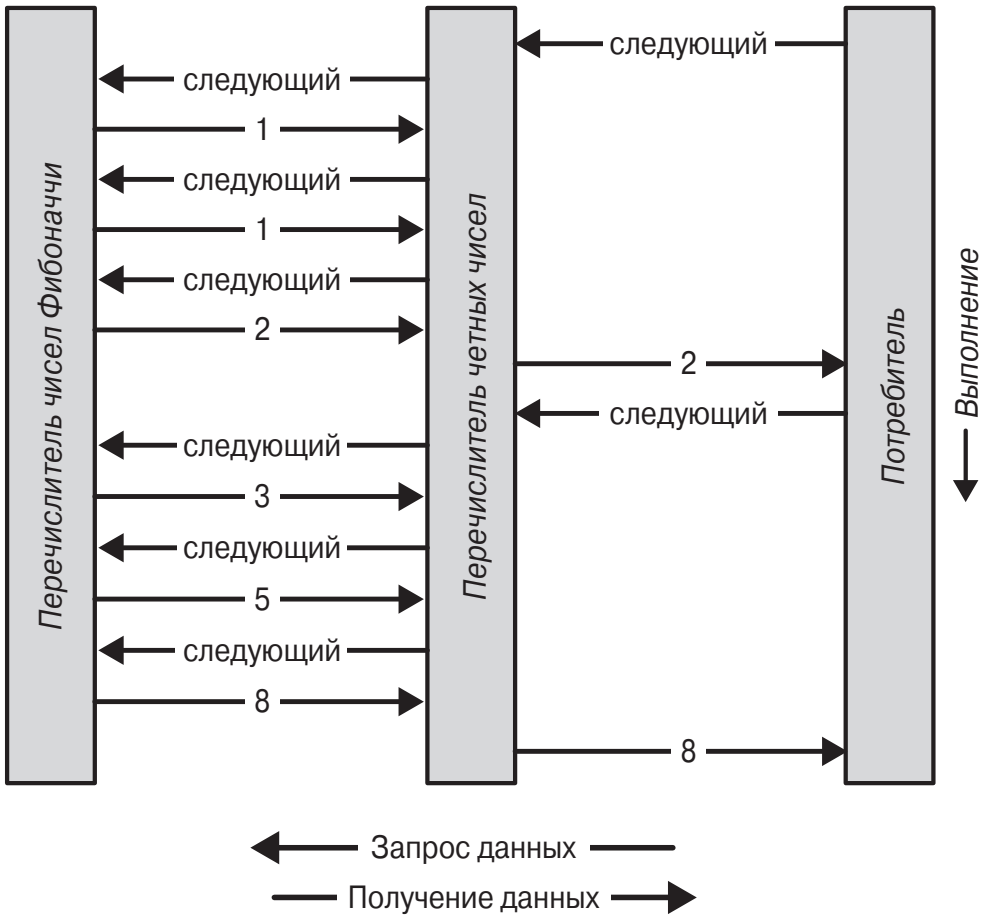


Рис. 5. *Компоновка последовательностей*

Чтобы представить `null` с помощью типа-значения, необходимо использовать специальную конструкцию, которая называется типом, *допускающим значение* `null` (nullable). Тип, допускающий значение `null`, обозначается как тип-значение, за которым следует символ ?:

```
int? i = null;        // OK, может принимать 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 значение по умолчанию — в противном случае.

Значением `T?` по умолчанию является `null`.

Преобразования типов, допускающих значение null

Преобразование из `T` в `T?` является неявным, в то время как из `T?` в `T` — явным. Например:

```
int? x = 5;      // Неявное
int y = (int)x;  // Явное
```

Явное приведение полностью эквивалентно вызову свойства `Value` объекта типа, допускающего `null`. Следовательно, если `HasValue` равно `false`, то генерируется исключение `InvalidOperationException`.

Упаковка и распаковка типов-значений, 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 оператора as. Если приведение неуспешно, то результатом будет null:

```
object o = "string";
int? x = o as int?;
Console.WriteLine(x.HasValue); // False
```

Подъем операторов

В структуре Nullable<T> не определены такие операторы, 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", оператор "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 с помощью оператора "меньше чем" типа int; в противном случае результатом будет false.

Подъем операторов означает возможность неявного использования операторов типа T с типом T?. Вы можете определить операторы 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.

Компилятор выполняет логику в отношении null в зависимости от категории оператора.

Операторы эквивалентности (== и !=)

Поднятые операторы эквивалентности обрабатывают значения null точно так же, как и ссылочные типы. Это означает, что два значения null равны:

```
Console.WriteLine(       null ==        null); //True
Console.WriteLine((bool?)null == (bool?)null); //True
```

Кроме того:

- ✓ если ровно один операнд имеет значение `null`, то операнды не равны;
- ✓ если оба операнда отличны от `null`, то сравниваются их свойства `Value`.

Операторы отношения (<, <=, >=, >)

Работа операторов 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);
```

Остальные операторы

Остальные операторы (+, -, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~) возвращают `null`, когда любой из операндов равен `null`. Такой шаблон должен быть хорошо знаком пользователям SQL.

```
int? c = x + y; // Транслируется в:

int? c = (x == null || y == null)
           ? null
           : (int?) (x.Value + y.Value);
```

Исключением является ситуация, когда операторы & и | применяются к `bool?`, что мы вскоре обсудим.

Смешивание в операторах 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 операторы & и |

Когда операнды имеют тип bool?, операторы & и | трактуют null как неизвестное значение. Таким образом, null|true дает true по следующим причинам:

- ✓ если неизвестное значение равно false, результатом будет true;
- ✓ если неизвестное значение равно true, результатом будет true.

Аналогично 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 операторы для работы с null

Типы, допускающие значение null, особенно хорошо работают с оператором ?? (см. раздел “Оператор объединения с null”). 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 оператора ?? эквивалентно вызову GetValueOrDefault() с явным значением по умолчанию за исключением

того, что выражение для значения по умолчанию никогда не вычисляется, если переменная не равна `null`.

Типы, допускающие значение `null`, также удобно применять с `null`-условным оператором (см. раздел "null-условный оператор"). В следующем примере переменная `length` получает значение `null`:

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

Скомбинировав этот код и оператор объединения с `null`, переменной `length` можно присвоить значение 0 вместо `null`:

```
int length = sb?.ToString().Length ?? 0;
```

Ссылочные типы, допускающие значение `null`

В то время как *типы-значения, допускающие `null`*, привносят поддержку `null` в типы-значения, *ссылочные типы, допускающие значение `null`* (*nullable-типы*; начиная с C# 8), делают противоположное и обеспечивают (в определенной степени) поддержку ссылочными типами невозможности значений `null`, чтобы помочь избегать исключений `NullReferenceException`.

Такие ссылочные типы позволяют введение уровня безопасности, который обеспечивается самим компилятором в форме предупреждений, когда он обнаруживает код, подвергающийся риску генерации исключения `NullReferenceException`[1].

Чтобы включить в компиляторе ссылочные типы, допускающие `null`, можно добавить элемент `Nullable` в файл проекта `.csproj` (если они нужны для всего проекта):

```
<Nullable>enable</Nullable>
```

[1] Говоря проще, начиная с C# 8 предусмотрена *глобальная* возможность заставить ссылочные типы вести себя по отношению к значению `null` так же, как и типы-значения, не допуская значений `null` и у ссылочных типов. Тем самым становится ненужной проверка на равенство значения ссылочного типа `null`. Однако поскольку такое присваивание значения `null` все же может оказаться в некоторых ситуациях важным, для этого случая — возможности присваивания `null` ссылочным типам при глобальном запрете такой возможности — и введены ссылочные типы, допускающие `null`. Без такого глобального запрета их существование особого смысла не имеет. — *Примеч. ред.*

Кроме того, в тех местах кода, где нужны такие типы, можно использовать следующие директивы:

```
#nullable enable  // Включение ссылочных nullable-типов,
                  // начиная с данной точки кода
#nullable disable // Отключение ссылочных nullable-типов,
                  // начиная с данной точки кода
#nullable restore // Сброс настройки ссылочных
                  // nullable-типов к настройкам проекта
```

После директивы `#nullable enable` компилятор делает поддержку запрета значения `null` принятой по умолчанию: если необходимо, чтобы ссылочный тип получал значения `null`, к нему придется применить суффикс `?` для указания ссылочного типа, допускающего `null`. В показанном ниже примере `s1` не допускает значения `null`, тогда как `s2` допускает:

```
#nullable enable   // Включение ссылочных nullable-типов,
                   // начиная с данной точки кода

string s1 = null;  // Генерация компилятором предупреждения
string? s2 = null; // OK: s2 допускает значение null
```

ПРИМЕЧАНИЕ

Поскольку ссылочные типы, допускающие `null`, являются конструкциями времени компиляции, во время выполнения между `string` и `string?` нет никаких различий. Типы-значения же, допускающие `null`, привносят в систему типов нечто 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, либо кодом 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-терпимый* (*null-forgiving*) оператор (`!`):

```
void Foo(string? s) => Console.Write(s!.Length);
```

Подобное применение `null`-терпимого оператора опасно тем, что мы можем в итоге получить то же исключение `NullReferenceException`, которого в первую очередь пытались избежать. Вот как можно исправить эту ситуацию:

```
void Foo (string? s)
{
    if (s != null) Console.Write(s.Length);
}
```

Обратите внимание, что теперь null-терпимый оператор не нужен. Дело 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 исключение `NullReferenceException` при нем возникнуть не может.

Способность компилятора к обнаружению и предупреждению отнюдь не безупречна, к тому же существуют пределы того, что она может охватить. Например, компилятор не имеет возможности узнать, был ли заполнен массив элементами, а потому показанный ниже код к выдаче предупреждения не приводит:

```
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("Kiev".IsCapitalized());
```

Вызов расширяющего метода при компиляции транслируется в обычный вызов статического метода:

```
Console.Write(StringHelper.IsCapitalized("Kiev"));
```

Интерфейсы также можно расширять:

```
public static T First<T> (this IEnumerable<T> sequence)
{
    foreach (T element in sequence)
        return element;
    throw new InvalidOperationException("Элементов нет!");
}
...
Console.WriteLine ("Kiev".First()); // K
```

Цепочки расширяющих методов

Как и методы экземпляра, расширяющие методы предлагают аккуратный способ 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 `"Moscow"`, но `x` использует расширяющие методы, тогда как `y` — статические:

```
string x = "Moscow".Pluralize().Capitalize();

string y = StringHelper.Capitalize
         (StringHelper.Pluralize ("Moscow"));
```

Неоднозначность и разрешение

Любой совместимый метод экземпляра всегда будет иметь преимущество над расширяющим методом — даже когда параметры расширяющего метода дают более точное соответствие типам.

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 точно соответствующие аргументы, то 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 = "Bob", Age = 1 };
```

Компилятор преобразует данное объявление в закрытый вложенный тип со свойствами только для чтения с именами `Name` (типа `string`) и `Age` (типа `int`). Для ссылки на анонимный тип должно использоваться ключевое слово `var`, так как имя этого типа генерируется компилятором.

Имя свойства анонимного типа может быть выведено из выражения, которое само по себе является идентификатором. Таким образом, код

```
int Age = 1;
var dude = new { Name = "Bob", Age };
```

эквивалентен следующему коду:

```
var dude = new { Name = "Bob", Age = Age };
```

Можно создавать массивы анонимных типов, как показано далее:

```
var dudes = new[]
{
    new { Name = "Bob",  Age = 30 },
```

```
    new { Name = "Mary", Age = 40 }
};
```

Анонимные типы главным образом используются при написании запросов LINQ.

Кортежи

Подобно анонимным типам кортежи (C# 7+) предлагают простой способ хранения набора значений. Главная цель кортежей — безопасный возврат множества значений из метода, не прибегая к параметрам `out` (то, что невозможно делать с помощью анонимных типов). Простейший способ создать *литеральный кортеж* — указать в круглых скобках список желаемых значений. В результате создается кортеж с *безымянными* элементами:

```
var bob = ("Bob", 23);
Console.WriteLine(bob.Item1);      // Bob
Console.WriteLine(bob.Item2);      // 23
```

В отличие от анонимных типов применять ключевое слово `var` необязательно и *тип кортежа* можно указывать явно:

```
(string,int) bob = ("Bob", 23);
```

Это означает, что кортеж можно успешно возвращать из метода:

```
(string,int) person = GetPerson();
Console.WriteLine(person.Item1);  // Bob
Console.WriteLine(person.Item2);  // 23

(string,int) GetPerson() => ("Bob", 23);
```

Кортежи хорошо сочетаются с обобщениями, так что все следующие типы корректны:

```
Task<(string,int)>
Dictionary<(string,int),Uri>
IEnumerable<(int ID, string Name)> // См. далее...
```

Кортежи являются *типами-значениями* с *изменяемыми* (предназначенными для чтения/записи) элементами. Таким образом, после создания кортежа можно модифицировать его элементы `Item1`, `Item2` и т.д.

Именование элементов кортежа

При создании литеральных кортежей элементам можно дополнительно назначать содержательные имена:

```
var tuple = (Name:"Bob", Age:23);
Console.WriteLine(tuple.Name);  // Bob
Console.WriteLine(tuple.Age);   // 23
```

То же самое разрешено делать при указании *типов кортежей*:

```
static (string Name, int Age) GetPerson() => ("Bob",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);     // OK
```

ПРИМЕЧАНИЕ

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"), так что кортеж легко *деконструировать* в отдельные переменные. Таким образом, взамен

```
var bob = ("Bob", 23);
string name = bob.Item1;
int age = bob.Item2;
```

можно написать

```
var bob = ("Bob", 23);
(string name, int age) = bob; // Деконструкция bob
```

```
Console.WriteLine(name);      // в name и age
Console.WriteLine(age);
```

Синтаксис деконструирования схож с синтаксисом объявления кортежа с именованными элементами, что может привести к путанице. Следующий код подчеркивает разницу между ними:

```
(string name, int age)      = bob; // Деконструкция
(string name, int age) bob2 = bob; // Объявление кортежа
```

Записи (C# 9)

Запись (record) — это особый класс, созданный для облегчения работы с неизменяемыми (предназначенными только для чтения) данными. Его самая полезная функциональная возможность — *недеструктивное изменение*, в то время как обычно, для того чтобы "модифицировать" неизменяемый объект, вы создаете новый объект и копируете в него данные, включающие ваши изменения.

Записи полезны также при создании типов, которые просто объединяют или хранят данные. В простых случаях они исключают шаблонный код и при этом соблюдают семантику *структурного равенства* (два объекта являются одинаковыми, если их данные совпадают), что обычно и требуется при работе с неизменяемыми типами.

Запись — это конструкция времени компиляции C#. Во время выполнения CLR видит записи просто как классы (с множеством дополнительных "синтезированных" членов, добавленных компилятором).

Определение записи

Определение записи схоже с определением класса и может содержать те же типы членов, включая поля, свойства, методы и т.д. Записи могут реализовывать интерфейсы и порождать другие записи (но не классы).

Простая запись может содержать один лишь набор только инициализируемых свойств и, возможно, конструктор:

```
record Point
{
    public Point (double x, double y) => (X, Y) = (x, y);
```

```
    public double X { get; init; }
    public double Y { get; init; }
}
```

После компиляции C# преобразует определение записи в класс и выполняет следующие дополнительные шаги.

- ✓ Создает защищенный *копирующий конструктор* (и скрытый метод *клонирования*) для облегчения поддержки недеструктивного изменения.
- ✓ Перекрывает/перегружает функции, связанные с проверкой равенства для реализации семантики структурного равенства.
- ✓ Перекрывает метод `ToString()` (для расширения общедоступных свойств записи, как и в случае с анонимными типами).

Предыдущее объявление записи расширяется до чего-то наподобие

```
class Point
{
    public Point (double x, double y) => (X, Y) = (x, y);

    public double X { get; init; }
    public double Y { get; init; }

    protected Point(Point original) // Копирующий
    {                               // конструктор
        this.X = original.X; this.Y = original.Y
    }

    // Метод со странным сгенерированным компилятором именем
    public virtual Point <Clone>$() => new Point (this);

    // Дополнительный код для перекрытия Equals, ==, !=,
    // GetHashCode, ToString()...
}
```

Списки параметров

Определение записи может также включать *список параметров*:

```
record Point(double X, double Y)
{
    ...
}
```

Параметры могут включать модификаторы `in` и `params`, но не `out` или `ref`. Если указан список параметров, компилятор выполняет следующие дополнительные шаги.

- ✓ Создает для каждого параметра только инициализируемое свойство.
- ✓ Создает *первичный конструктор* для заполнения свойств.
- ✓ Создает деконструктор.

Это означает, что мы можем объявить нашу запись Point просто следующим образом:

```
record Point(double X, double Y);
```

В конечном итоге компилятор сгенерирует (почти) именно то, что мы перечислили в предыдущем расширении. Незначительная разница в том, что имена параметров в первичном конструкторе в итоге будут выглядеть как `X` и `Y` вместо `x` и `y`:

```
public Point (double X, double Y)
{
    this.X = X; this.Y = Y;
}
```

ПРИМЕЧАНИЕ

Кроме того, поскольку это *первичный конструктор*, параметры `X` и `Y` волшебным образом становятся доступными для инициализаторов любого поля или свойства в вашей записи. Мы подробнее обсудим этот вопрос ниже, в разделе “Первичные конструкторы”.

Еще одно отличие при определении списка параметров заключается в том, что компилятор генерирует и деконструктор:

```
public void Deconstruct (out double X, out double Y)
{
    X = this.X; Y = this.Y;
}
```

Записи со списками параметров могут порождать подклассы с использованием следующего синтаксиса:

```
record Point3D (double X, double Y, double Z)
    : Point (X, Y);
```

В этом случае компилятор генерирует следующий первичный конструктор:

```
class Point3D : Point
{
    public double Z { get; init; }

    public Point3D (double X, double Y, double Z)
        : base (X, Y)
        => this.Z = Z;
}
```

ПРИМЕЧАНИЕ

Списки параметров представляют собой удобное сокращение, когда вам нужен класс, который просто группирует набор значений, а также может быть полезным для прототипирования. Они не так полезны, когда вам нужно добавить логику к средствам доступа `init` (например, такую как проверка аргументов).

Недеструктивное изменение

Самое важное, что компилятор выполняет со всеми записями, — это создание *копирующего конструктора* (и скрытого метода *клонирования*). Они позволяют выполнять недеструктивное изменение с использованием ключевого слова C# 9 `with`:

```
Point p1 = new Point (3, 3);
Point p2 = p1 with { Y = 4 };
Console.WriteLine (p2);         // Point { X = 3, Y = 4 }

record Point (double X, double Y);
```

В этом примере `p2` является копией `p1`, но для его свойства `Y` установлено значение 4. Чем больше имеется свойств, тем больше выгода от данного синтаксиса.

Недеструктивное изменение происходит в два этапа.

1. Сначала *копирующий конструктор* клонирует запись. По умолчанию копируется каждое из базовых полей записи, создавая точную копию и обходя любую логику в средствах доступа `init`. Включаются все поля (общедоступные, закрытые и скрытые).
2. Затем каждое свойство в *списке инициализаторов членов* обновляется (на этот раз с использованием средств доступа `init`).

Таким образом, компилятор транслирует код

```
Test t2 = t1 with { A = 10, C = 30 };
```

в нечто, функционально эквивалентное следующему коду:

```
Test t2 = new Test(t1); // Клонирование t1
t2.A = 10;              // Обновление свойства A
t2.C = 30;              // Обновление свойства C
```

(Этот код не скомпилировался бы, если бы вы написали его явно, — потому что `A` и `C` доступны только для инициализации. Кроме того, копирующий конструктор *защищен*; C# обходит эту защиту, вызывая его через общедоступный скрытый метод с именем `<Clone>$`, который он добавляет в запись.)

При необходимости вы можете определить собственный *копирующий конструктор*. C# будет в таком случае использовать ваше определение вместо того, чтобы создавать свое:

```
protected Point (Point original)
{
    this.X = original.X; this.Y = original.Y;
}
```

При наследовании другой записи копирующий конструктор отвечает только за копирование собственных полей. Чтобы скопировать поля базовой записи, выполните ее делегирование конструктору:

```
protected Point (Point original) : base (original)
{
    ...
}
```

Первичные конструкторы

Когда вы определяете запись со списком параметров, компилятор генерирует объявления свойств автоматически, как и *первичный конструктор* (primary constructor) и деконструктор. Это хорошо работает в простых случаях; в более сложных случаях вы можете опустить список параметров и написать объявления свойств и конструктор вручную. C# также предлагает умеренно полезную промежуточную возможность определения списка параметров при самостоятельном написании некоторых или всех объявлений свойств:

```
record Student(int ID, string Surname, string FirstName)
{
    public int ID { get; } = ID;
}
```

В данном случае мы "взяли на себя" определение свойства `ID`, определив, что оно доступно только для чтения (а не является только инициализируемым), что предотвращает его участие в недеструктивном изменении. Если вам никогда не потребуется недеструктивно изменять некоторое конкретное свойство, объявив его доступным только для чтения, вы кешируете вычисленные данные в записи без необходимости кодировать механизм его обновления.

Обратите внимание, что мы обязаны включить *инициализатор свойства* (полужирный шрифт):

```
public int ID { get; } = ID;
```

Взяв на себя объявление свойства, вы становитесь ответственным за инициализацию его значения; первичный конструктор больше не делает это автоматически. Обратите внимание, что **`ID`**, выделенный полужирным шрифтом, ссылается не на свойство `ID`, а на *параметр первичного конструктора*. Уникальной особенностью первичных конструкторов является то, что их параметры (в данном случае — `ID`, `Surname` и `FirstName`) волшебным образом видны всем инициализаторам полей и свойств.

Вы также можете взять на себя определение свойства с явными средствами доступа:

```
int _id = ID;
public int ID { get => _id; init => _id = value; }
```

Здесь вновь **ID**, выделенный полужирным шрифтом, ссылается на параметр первичного конструктора, а не на свойство. (Причина отсутствия неоднозначности заключается в том, что доступ к свойствам из инициализаторов не разрешен.)

Тот факт, что мы должны инициализировать свойство `_id` с помощью `ID`, делает такой "захват" менее полезным, поскольку любая логика в средстве доступа `init` (например, проверка значений) будет опущена первичным конструктором.

Записи и сравнение на равенство

Как и в случае со структурами, анонимными типами и кортежами, записи предоставляют структурное равенство "из коробки", а это означает, что две записи равны, если равны их поля (и автоматические свойства):

```
var p1 = new Point(1, 2);
var p2 = new Point(1, 2);
Console.WriteLine(p1.Equals (p2));  // True

record Point(double X, double Y);
```

Оператор равенства также в состоянии работать с записями (как и с кортежами):

```
Console.WriteLine (p1 == p2);       // True
```

В отличие от классов и структур, вы не должны (и не можете) перекрывать метод `object.Equals`, если вы хотите настроить поведение проверки на равенство. Вместо этого вы определяете общедоступный метод `Equals` со следующей сигнатурой:

```
record Point(double X, double Y)
{
    public virtual bool Equals(Point other) =>
        other != null && X == other.X && Y == other.Y;
}
```

Метод `Equals` должен быть `virtual` (не `override`!) и должен быть *строго типизирован*, чтобы принимать тип фактической записи (в данном случае — `Point`, а не `object`). Как только вы создадите верную сигнатуру, компилятор автоматически внесет исправления в ваш метод.

Как и в случае с любым другим типом, если вы берете на себя сравнение на равенство, вы также должны перекрыть `GetHashCode()`. Приятная особенность записей заключается в том, что вы не перегружаете `!=` или `==` и не реализуете `IEquatable<T>`: все это сделано вместо вас. Полностью эта тема рассматривается в соответствующем разделе главы 6 книги *C# 9.0. Справочник. Полное описание языка*.

Сопоставление с образцом

Ранее мы продемонстрировали, как применять оператор `is` для проверки, будет ли ссылочное преобразование успешным, с последующим использованием полученного преобразованного значения:

```
if (obj is string s)
    Console.WriteLine(s.Length);
```

При этом используется разновидность образца (pattern), именуемая *образцом типа*. Оператор `is` поддерживает и другие образцы, которые были введены в последние версии C#. Образцы поддерживаются в следующих контекстах:

- ✓ после оператора `is` (*переменная* `is` *образец*);
- ✓ в инструкциях `switch`;
- ✓ в выражениях `switch`.

Мы уже рассматривали образец типа в разделах “Инструкция switch для типов” и “Оператор is”. В этом разделе мы рассмотрим более сложные образцы, представленные в последних версиях C#.

Некоторые из более специализированных образцов предназначены для использования в инструкциях/выражениях `switch`. Здесь они уменьшают потребность в конструкциях `when`, и вы можете использовать `switch` там, где вы не могли делать это ранее.

Образец переменной

Образец переменной (var pattern) — это вариант *образца типа*, с помощью которого вы замените имя типа ключевым словом `var`. Преобразование всегда успешно, поэтому его цель — просто позволить вам повторно использовать переменную:

```
bool IsJanetOrJohn(string name) =>
    name.ToUpper() is var upper &&
    (upper == "JANET" || upper == "JOHN");
```

Этот код эквивалентен следующему:

```
bool IsJanetOrJohn(string name)
{
    string upper = name.ToUpper();
    return upper == "JANET" || upper == "JOHN";
}
```

Образец константы

Образец константы позволяет вам непосредственно выполнять сопоставление константе и полезен при работе с типом `object`:

```
void Foo(object obj)
{
    if (obj is 3) ...
}
```

Это выражение (выделенное полужирным шрифтом) эквивалентно следующему:

```
obj is int && (int)obj == 3
```

Как мы скоро увидим, образец константы шаблон может быть более полезным с *комбинаторами образцов*.

Образцы отношений (C# 9)

Начиная с C# 9, в образцах можно использовать операторы <, >, <= и >=:

```
if (x is > 100) Console.Write("x больше 100");
```

Эта возможность становится очень полезной в инструкции `switch`:

```
string GetWeightCategory(decimal bmi) => bmi switch
{
    < 18.5m => "слишком легкий",
    < 25m   => "нормальный",
    < 30m   => "слишком тяжелый",
    _       => "совсем плохо"
};
```

Комбинаторы образцов (C# 9)

Начиная с C# 9, для объединения образцов вы можете использовать ключевые слова `and`, `or` и `not`:

```
bool IsJanetOrJohn (string name)
    => name.ToUpper() is "JANET" or "JOHN";

bool IsVowel (char c)
    => c is 'a' or 'e' or 'i' or 'o' or 'u';

bool Between1And9 (int n) => n is >= 1 and <= 9;

bool IsLetter (char c) => c is >= 'a' and <= 'z'
                           or >= 'A' and <= 'Z';
```

Как и в случае операторов && и ||, `and` имеет более высокий приоритет, чем `or`, но вы можете использовать скобки. Хороший трюк — объединение комбинатора `not` с *образцом типа* для проверки, что объект *не* является типом:

```
if (obj is not string) ...
```

Впрочем, этот код выглядит более красиво, если переписать его как

```
if (!(obj is string)) ...
```

Кортежи и позиционные образцы

Образец кортежа (введенный в C# 8) проверяет соответствие кортежам:

```
var p = (2, 3);
Console.WriteLine(p is (2, 3)); // True
```

Образец кортежа можно рассматривать как частный случай *позиционного образца* (C# 8+), который соответствует любому типу, предоставляющему метод `Deconstruct` (см. раздел "Деконструкторы"). В следующем примере мы используем деконструктор, созданный компилятором для записи `Point`:

```
var p = new Point(2, 2);
Console.WriteLine(p is (2, 2)); // True

record Point (int X, int Y);
```

При соответствии можно выполнить деконструкцию, используя следующий синтаксис:

```
Console.WriteLine(p is (var x, var y) && x == y);
```

Вот выражение `switch`, которое объединяет образец типа с *позиционным образцом*:

```
string Print (object obj) => obj switch
{
    Point(0, 0)                       => "Пустая точка",
    Point(var x, var y) when x == y => "Диагональ"
    ...
};
```

Образцы свойств

Образец свойства (C# 8+) соответствует одному или нескольким из значений свойств объекта:

```
if (obj is string { Length:4 }) ...
```

Однако это не является большой экономией по сравнению со следующим кодом:

```
if (obj is string s && s.Length == 4) ...
```

С инструкциями и выражениями `switch` образцы свойств более полезны. Рассмотрим класс `System.Uri`, представляющий некоторый URI. Он имеет такие свойства, как `Scheme`, `Host`, `Port` и `IsLoopback`. При написании брандмауэра мы можем решать, разрешить или заблокировать URI, используя выражение `switch` , которое использует образцы свойств:

```
bool ShouldAllow(Uri uri) => uri switch
{
    { Scheme: "http", Port: 80 }   => true,
    { Scheme: "https", Port: 443 } => true,
    { Scheme: "ftp", Port: 21 }    => true,
    { IsLoopback: true }           => true,
    _ => false
};
```

Можно использовать вложенные свойства — следующая конструкция вполне корректна:

```
{ Scheme: { Length: 4 }, Port: 80 } => true,
```

Внутри образцов свойств можно использовать другие образцы, включая образцы отношений:

```
{ Host: { Length: < 1000 }, Port: > 0 } => true,
```

Вы можете ввести переменную в конце такой конструкции, а затем использовать ее в конструкции `when`:

```
{ Scheme: "http", Port: 80 } httpUri
    when httpUri.Host.Length < 1000 => true,
```

Вы также можете ввести переменные на уровне *свойств*:

```
{ Scheme: "http", Port: 80, Host: var host }
    when host.Length < 1000 => true,
```

Однако в таком случае короче и проще написать так:

```
{ Scheme: "http", Port: 80, Host: { Length: < 1000 } }
```

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 запрашивать любую коллекцию, реализующую 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.

ПРИМЕЧАНИЕ

Удобно экспериментировать с LINQ, 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 каждый член этой последовательности является элементом. В следующем примере `names` является последовательностью, а `"Tom"`, `"Dick"` и `"Harry"` — элементами:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Последовательность такого рода называется *локальной последовательностью*, потому что она представляет локальную коллекцию объектов в памяти.

Оператор запроса — это метод, который трансформирует последовательность. Типичный оператор запроса принимает *входную последовательность* и выдает трансформированную *выходную последовательность*. В классе `Enumerable` из пространства имен `System.Linq` имеется около 40 операторов запросов, которые реализованы в виде статических расширяющих методов. Их называют *стандартными операторами запросов*.

ПРИМЕЧАНИЕ

Язык LINQ поддерживает также последовательности, которые могут динамически наполняться из удаленного источника данных, наподобие SQL Server. Такие последовательности дополнительно реализуют интерфейс `IQueryable<>` и поддерживаются через соответствующий набор стандартных операторов запросов в классе `Queryable`.

Простой запрос

Запрос — это выражение, которое трансформирует последовательности с помощью одного или более операторов запросов. Простейший запрос состоит из одной входной последовательности и одного оператора. Например, мы можем применить оператор `Where` к простому массиву для извлечения элементов длиной не менее четырех символов:

```
string[] names = { "Tom", "Dick", "Harry" };

IEnumerable<string> filteredNames =
    System.Linq.Enumerable.Where(
        names, n => n.Length >= 4);

foreach (string n in filteredNames)
    Console.Write(n + "|"); // Dick|Harry|
```

Поскольку стандартные операторы запросов реализованы в виде расширяющих методов, мы можем вызывать оператор `Where` непосредственно для `names` — как если бы он был методом экземпляра:

```
IEnumerable<string> filteredNames =
        names.Where(n => n.Length >= 4);
```

(Чтобы этот код скомпилировался, потребуется импортировать пространство имен `System.Linq` с помощью директивы `using`.) Метод `Where` в `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 возвращает `true`. Внутренне он реализован посредством итератора — вот как выглядит исходный текст:

```
foreach(TSource element in source)
    if (predicate(element))
        yield return element;
```

Проецирование

Еще одним фундаментальным оператором запроса является метод `Select`, который трансформирует (*проецирует*) каждый элемент входной последовательности с помощью заданного лямбда-выражения:

```
string[] names = { "Tom", "Dick", "Harry" };

IEnumerable<string> upperNames =
    names.Select (n => n.ToUpper());
```

```
foreach (string n in upperNames)
    Console.Write (n + "|"); // TOM|DICK|HARRY|
```

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 какой вид имеет результат:

```
{ Name = Tom,   Length = 3 }
{ Name = Dick,  Length = 4 }
{ Name = Harry, Length = 5 }
```

Take и Skip

В LINQ важно первоначальное упорядочение элементов внутри входной последовательности. На это поведение полагаются некоторые операторы запросов, такие как `Take`, `Skip` и `Reverse`. Оператор `Take` выдает первые `X` элементов, отбрасывая остальные:

```
int[] numbers = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take(3);
// firstThree равно { 10, 9, 8 }
```

Оператор `Skip` пропускает первые `X` элементов и выдает остальные:

```
IEnumerable<int> lastTwo = numbers.Skip(3);
```

Операторы над элементами

Не все операторы запросов возвращают последовательности. Операторы над *элементами* извлекают из входной последовательности один элемент; примерами таких операторов 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 операторы 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 по умолчанию для соответствующих типов-значений).

Методы `Single` и `SingleOrDefault` эквивалентны методам `First` и `FirstOrDefault`, но генерируют исключение при наличии более одного соответствия. Такое поведение полезно при запросе строки по первичному ключу из таблицы базы данных.

Операторы агрегации

Операторы *агрегации* возвращают скалярное значение, обычно числового типа. Наиболее распространенными операторами агрегации 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
```

Оператор `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
```

Операторы `Min`, `Max` и `Average` принимают необязательный аргумент, который трансформирует каждый элемент до того, как он будет подвергнут агрегации:

```
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));
```

Квантификаторы

Квантификаторы возвращают значение типа `bool`. Квантификаторами являются операторы `Contains`, `Any` и `All`, а также `SequenceEquals`, который 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
```

Операторы над множествами

Операторы над множествами принимают две входные последовательности одного и того же типа. Оператор `Concat` добавляет одну последовательность в конец другой; `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 два оператора — `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 операторов запросов заключается в том, что они выполняются не тогда, когда создаются, а когда происходит *перечисление* (другими словами, когда вызывается метод `MoveNext` перечислителя). Рассмотрим следующий запрос:

```
var numbers = new List<int> { 1 };

IEnumerable<int> query = numbers.Select (n => n * 10);
numbers.Add(2);    // Вставка дополнительного элемента

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 выполняется вплоть до выполне-

ния инструкции `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 запрос базы данных без извлечения всех строк для клиента. Все стандартные операторы запросов обеспечивают отложенное выполнение со следующими исключениями:

- ✓ операторы, которые возвращают одиночный элемент или скалярное значение (*операторы над элементами*, *операторы агрегации* и *квантификаторы*);
- ✓ операторы *преобразования* `ToArray`, `ToList`, `ToDictionary`, `ToLookup` и `ToHashSet`.

Операторы преобразования отчасти удобны 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 заключается в том, что для того, чтобы запрос был выполнен повторно, вы должны позже выполнить его перечисление заново.)

В следующем примере проиллюстрирован оператор `ToList`:

```
var numbers = new List<int>() { 1, 2 };

List<int> timesTen = numbers
        .Select (n => n * 10)
        .ToList(); // Выполняется немедленно
                   // с преобразованием в List<int>
numbers.Clear();
Console.WriteLine(timesTen.Count);   // Все еще 2
```

ПРИМЕЧАНИЕ

Подзапросы обеспечивают еще один уровень косвенности. Все, что находится в подзапросе, подпадает под отложенное выполнение, включая методы агрегации и преобразования, так как сами подзапросы выполняются только отложенно, по требованию. В предположении, что

`names` — строковый массив, подзапрос выглядит примерно так:

```
names.Where(n => n.Length == names.Min(n2 =>
n2.Length))
```

Стандартные операторы запросов

Стандартные операторы запросов (реализованные в классе `System.Linq.Enumerable`) могут быть разделены на 12 категорий, которые кратко подытожены в табл. 7.

Таблица 7. Категории операторов запросов

Категория	Описание	Отложенное выполнение
Фильтрация	Возвращают подмножество элементов, удовлетворяющих заданному условию	Да
Проекция	Преобразуют каждый элемент с помощью лямбда-функции (возможно) с расширением подпоследовательностей	Да
Соединение	Объединяют элементы одной коллекции с элементами другой, используя стратегию поиска, эффективную по времени	Да
Упорядочение	Возвращают переупорядоченную последовательность	Да
Группирование	Группируют последовательность в подпоследовательности	Да
Работа с множествами	Принимают две однотипные последовательности и возвращают их объединение, сумму или разность	Да
Работа с элементами	Выбирают один элемент из последовательности	Нет
Агрегация	Выполняют вычисления над последовательностью, возвращая скалярное значение (обычно число)	Нет
Квантификация	Выполняют вычисления над последовательностью, возвращая `true` или `false`	Нет
Преобразование: импорт	Преобразуют необобщенную последовательность в (поддерживающую запросы) обобщенную последовательность	Да

Окончание табл. 7

Категория	Описание	Отложенное выполнение
Преобразование: экспорт	Преобразуют последовательность в массив, список, словарь, обеспечивая немедленное вычисление	Нет
Генерация	Производят простую последовательность	Да

В табл. 8–19 приведены описания всех операторов запросов. Операторы, выделенные полужирным, имеют специальную поддержку в языке C# (см. раздел "Выражения запросов").

Таблица 8. Операторы фильтрации

Метод	Описание
`Where`	Возвращает подмножество элементов, удовлетворяющих данному условию
`Take`	Возвращает первые `X` элементов, отбрасывая остальные
`Skip`	Игнорирует первые `X` элементов, возвращая остальные
`TakeWhile`	Выдает элементы входной последовательности, пока заданный предикат возвращает `true`
`SkipWhile`	Игнорирует элементы входной последовательности, пока заданный предикат возвращает `true`, возвращая остальные
`Distinct`	Возвращает коллекцию с удаленными дубликатами

Таблица 9. Операторы проекции

Метод	Описание
`Select`	Преобразует каждый входной элемент с помощью заданного лямбда-выражения
`SelectMany`	Преобразует каждый входной элемент, а затем выравнивает и объединяет результирующие подпоследовательности

Таблица 10. Операторы 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 один шаг, возвращая последовательность, в которой к каждой паре элементов применена функция

Таблица 11. Операторы упорядочения

Метод	Описание
`OrderBy`, **`ThenBy`**	Возвращает элементы, отсортированные в возрастающем порядке
`OrderByDescending`, **`ThenByDescending`**	Возвращает элементы, отсортированные в убывающем порядке
`Reverse`	Возвращает элементы в обратном порядке

Таблица 12. Оператор группирования

Метод	Описание
`GroupBy`	Группирует последовательность в подпоследовательности

Таблица 13. Операторы работы с множествами

Метод	Описание
`Concat`	Конкатенация двух последовательностей
`Union`	Конкатенация двух последовательностей с удалением дубликатов
`Intersect`	Возвращает элементы, присутствующие в обоих множествах
`Except`	Возвращает элементы, присутствующие в первом множестве, но не во втором

Таблица 14. Операторы работы с 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 при наличии более одного подходящего элемента
`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 15. Операторы агрегации

Метод	Описание
`Count`, `LongCount`	Возвращают общее количество элементов входной последовательности или количество элементов, удовлетворяющих заданному предикату
`Min`, `Max`	Возвращают наименьший или наибольший элемент последовательности
`Sum`, `Average`	Подсчитывают числовую сумму или среднее значение элементов последовательности
`Aggregate`	Выполняет пользовательскую агрегацию

Таблица 16. Операторы квантификации

Метод	Описание
`Contains`	Возвращает `true`, если входная последовательность содержит заданный элемент

Окончание табл. 16

Метод	Описание
`Any`	Возвращает `true`, если существуют элементы входной последовательности, удовлетворяющие заданному предикату
`All`	Возвращает `true`, если все элементы входной последовательности удовлетворяют заданному предикату
`SequenceEqual`	Возвращает `true`, если вторая последовательность содержит элементы, идентичные элементам входной последовательности

Таблица 17. Операторы преобразования (импорт)

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 18. Операторы преобразования (экспорт)

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 19. Операторы генерации

Метод	Описание
`Empty`	Создает пустую последовательность
`Repeat`	Создает последовательность повторяющихся элементов
`Range`	Создает последовательность целых чисел

Цепочки операторов запросов

Для построения более сложных запросов допускается объединять операторы запросов в цепочки. Например, следующий запрос извлекает все строки, содержащие букву *a*, сортирует их по длине, а затем преобразует результаты в символы верхнего регистра:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

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: JAY|MARY|HARRY|
```

`Where`, `OrderBy` и `Select` — это стандартные операторы запросов, которые разрешаются в вызовы расширяющих методов класса `Enumerable`. Оператор `Where` выдает отфильтрованную версию входной последовательности; оператор `OrderBy` — отсортированную версию входной последовательности; оператор `Select` — последовательность, в которой каждый входной элемент трансформирован, или *проецирован*, с помощью заданного лямбда-выражения (`n.ToUpper()` в рассмотренном случае). Данные протекают слева направо через цепочку операторов, поэтому они сначала фильтруются, затем сортируются и наконец проецируются. Конечный результат напоминает производственную линию с ленточными конвейерами, показанную на рис. 6.

Отложенное выполнение соблюдается операторами повсеместно, так что ни фильтрация, ни сортировка, ни проецирование не происходят до тех пор, пока не начнется фактическое перечисление результатов запроса.

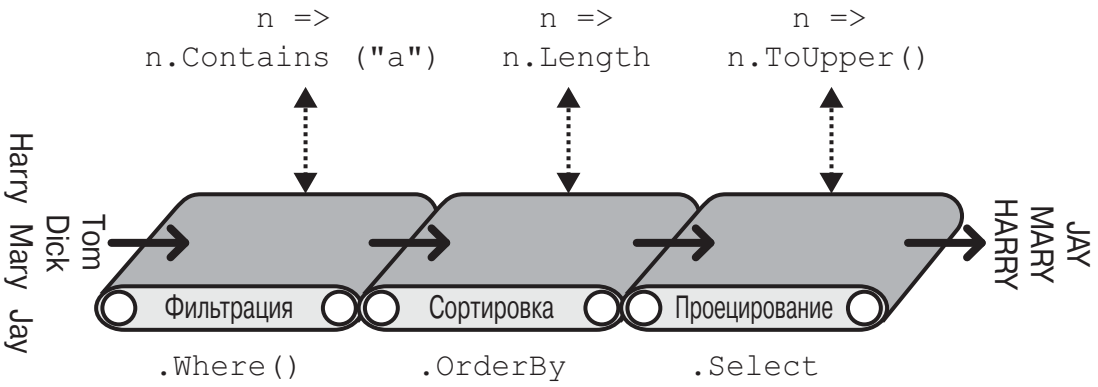


Рис. 6. *Цепочка операторов запросов*

Выражения запросов

До сих пор мы писали запросы, вызывая расширяющие методы в классе `Enumerable`. В настоящей книге мы называем это *текучим синтаксисом* (fluent syntax). 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 проиллюстрирован полный синтаксис выражения запроса.

ПРИМЕЧАНИЕ

Если вы знакомы с языком SQL, то синтаксис выражений запросов LINQ — с конструкцией `from` в начале и конструкцией `select` в конце — может показаться вам странным. На самом деле синтаксис выражений запросов более логичен, поскольку конструкции появляются *в по-*

рядке, 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.

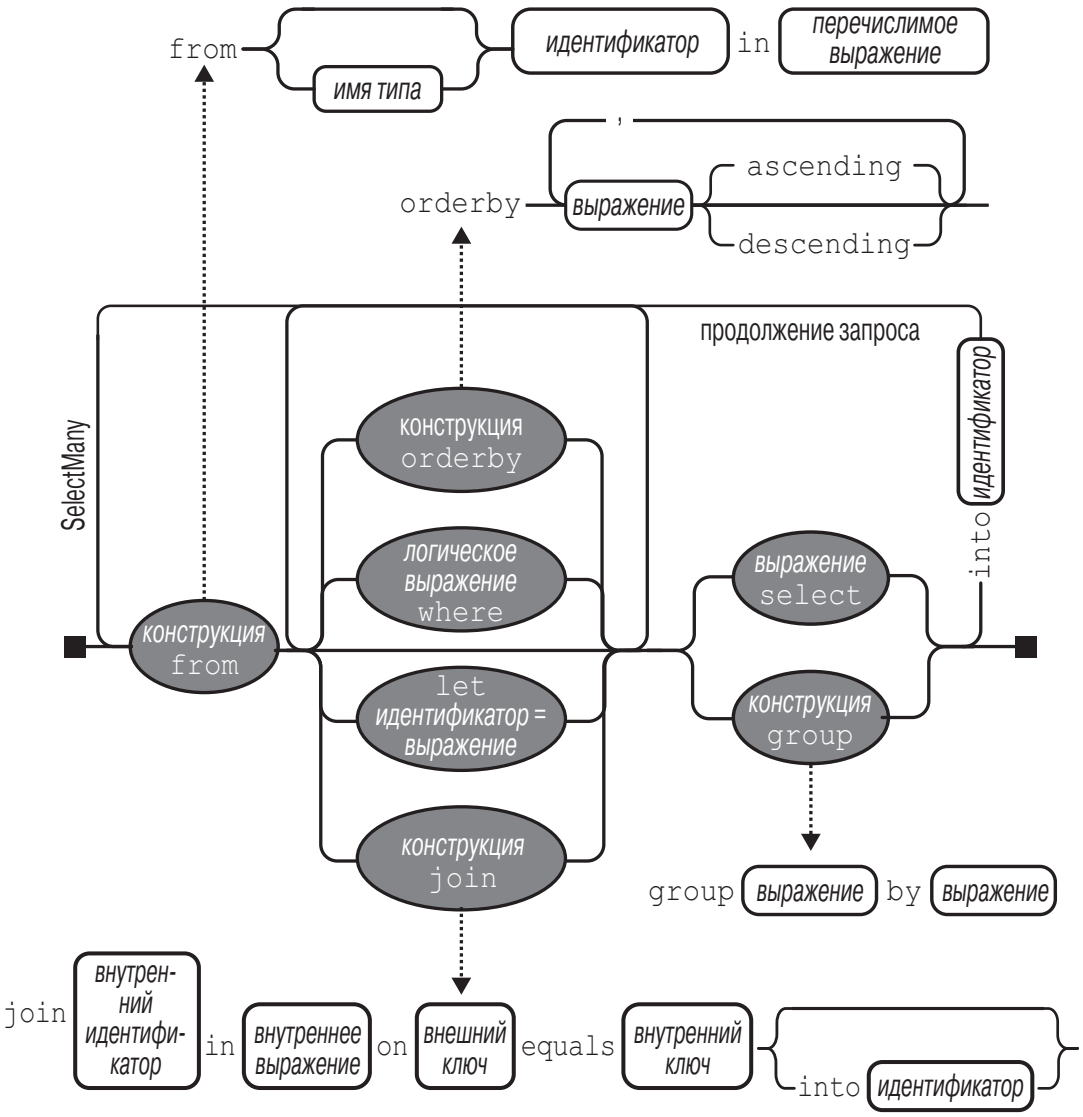


Рис. 7. *Синтаксис выражения запроса*

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 операторы `Where()`, `OrderBy()` и `Select()` разрешаются с использованием тех же правил, которые применялись бы к запросу, написанному с помощью текучего синтаксиса. В данном случае операторы привязываются к расширяющим методам в классе `Enumerable` (предполагая импорт пространства имен `System.Linq`), потому что `names` реализует интерфейс `IEnumerable<string>`. Однако при трансляции синтаксиса запросов компилятор не оказывает специальной поддержки классу `Enumerable`. Можно считать, что компилятор механически вводит слова `Where`, `OrderBy` и `Select` внутрь инструкции, после чего компилирует ее, как если бы вы набирали имена методов самостоятельно. В итоге обеспечивается гибкость в способе их разрешения — например, операторы в запросах Entity Framework привязываются к расширяющим методам в классе `Queryable`.

Выражения запросов и текучий синтаксис

И синтаксису выражений запросов, и текучему синтаксису присущи свои преимущества.

Выражения запросов поддерживают только небольшое подмножество операторов запросов, а именно:

```
Where,    Select,   SelectMany
OrderBy,  ThenBy,   OrderByDescending, ThenByDescending
GroupBy,  Join,     GroupJoin
```

Запросы, которые используют другие операторы, придется записывать либо полностью в текучем синтаксисе, либо в смешанном синтаксисе. Например:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

IEnumerable<string> query =
    from n in names
    where n.Length == names.Min(n2 => n2.Length)
    select n;
```

Приведенный запрос возвращает имена с наименьшей длиной (“Tom” и “Jay”). Подзапрос (выделенный полужирным) вычисляет минимальную длину имен и получает значение 3. Для данного подзапроса должен применяться текучий синтаксис, так как оператор `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 оператора `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 превышает два символа:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

IEnumerable<string> query =
    from n in names
    let vowelless = Regex.Replace(n, "[aeiou]", "")
    where vowelless.Length > 2
    orderby vowelless
    select n + " - " + vowelless;
```

Вот вывод, полученный при перечислении результатов этого запроса:

```
Dick - Dck
Harry - Hrry
Mary - Mry
```

Конструкция `let` выполняет вычисление для каждого элемента, не утрачивая при этом исходный элемент. В нашем запросе последующие конструкции (`where`, `orderby` и `select`) имеют доступ как к `n`, так и к `vowelless`. Запрос может включать любое количество конструкций `let`, и они могут сопровождаться дополнительными конструкциями `where` и `join`.

Компилятор транслирует ключевое слово `let` путем проекции во временный анонимный тип, который содержит исходные и трансформированные элементы:

```
IEnumerable<string> query = names
    .Select(n => new
        {
            n = n,
            vowelless = Regex.Replace (n, "[aeiou]", "")
        }
    )
    .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`, то нужно использовать ключевое слово `into`, чтобы "продолжить" запрос. Например:

```
from c in "The quick brown tiger".Split()
select c.ToUpper() into upper
where upper.StartsWith("T")
select upper

// Результат: "THE", "TIGER"
```

После конструкции `into` предыдущая переменная диапазона находится за пределами области видимости.

Компилятор просто транслирует запросы с ключевым словом `into` в более длинную цепочку операторов:

```
"The quick brown tiger".Split()
    .Select (c => c.ToUpper())
    .Where (upper => upper.StartsWith ("T"))
```

(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", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;
```

Результатом является перекрестное произведение, очень схожее с тем, которое можно было бы получить с помощью вложенных циклов `foreach`:

```
"1a", "1b", "2a", "2b", "3a", "3b"
```

При наличии в запросе более одной конструкции `from` компилятор генерирует 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 перекрестное произведение можно отфильтровать и спроецировать результат подобно *соединению*:

```
string[] players = { "Tom", "Jay", "Mary" };

IEnumerable<string> query =
    from name1 in players
    from name2 in players
    where name1.CompareTo (name2) < 0
    orderby name1, name2
    select name1 + " vs " + name2;

Результат: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

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 =
    { "Anne Williams", "John Fred Smith", "Sue Green" };

IEnumerable<string> query =
    from fullName in fullNames
    from name in fullName.Split()
    select name + " came from " + fullName;

Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
```

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 оператора *соединения*, из которых главными являются `Join` и `GroupJoin`, выполняющие соединения на основе ключей поиска. Операторы `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 основе хеш-таблиц, а не выполняют вложенные циклы. (В случае запросов Entity Framework операторы соединения не имеют никаких преимуществ перед множеством генераторов.)

Операторы `Join` и `GroupJoin` поддерживают только *эквисоединения* (т.е. условие соединения должно использовать оператор 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 следующим образом:

```
from внешняя-переменная in внешняя-последовательность
join внутренняя-переменная in внутренняя-последовательность
```

```
    on выражение-внешнего-ключа equals
       выражение-внутреннего-ключа
```

Например, рассмотрим следующие коллекции:

```
var customers = new[]
{
    new { ID = 1, Name = "Tom" },
    new { ID = 2, Name = "Dick" },
    new { ID = 3, Name = "Harry" }
};
var purchases = new[]
{
    new { CustomerID = 1, Product = "House" },
    new { CustomerID = 2, Product = "Boat" },
    new { CustomerID = 2, Product = "Car" },
    new { CustomerID = 3, Product = "Holiday" }
};
```

Мы можем выполнить их соединение следующим образом:

```
IEnumerable<string> query =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    select c.Name + " bought a " + 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 результата
        c.Name + " bought a " + p.Product
);
```

А вот как выглядит результат:

```
Tom bought a House
Dick bought a Boat
Dick bought a Car
Harry bought a Holiday
```

В случае локальных последовательностей при обработке крупных коллекций операторы `Join` и `GroupJoin` более эффективны, чем `SelectMany`, поскольку они сначала загружают внутреннюю последовательность в хеш-таблицу поиска по ключу. Однако в случае запроса к базе данных тот же результат с такой же эффективностью можно получить следующим образом:

```
from c in customers
from p in purchases
where c.ID == p.CustomerID
select c.Name + " bought a " + p.Product;
```

GroupJoin

Оператор 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 ним следует ключевое слово into. Ниже приведен простейший пример, в котором задействованы коллекции customers и purchases, подготовленные в предыдущем разделе:

```
var query =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select custPurchases; // custPurchases представляет
                          // собой последовательность
```

ПРИМЕЧАНИЕ

Конструкция into транслируется в GroupJoin, только когда она появляется непосредственно после конструкции join. При размещении после конструкции select или group она означает *продолжение запроса*. Такие два применения ключевого слова into существенно различаются, хотя и обладают одной общей характеристикой: в обоих случаях вводится новая переменная диапазона.

Результатом будет последовательность последовательностей IEnumerable<IEnumerable<T>>, обход которой можно было бы организовать следующим образом:

```
foreach (var purchaseSequence in query)
    foreach (var purchase in purchaseSequence)
        Console.WriteLine (purchase.Product);
```

Однако это не слишком полезно, так как purchaseSequence не имеет ссылок на внешнего заказчика из customers. Чаще всего ссылка на внешнюю переменную диапазона производится в проекции:

```
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 проецированием в анонимный тип, который включает подзапрос:

```
from c in customers
select new
{
    CustName = c.Name,
    custPurchases =
        purchases.Where (p => c.ID == p.CustomerID)
}
```

Zip

Оператор `Zip` является простейшим оператором соединения. Он обходит две последовательности одновременно (подобно застежке-молнии (zipper)) и возвращает последовательность, полученную в результате применения функции к каждой паре элементов. Так, код

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip =
    numbers.Zip(words, (n, w) => n + "=" + w);
```

приводит к последовательности с элементами

```
3=three
5=five
7=seven
```

Излишние элементы в любой из входных последовательностей игнорируются. Оператор `Zip` не поддерживается в запросах к базам данных.

Упорядочение

Последовательность сортируется с помощью ключевого слова `orderby`. Разрешено указывать любое количество выражений, по которым выполняется сортировка:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

IEnumerable<string> query = from n in names
                            orderby n.Length, n
                            select n;
```

Сначала выполняется сортировка по длине, а затем — по имени, что приводит к следующему результату:

```
Jay, Tom, Dick, Mary, Harry
```

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)
```

Оператор `ThenBy` уточняет результаты предшествующей сортировки, а не заменяет их.

После любого выражения `orderby` можно размещать ключевое слово `descending`:

```
orderby n.Length descending, n
```

Запрос при этом транслируется в

```
.OrderByDescending(n => n.Length).ThenBy(n => n)
```

ПРИМЕЧАНИЕ

Операторы упорядочения возвращают расширенный тип `IEnumerable<T>` с именем `IOrderedEnumerable<T>`. В данном интерфейсе определена дополнительная функциональность, требуемая оператором `ThenBy`.

Группирование

Оператор `GroupBy` превращает плоскую входную последовательность в последовательность *групп*. Например, приведенный ниже код группирует последовательность имен по их длине:

```
string[] names = { "Tom","Dick","Harry","Mary","Jay" };

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);
}

Length=3: Tom Jay
Length=4: Dick Mary
Length=5: Harry
```

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 применим к подпоследовательности в целом
    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
```

что транслируется в

```
names.GroupBy(
    name => name.Length,
    name => name.ToUpper())
```

Подколлекции не выдаются в порядке следования ключей. Оператор `GroupBy` не выполняет сортировку (фактически он сохраняет исходное упорядочение). Для сортировки следует до-

бавить оператор OrderBy (что в первую очередь означает добавление конструкции into, так как 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 ровно по два совпадения:

```
from name in names
group name.ToUpper() by name.Length into grouping
where grouping.Count() == 2
select grouping
```

ПРИМЕЧАНИЕ

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 отдельным элементам.

OfType и Cast

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>();
```

Это полезно тем, что появляется возможность запрашивать коллекции, разработанные до выхода версии C# 2.0 (в которой был введен интерфейс IEnumerable<T>), такие как ControlCollection из System.Windows.Forms.

Cast и OfType различаются своим поведением, когда встречается входной элемент несовместимого типа: Cast генерирует исключение, а OfType такой элемент игнорирует.

Правила совместимости элементов соответствуют аналогичным правилам для оператора `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 оператор `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 разрешения типов, членов и операторов — со времени компиляции до времени выполнения. Динамическое связывание удобно, когда во время компиляции *вы* знаете, что определенная функция, член или оператор 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 в противном случае 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()`, но просто не можем проверить это статически. Поскольку `d` относится к динамическому типу, компилятор откладывает связывание `Quack()` с `d` до времени выполнения. Понима-

ние смысла такого действия требует понимания различий между статическим и динамическим связываниями.

Статическое и динамическое связывания

Канонический пример связывания предусматривает при компиляции выражения отображение имени в конкретную функцию. Для компиляции следующего выражения компилятор должен найти реализацию метода с именем `Quack()`:

```
d.Quack();
```

Давайте предположим, что статическим типом `d` является `Duck`:

```
Duck d = ...
d.Quack();
```

В таком простом случае компилятор осуществляет связывание за счет поиска в типе `Duck` метода без параметров по имени `Quack()`. Если найти такой метод не удается, 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, генерируется ошибка компиляции. Независимо от того, с каким методом выполнено связывание, суть заключается в том, что связывание выполняется компилятором, и оно полностью зависит от статических сведений о типах операндов (в данном случае — `d`). Именно поэтому такой процесс называется *статическим связыванием*.

Теперь изменим статический тип `d` на `object`:

```
object d = ...
d.Quack();
```

Вызов `Quack()` приводит к ошибке времени компиляции, так как несмотря на то, что хранящееся в `d` значение может содержать метод с именем `Quack()`, компилятор не может об этом знать, поскольку единственная информация, которой он располагает, — тип переменной, которым в рассматриваемом случае является `object`. Давайте теперь изменим статический тип `d` на `dynamic`:

```
dynamic d = ...
d.Quack();
```

Тип dynamic схож с object — он точно так же не описывает конкретный тип. Отличие заключается в том, что тип dynamic допускает применение способами, которые во время компиляции не известны. Динамический объект связывается во время выполнения на основе своего типа времени выполнения, а не типа времени 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 этот интерфейс используется для связывания во время выполнения. В противном случае связывание осуществляется почти так же, как в ситуации, когда компилятору известен тип динамического объекта времени выполнения. Эти две альтернативы называются *пользовательским связыванием* и *языковым связыванием*.

Пользовательское связывание

Пользовательское связывание (custom binding) осуществляется, когда динамический объект реализует интерфейс IDynamicMetaObjectProvider. Хотя интерфейс IDynamicMetaObjectProvider можно реализовать в типах, которые вы пишете на языке C#, и поступать так полезно, более распространенный случай предусматривает запрос объекта, реализующего IDynamicMetaObjectProvider, из динамического языка, который реализован в .NET с помощью исполняющей среды динамического языка (Dynamic Language Runtime — DLR), скажем, IronPython или IronRuby. Объекты из таких языков неявно реализуют интерфейс IDynamicMetaObjectProvider в качестве средства непосредственного управления смыслом выполняемых над ними операций. Ниже приведен простой пример:

```
dynamic d = new Duck();
d.Quack();    // Вызывается Quack
d.Waddle();   // Вызывается Waddle
public class Duck : DynamicObject  // В System.Dynamic
{
    public override bool TryInvokeMember(
            InvokeMemberBinder binder,
```

```
            object[] args, out object result)
    {
    Console.WriteLine("Вызывается " + binder.Name);
        result = null;
        return true;
    }
}
```

Класс `Duck` в действительности не имеет метода `Quack()`. Вместо этого он применяет пользовательское связывание для перехвата и интерпретации всех вызовов методов. Пользовательское связывание более подробно обсуждается в книге *C# 9.0. Справочник. Полное описание языка*.

Языковое связывание

Языковое связывание осуществляется, 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 тем, что не имеют общего интерфейса. Ранее было показано, что методы могут быть привязаны динамически; то же самое справедливо и для операторов:

```
int x = 3, y = 4;
Console.WriteLine(Mean(x, y));

dynamic Mean(dynamic x, dynamic y) => (x+y) / 2;
```

Преимущество этого подхода 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 времени компиляции.

ПРИМЕЧАНИЕ

Динамическое связывание обходит систему статической безопасности типов, но не динамической безопасности. В отличие от рефлексии с помощью динамического связывания обойти правила доступности членов невозможно.

Языковое связывание времени выполнения преднамеренно ведет себя максимально схоже со статическим связыванием, как будто типы времени выполнения динамических объектов были известны еще во время компиляции. В предыдущем примере поведение программы окажется идентичным тому, как если бы метод `Mean()` был жестко закодирован для работы с типом `int`. Наиболее заметным исключением при проведении аналогии между статическим и динамическим связываниями являются расширяющие методы, которые рассматриваются в разделе "Невызываемые функции".

ПРИМЕЧАНИЕ

Динамическое связывание 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 временны*е* издержки при выполнении простого динамического выражения на современном оборудовании до менее чем 100 нс.

Исключение `RuntimeBinderException`

Если привязка к члену не удается, генерируется исключение `RuntimeBinderException`. Его можно считать ошибкой времени компиляции, перенесенной на время выполнения:

```
dynamic d = 5;
d.Hello(); // Генерация RuntimeBinderException
```

Исключение генерируется из-за того, что тип `int` не имеет метода `Hello()`.

Представление `dynamic` времени выполнения

Между типами `dynamic` и `object` имеется глубокая эквивалентность. Исполняющая среда трактует следующее выражение как `true`:

```
typeof (dynamic) == typeof (object)
```

Данный принцип распространяется также 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 = "hello";
Console.WriteLine(x.GetType().Name);            // String
x = 123;      // Ошибки нет (несмотря на ту же переменную)
Console.WriteLine(x.GetType().Name);            // Int32
```

Структурно какие-либо различия 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("hello");
Console.WriteLine(o); // hello
```

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`, так как тип `int` не может быть неявно преобразован в `short`:

```
int i = 7;
dynamic d = i;
long l = d;  // OK - работает неявное преобразование
short j = d; // Генерация RuntimeBinderException
```

Сравнение `var` и `dynamic`

Несмотря на внешнее сходство типов `var` и `dynamic`, разница между ними существенна:

- ✓ `var` говорит: “пусть компилятор выведет тип”;
- ✓ `dynamic` говорит: “пусть исполняющая среда выведет тип”.

Вот иллюстрация сказанного:

```
dynamic x = "hello"; // Статический тип - dynamic
var y = "hello";     // Статический тип - string
int i = x;           // Ошибка времени выполнения
int j = y;           // Ошибка времени компиляции
```

Динамические выражения

Поля, свойства, методы, события, конструкторы, индексаторы, операторы и преобразования могут вызываться динамически.

Попытка использования результата динамического выражения с возвращаемым типом `void` запрещена — точно так же, как и в случае статически типизированного выражения. Отличие заключается в том, что возникает ошибка времени выполнения.

Выражения, содержащие динамические операнды, обычно сами являются динамическими, так как эффект отсутствия информации о типе имеет каскадный характер:

```
dynamic x = 2;
var y = x * 3; // Статический тип y - 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 вызываются с динамическими аргументами.

Кроме того, существует несколько особых случаев, когда выражение, содержащее динамический аргумент, является статическим, включая передачу индекса массиву и выражения для создания делегатов.

Разрешение перегруженных динамических членов

В каноническом сценарии использования `dynamic` участвует динамический *получатель* (receiver). Это означает, что получателем динамического вызова функции является динамический объект:

```
dynamic x = ...;
x.Foo (123); // x - получатель
```

Однако динамическое связывание не ограничивается получателями: аргументы методов также пригодны для динамического связывания. Следствием вызова функции с динамическими аргументами будет откладывание распознавания перегруженных версий со времени 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 = "Hello";
    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, возникает ошибка времени компиляции.

Если функция вызывается с комбинацией динамических и статических аргументов, то окончательный выбор метода будет отражать комбинацию 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 = "hello";
    dynamic d = "goodbye";
    X(o,d);    // os
}
```

Вызов `X(o,d)` связывается динамически, потому что один из его аргументов, `d`, определен как `dynamic`. Но поскольку переменная `o` известна статически, связывание — хотя оно происходит

динамически — будет использовать эту статическую информацию. 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".

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 две части информации: имя вызываемой функции и объекта, для которого должна вызываться функция. Однако в каждом из трех невызываемых сценариев участвует *дополнительный тип*, который известен только во время 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.) Подобная ситуация возникает при вызове скрытого члена базового класса: дополнительный тип должен быть указан либо через приведение, либо через ключевое слово `base` — а во время выполнения этот дополнительный тип утрачивается.

Перегрузка операторов

Операторы могут быть перегружены для обеспечения пользовательских типов более естественным синтаксисом. Перегрузку

операторов наиболее целесообразно использовать при реализации пользовательских структур, которые представляют относительно примитивные типы данных. Например, хорошим кандидатом на перегрузку операторов может служить пользовательский числовой тип.

Разрешено перегружать следующие символьные операторы:

```
+     -     *     /     ++    --    !     ~     %
&     |     ^     ==    !=    <     <<    >>    >
```

Можно также перекрывать явные и неявные преобразования (с применением ключевых слов `explicit` и `implicit`), а также операторы `true` и `false`.

Составные операторы присваивания (например, `+=` и `/=`) автоматически перекрываются при перекрытии обычных операторов (например, `+` и `/`).

Функции операторов

Чтобы перегрузить оператор, следует объявить *функцию оператора*. Функция оператора должна быть статической, и по крайней мере один из операндов обязан иметь тип, в котором объявлена функция оператора. В следующем примере мы определяем структуру с именем `Note`, представляющую музыкальную ноту, а затем перегружаем оператор `+`:

```
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 оператор +, можно использовать и оператор +=:

```
CSharp += 2;
```

Подобно методам и свойствам функции операторов, состоящие из одиночного выражения, разрешено записывать более кратко с помощью синтаксиса функций, сжатых до выражений:

```
public static Note operator + (Note x, int semitones)
            => new Note (x.value + semitones);
```

Перегрузка операторов эквивалентности и сравнения

Операторы эквивалентности и сравнения часто перегружаются при написании структур, и в редких случаях — при написании классов. При перегрузке операторов эквивалентности и сравнения должны соблюдаться специальные правила и обязательства.

Парность

Компилятор C# требует, чтобы были определены оба оператора в логической паре — такими парами являются == и !=, < и >, а также <= и >=.

`Equals()` и `GetHashCode()`

При перегрузке операторов == и != для типа обычно необходимо перекрывать методы `Equals()` и `GetHashCode()` класса `object`, чтобы коллекции и хеш-таблицы могли надежно работать с данным типом.

`IComparable` и `IComparable<T>`

Если перегружаются операторы < и >, то обычно должны быть реализованы интерфейсы `IComparable` и `IComparable<T>`.

Расширим предыдущий пример, чтобы показать, каким образом можно было бы перегрузить операторы 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 хеш-кода используется хеш-код value:
public override int GetHashCode() => value.GetHashCode();
```

Пользовательские явные и неявные преобразования

Неявные и явные преобразования являются перегружаемыми операторами. Как правило, эти операторы 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 герцев
// (к ближайшему полутону)
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
```

ПРИМЕЧАНИЕ

Данный пример несколько надуманный — в действительности такие преобразования можно реализовать эффективнее с помощью метода `ToFrequency()` и (статического) метода `FromFrequency()`.

Операторы `as` и `is` игнорируют пользовательские преобразования.

Атрибуты

Вам уже знакомо понятие атрибуции элементов кода в форме модификаторов, таких как `virtual` или `ref`. Эти конструкции встроены в язык. *Атрибуты* представляют собой расширяемый механизм для добавления пользовательской информации к элементам кода (сборкам, типам, членам, возвращаемым значениям и параметрам). Такая расширяемость полезна для служб, глубоко интегрированных в систему типов, и не требует специальных ключевых слов или конструкций языка C#.

Хорошим сценарием применения атрибутов является *сериализация* — процесс преобразования произвольных объектов в определенный формат и обратно с целью хранения или передачи. В таком сценарии атрибут поля может определять трансляцию между представлением поля в C# и его представлением в применяемом формате.

Классы атрибутов

Атрибут определяется классом, который наследован (непосредственно или опосредованно) от абстрактного класса `System.Attribute`. Чтобы присоединить атрибут к элементу кода, перед этим элементом требуется указать имя типа атрибута в квадратных скобках. Например, в приведенном ниже коде к классу `Foo` присоединен атрибут `ObsoleteAttribute`:

```
[ObsoleteAttribute]
public class Foo {...}
```

Этот конкретный атрибут распознается компилятором и приводит к выдаче компилятором предупреждения, если в коде встретится ссылка на тип или член, помеченный таким атрибутом. По соглашению имена всех типов атрибутов оканчиваются словом `Attribute`. Данное соглашение поддерживается компилятором C# и позволяет опускать суффикс `Attribute` при присоединении атрибута:

```
[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 `System.Xml.Serialization`) о том, что объект представлен в виде XML, и принимает несколько *параметров атрибута*. В итоге атрибут отображает класс `CustomerEntity` на XML-элемент с именем `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. Однако атрибуты можно присоединять и 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 этих двух способов). Следующие два примера семантически идентичны:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}
```

Написание пользовательских атрибутов

Путем создания подклассов класса `System.Attribute` можно определять собственные атрибуты. Например, можно использовать следующий пользовательский 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;
}
```

Вот как можно применить данный атрибут:

```
class Foo
{
    [Test]
    public void Method1() { ... }

    [Test(20)]
    public void Method2() { ... }

    [Test(20, FailureMessage="Debugging Time!")]
    public void Method3() { ... }
}
```

Атрибут `AttributeUsage` определяет конструкцию (или комбинацию конструкций), к которой может быть применен пользовательский атрибут. Перечисление `AttributeTargets` включает такие члены, как `Class`, `Method`, `Parameter` и `Constructor` (а также `All` для объединения всех целей).

Получение атрибутов во время выполнения

Существуют два стандартных способа получения атрибутов во время выполнения:

- ✓ вызов метода `GetCustomAttributes()` любого объекта `Type` или `MemberInfo`;
- ✓ вызов метода `Attribute.GetCustomAttribute()` или `Attribute.GetCustomAttributes()`.

Последние два метода перегружены для приема любого объекта рефлексии, который соответствует корректной цели атрибута (`Type`, `Assembly`, `Module`, `MemberInfo` или `ParameterInfo`).

Вот как можно выполнить перечисление всех методов рассмотренного выше класса `Foo`, которые имеют атрибут `Test Attribute`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
    TestAttribute att = (TestAttribute)
        Attribute.GetCustomAttribute
            (mi, typeof (TestAttribute));

    if (att != null)
        Console.WriteLine (
```

```
            "{0} will be tested; reps={1}; msg={2}",
            mi.Name, att.Repetitions, att.FailureMessage);
}
```

Вывод имеет следующий вид:

```
Method1 will be tested; reps=1; msg=
Method2 will be tested; reps=20; msg=
Method3 will be tested; reps=20; msg=Debugging Time!
```

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 значение параметра по умолчанию:

- ✓ `[CallerMemberName]` дает имя члена вызывающего компонента;
- ✓ `[CallerFilePath]` дает путь к файлу исходного кода вызывающего компонента;
- ✓ `[CallerLineNumber]` дает 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);
    }
}
```

В предположении, что код находится в файле `c:\source\test\Program.cs`, вывод будет таким:

```
Main
c:\source\test\Program.cs
6
```

Как и в случае стандартных необязательных параметров, 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 журналирования, а также при реализации шаблонов уведомления об изменениях. Например, мы можем вызвать метод, подобный приведенному ниже, из средства доступа `set` определенного свойства без необходимости указывать имя этого свойства:

```
void RaisePropertyChanged (
    [CallerMemberName] string propertyName = null)
    {
        ...
    }
```

Асинхронные функции

Ключевые слова `await` и `async` поддерживают *асинхронное программирование* — стиль программирования, при котором длительно выполняющиеся функции делают бо́льшую часть своей работы (или даже всю) *после* того, как управление возвращено вызывающему компоненту. Такое программирование кардинально отличается от нормального синхронного программирования, когда длительно выполняющиеся функции *блокируют* вызывающий код до тех пор, пока операция не будет завершена. Асинхронное программирование подразумевает *параллелизм*, потому что дли-

тельно выполняющаяся операция продолжает выполняться *параллельно* с работой вызвавшего ее кода. Разработчик асинхронной функции инициирует такой параллелизм либо с помощью многопоточности (для операций с интенсивными вычислениями), либо посредством механизма обратных вызовов (для операций с интенсивным вводом-выводом).

ПРИМЕЧАНИЕ

Многопоточность, параллелизм и асинхронное программирование — обширные темы. Им посвящены две главы в книге *C# 9.0. Справочник. Полное описание языка*; кроме того, они рассматриваются по адресу `http://albahari.com/threading`.

Например, рассмотрим следующий *синхронный* метод, интенсивный в плане вычислений и требующий длительного времени работы:

```
int ComplexCalculation()
{
    double x = 2;
    for (int i = 1; i < 100000000; i++)
        x += Math.Sqrt (x) / i;
    return (int)x;
}
```

Метод `ComplexCalculation()` блокирует вызывающий код на несколько секунд, пока не выполнит вычисления, и только затем возвращает полученный результат вызывающему коду:

```
int result = ComplexCalculation();
// Через несколько секунд:
Console.WriteLine (result); // 116
```

В среде CLR определен класс с именем `Task<TResult>` (из пространства имен `System.Threading.Tasks`), предназначенный для инкапсуляции концепции операции, которая завершается в будущем. Можно сгенерировать объект `Task<TResult>` для операции с интенсивными вычислениями с помощью вызова метода `Task.Run()`, который сообщает среде CLR о необходимости выполнения указанного делегата в отдельном потоке, выполняющемся параллельно вызывающему компоненту:

```
Task<int> ComplexCalculationAsync()
    => Task.Run( () => ComplexCalculation() );
```

Этот метод является *асинхронным*, потому что он немедленно возвращает управление вызывающему коду и продолжает выполняться параллельно. Однако нам нужен некий механизм, который давал бы возможность вызывающему коду указывать, что должно произойти, когда вычисления завершатся и результат станет доступным. Класс `Task<TResult>` решает эту задачу, открывая доступ к методу `GetAwaiter()`, который позволяет вызывающему коду 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. Наше продолжение сначала вызывает метод `GetResult()`, который возвращает результат вычисления. (Или, если задание завершилось *неудачно*, сгенерировав исключение, вызов `GetResult()` сгенерирует это исключение повторно.) Затем продолжение выводит на консоль результат с помощью `Console.WriteLine()`.

Ключевые слова `await` и `async`

Ключевое слово `await` упрощает присоединение продолжений. Начнем с базового сценария и рассмотрим следующий код:

```
var результат = await выражение;
инструкция (инструкции);
```

Компилятор преобразует его в код, функционально подобный показанному ниже:

```
var awaiter = выражение.GetAwaiter();
awaiter.OnCompleted (() =>
{
    var результат = awaiter.GetResult();
    инструкция (инструкции)
);
```

ПРИМЕЧАНИЕ

Компилятор также создает код для оптимизации сценария синхронного (немедленного) завершения операции. Распространенная причина немедленного завершения асинхронной операции — операция реализует внутренний механизм кеширования и результат уже находится в кеше.

Следовательно, мы можем вызвать определенный ранее метод `ComplexCalculationAsync()` следующим образом:

```
int result = await ComplexCalculationAsync();
Console.WriteLine(result);
```

Чтобы код скомпилировался, к содержащему его методу необходимо добавить модификатор `async`:

```
async void Test()
{
    int result = await ComplexCalculationAsync();
    Console.WriteLine(result);
}
```

Модификатор `async` сообщает компилятору о том, что `await` необходимо трактовать как ключевое слово, а не как идентификатор (в итоге код, написанный до выхода версии C# 5.0, где слово `await` могло быть идентификатором, по-прежнему будет успешно компилироваться). Модификатор `async` может применяться только к методам (и лямбда-выражениям), которые возвращают `void` либо (как вы увидите позже) объект `Task` или `Task<TResult>`.

ПРИМЕЧАНИЕ

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 по себе обычно являются асин-

хронными. Чтобы увидеть, почему, давайте посмотрим, каким образом процесс выполнения проходит через асинхронную функцию.

Встретив выражение `await`, управление (обычно) возвращается вызывающему коду, что очень похоже на поведение `yield return` в итераторе. Но перед возвратом исполняющая среда присоединяет к ожидающему заданию продолжение, гарантирующее, что когда задание завершится, поток управления вернется обратно в метод и продолжит работу с того места, где он его оставил. Если в задании возникает ошибка, то ее исключение генерируется повторно (благодаря вызову `GetResult()`); в противном случае выражению `await` присваивается возвращаемое значение задания.

ПРИМЕЧАНИЕ

Реализация средой CLR метода `OnCompleted()` объекта ожидания задания гарантирует, что по умолчанию продолжения отправляются, при его наличии, через текущий *контекст синхронизации*. На деле это означает, что если в сценариях с богатыми пользовательскими интерфейсами (WPF, UWP (универсальная платформа Windows) и Windows Forms) внутри потока пользовательского интерфейса используется `await`, то выполнение кода будет продолжено в том же самом потоке. В итоге упрощается обеспечение безопасности относительно потоков.

Выражение, к которому применяется `await`, обычно является заданием. Однако 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 логическое свойство `IsCompleted`, которое выполняет проверку синхронного завершения.

Обратите внимание, что выражение `await` вычисляется как имеющее тип `int`; именно поэтому ожидаемым выражением было `Task<int>` (метод `GetAwaiter().GetResult()` которого возвращает значение типа `int`).

Ожидание необобщенного задания вполне корректно и генерирует выражение `void`:

```
await Task.Delay(5000);
Console.WriteLine("Прошло 5 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` или 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 коду 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, подобно тому, 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 требуется:

```
async Task PrintAnswerToLife()
{
    await Task.Delay(5000);
    int answer = 21 * 2;
    Console.WriteLine (answer);
}
```

Обратите внимание, что в теле метода мы не возвращаем задание явно. Компилятор самостоятельно создает задание, которое сигнализирует о завершении данного метода (или о возникновении необработанного исключения). В результате облегчается создание цепочек асинхронных вызовов:

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine("Выполнено");
}
```

(А поскольку `Go()` возвращает `Task`, сам метод `Go()` поддерживает ожидание посредством `await`.) Компилятор разворачивает асинхронные функции, возвращающие задания, в код, косвенно использующий класс `TaskCompletionSource` для создания задания, которое затем отправляет сигнал о завершении или сбое.

ПРИМЕЧАНИЕ

`TaskCompletionSource` — это тип CLR, позволяющий создавать задания, которыми вы управляете вручную, сигнализируя об их завершении с помощью результата (или о сбое выполнения с помощью генерации исключения). В отличие от `Task.Run()`, тип `TaskCompletionSource` не связывает поток на протяжении выполнения операции. Он также применяется при написании методов с интенсивным вводом-выводом, возвращающих объекты заданий (наподобие `Task.Delay()`).

Цель заключается в том, чтобы при завершении асинхронного метода, возвращающего объект задания, обеспечить возможность передачи управления в место кода, где происходит ожидание, с помощью продолжения.

Возврат 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, так что
    // метод возвращает Task<int>
    return answer;
}
```

Продемонстрировать работу метода GetAnswerToLife() можно путем вызова его из метода PrintAnswerToLife() (который, в свою очередь, вызывается из Go()):

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine ("Done");
}
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 схожим с синхронным. 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 ("Done");
}
void PrintAnswerToLife()
{
    int answer = GetAnswerToLife();
    Console.WriteLine (answer);
}
int GetAnswerToLife()
{
    Thread.Sleep (5000);
    int answer = 21 * 2;
    return answer;
}
```

Здесь также проиллюстрирован базовый принцип проектирования с использованием асинхронных функций в C#, который предусматривает написание ваших методов синхронно, с последующей заменой вызовов *синхронных* методов вызовами *асинхронных* методов, и применение к ним `await`.

Параллелизм

Мы только что продемонстрировали наиболее распространенный подход, при котором ожидание функций, возвращающих объекты заданий, выполняется 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 выполняется параллельно. Например, показанный ниже код дважды параллельно выполняет метод `PrintAnswerToLife()`:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

Применение `await` к обеим операциям "завершает" параллелизм в данной точке (и повторно генерирует любые исключения, которые могли поступить из этих заданий). Класс `Task` предоставляет статический метод с именем `WhenAll()`, позволяющий достичь того же результата чуть более эффективно. Метод

`WhenAll()` возвращает задание, которое завершается, когда завершаются все переданные ему задания:

```
await Task.WhenAll(PrintAnswerToLife(),
                   PrintAnswerToLife());
```

Метод `WhenAll()` называется *комбинатором заданий*. (Класс `Task` предлагает также комбинатор заданий с именем `WhenAny()`, возвращающий задание, которое завершается, когда завершается любое из заданий, переданных `WhenAny()`.) Комбинаторы заданий подробно рассматриваются в книге *C# 9.0. Справочник. Полное описание языка*.

Асинхронные лямбда-выражения

Подобно тому, как могут быть асинхронными обычные *именованные* методы:

```
async Task NamedMethod()
{
    await Task.Delay(1000);
    Console.WriteLine("Foo");
}
```

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("Foo");
};
```

Вызывать их и ожидать их завершения можно точно так же:

```
await NamedMethod();
await unnamed();
```

Асинхронные лямбда-выражения могут использоваться при присоединении обработчиков событий:

```
myButton.Click += async (sender, args) =>
{
    await Task.Delay (1000);
    myButton.Content = "Done";
};
```

Это более лаконично, чем следующий код, обеспечивающий тот же результат:

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
{
    await Task.Delay (1000);
    myButton.Content = "Done";
};
```

Асинхронные лямбда-выражения также могут возвращать `Task<TResult>`:

```
Func<Task<int>> unnamed = async () =>
{
    await Task.Delay (1000);
    return 123;
};
int answer = await unnamed();
```

Асинхронные потоки

Наличие конструкции `yield return` позволяет писать итераторы; наличие `await` позволяет писать асинхронные функции. *Асинхронные потоки* (C# 8) объединяют эти концепции и позволяют писать итераторы с ожиданием, производящие элементы асинхронно. Эта поддержка основана на следующей паре интерфейсов, которые являются асинхронными аналогами интерфейсов, описанных в разделе “Перечисление и итераторы”:

```
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>` с тем отличием, что она делает возможным более эффективное выполнение, когда задание 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();
}
```

ПРИМЕЧАНИЕ

Действие по извлечению каждого элемента из последовательности (`MoveNextAsync()`) представляет собой асинхронную операцию, поэтому асинхронные потоки наиболее подходят, когда элементы поступают постепенно (как при обработке данных из видеопотока). И наоборот, тип `Task<IEnumerable<T>>` больше подходит, когда последовательность *как единое целое* задерживается, а все элементы поступают вместе.

Для генерации асинхронного потока требуется написать метод, который сочетает в себе принципы итераторов и асинхронных методов. Другими словами, метод должен включать `yield return` и `await` и должен 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 инструкцию `await foreach`:

```
await foreach(var number in RangeAsync (0, 10, 100))
    Console.WriteLine(number);
```

Небезопасный код и указатели

Язык C# поддерживает прямую работу с памятью через указатели внутри блоков кода, которые помечены как небезопасные и скомпилированы с ключом компилятора `/unsafe`. Типы указателей полезны главным образом при взаимодействии с API-интерфейсами C, но могут применяться и для доступа к памяти за пределами управляемой кучи или для узких мест, критичных для производительности.

Основы работы с указателями

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. В табл. 20 показаны основные операторы над указателями.

Таблица 20. Операторы для работы с указателями

Оператор	Описание
`&`	Оператор *взятия адреса* возвращает указатель на переменную
`*`	Оператор разыменования возвращает переменную по адресу в указателе
`->`	Оператор *указателя на член* представляет собой синтаксическое сокращение: `x->y` эквивалентно `(*x).y`

Небезопасный код

Помечая тип, член типа или блок инструкций ключевым словом `unsafe`, вы разрешаете в этой области видимости использовать типы указателей и выполнять операции над указателями в стиле C++. Ниже показан пример применения указателей для быстрой обработки изображения:

```
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 может выполняться быстрее, чем соответствующая ему безопасная реализация. В показанном примере код требовал бы вложенного цикла с индексацией в массиве и проверкой границ. Небезопасный метод C# может даже оказаться быстрее, чем вызов внешней функции C, поскольку при этом не будет никаких накладных расходов, связанных с покиданием управляемой среды выполнения.

Инструкция `fixed`

Инструкция `fixed` необходима для закрепления управляемого объекта, такого как изображение 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 работы с ним, а потому инструкция `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 инструкции `fixed` можно получать указатель на тип-значение, массив типов-значений или строку. В случае массивов и строк указатель фактически указывает на первый элемент, который имеет тип-значение.

Типы-значения, 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) // Фиксация test
        {
            *p = 9;
        }
        System.Console.WriteLine (test.x);
    }
}
```

Оператор указателя на член

В дополнение к операторам & и * язык C# предлагает оператор -> в стиле C++, который 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 в стеке с помощью ключевого слова `stackalloc`. Из-за распределения в стеке время жизни блока памяти ограничивается временем выполнения метода, в точности как для любой другой локальной переменной. Блок может использовать оператор `[]` для индексации внутри памяти:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
    Console.WriteLine(a[i]); // Вывод из памяти
```

Буфера 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];
    }
}
```

Буфера фиксированных размеров не являются массивами: 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` используется также для закрепления в куче объекта, содержащего буфер (который будет экземпляром `UnsafeClass`).

void*

Указатель на `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:

```
short[] a = {1,1,2,3,5,8,13,21,34,55};
fixed (short* p = a)
{
    //sizeof возвращает размер типа-значения в байтах
    Zap(p, a.Length * sizeof (short));
}

foreach (short x in a)
    System.Console.WriteLine (x); // Вывод нулей
```

```
unsafe void Zap(void* memory, int byteCount)
{
    byte* b = (byte*) memory;
        for (int i = 0; i < byteCount; i++)
            *b++ = 0;
}
```

Указатели на функции (C# 9)

Указатель на функцию схож с делегатом, но без косвенности в виде экземпляра делегата; вместо этого он указывает непосредственно на метод. Указатель на функцию может указывать только на статические методы, без возможности многоадресной рассылки и требует контекст `unsafe` (потому что при этом не работает безопасность типов времени выполнения). Его основная цель — упростить и оптимизировать взаимодействие с помощью неуправляемых API (это взаимодействие детально рассматривается в книге *C# 9.0. Справочник. Полное описание языка*).

Тип указателя функции объявляется следующим образом (тип возвращаемого значения указывается последним):

```
delegate*<int, char, string, void>
```

Это объявление соответствует функции со следующей сигнатурой:

```
void SomeFunction (int x, char y, string z)
```

Указатель на функцию создается с помощью оператора &. Вот полный пример:

```
unsafe
{
    delegate*<string, int> functionPointer = &GetLength;
    int length = functionPointer ("Hello, world");

    static int GetLength (string s) => s.Length;
}
```

В этом примере `functionPointer` не является *объектом*, для которого вы можете вызвать такой метод, как `Invoke` (или с помощью ссылки на объект `Target`). Это переменная, которая указывает непосредственно на адрес целевого метода в памяти:

```
Console.WriteLine((IntPtr)functionPointer);
```

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 данном классе инструкция внутри метода `Foo()` компилируется *условно*, в зависимости наличия определенного символа `DEBUG`. Если удалить определение символа `DEBUG`, эта инструкция в `Foo()` компилироваться не будет. Символы препроцессора могут определяться внутри файла исходного кода (как сделано в рассматриваемом примере), а также передаваться компилятору в командной строке (`/define:`*`символ`*) либо в файле проекта в случае использования Visual Studio или MSBuild.

В директивах `#if` и `#elif` можно применять операторы `||`, `&&` 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.

В табл. 21 перечислены все директивы препроцессора.

Таблица 21. Директивы препроцессора

Директива препроцессора	Действие
`#define` *символ*	Определяет *символ*
`#undef` *символ*	Отменяет определение *символа*
`#if` *символ* [*оператор символ2*]	Условная компиляция (*операторами* являются `==`, `!=`, `&&`, `||`)
`#endif`	Завершение директивы условной компиляции
`#warning` *текст*	Заставляет компилятор вывести *текст предупреждения*
`#error` *текст*	Заставляет компилятор вывести *текст ошибки*
`#line` [*номер* ["*файл*"]|hidden]	*Номер* задает строку исходного кода, *файл* — имя файла, выводимое компилятором, `hidden` указывает отладчику пропустить код от этой точки до следующей директивы `#line`
`#region` *имя*	Обозначает начало области
`#endregion`	Обозначает конец области
`#pragma warning`	См. следующий раздел книги
`#nullable` *параметр*	См. раздел "Ссылочные типы, допускающие значение null"

Директива `#pragma warning`

Компилятор генерирует предупреждение, когда обнаруживает в коде что-то, кажущееся ему непреднамеренной опиской. В отличие от ошибок предупреждения обычно не препятствуют компиляции приложения.

Предупреждения компилятора могут быть исключительно полезными при выявлении ошибок, но их полезность снижается в случае выдачи ложных предупреждений. В большом приложении

очень важно поддерживать подходящее отношение "сигнал/шум", чтобы были замечены "настоящие" предупреждения.

С этой целью компилятор позволяет избирательно подавлять выдачу предупреждений с помощью директивы `#pragma warning` там, где вы точно знаете, что*о* делаете. В следующем примере мы указываем компилятору, что выдавать предупреждения о том, что поле `Message` не используется, не нужно:

```
public class Foo
{
    #pragma warning disable 414
    static string Message = "Hello";
    #pragma warning restore 414
}
```

Если в директиве `#pragma warning` отсутствует конкретное числовое значение, то будет отключена (или восстановлена) выдача всех предупреждений, с любыми кодами.

Если вы интенсивно применяете эту директиву, то можете скомпилировать код с переключателем командной строки `/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() { ... }
```

При компиляции с переключателем командной строки `/doc` (или при включении генерации XML-документации в файле проекта) компилятор извлекает и накапливает документирующие комментарии в специальном XML-файле. Имеется два основных сценария использования такого файла.

- ✓ Если он размещен в той же папке, что и скомпилированная сборка, то IntelliSense Visual Studio автоматически читает этот XML-файл и использует содержащуюся в нем информацию для предоставления списка членов пользователям сборки с тем же именем, что и у XML-файла.
- ✓ Сторонние инструменты (такие, как Sandcastle и NDoc) могут преобразовать этот XML-файл в справочный HTML-файл.

Стандартные дескрипторы документации

Ниже перечислены стандартные XML-дескрипторы, которые распознаются Visual Studio и генераторами документации.

`<summary>`

`<summary>...</summary>`

Указывает всплывающую подсказку, которую IntelliSense отображает 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>`

Пояснения к параметру метода.

<returns>

```
<returns>...</returns>
```

Пояснения к возвращаемому значению метода.

<exception>

```
<exception [cref="тип"]>...</exception>
```

Указывает исключение, которое может генерировать данный метод (в `cref` задается тип исключения).

<permission>

```
<permission [cref="тип"]>...</permission>
```

Указывает тип `IPermission`, требуемый документируемым типом или членом.

<example>

```
<example>...</example>
```

Описывает 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 этот дескриптор в гиперссылку. Компилятор выдает предупреждение, если указано некорректное имя типа или члена.

<seealso>

```
<seealso cref="член">...</seealso>
```

Вставляет перекрестную ссылку на другой тип или член. Генераторы документации обычно записывают ее 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>
```

Уведомляет генератор документации о необходимости генерации маркированного (bullet), нумерованного (number) или табличного (table) списка.

<para>

```
<para>...</para>
```

Уведомляет 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