Содержание

Об авторах 7
Об изображении на обложке 8
Ждем ваших отзывов! 9

Язык C# 9.0. Карманный справочник 11

Первая программа на C# 11
Синтаксис 14
Типы в C# 17
Числовые типы 28
Логический тип и операторы 35
Строки и символы 37
Массивы 41
Переменные и параметры 47
Выражения и операторы 56
`null`-операторы 62
Инструкции 64
Пространства имен 74
Классы 78
Наследование 96
Тип `object` 105
Структуры 109
Модификаторы доступа 111
Интерфейсы 113
Перечисления 118
Вложенные типы 121
Обобщения 122
Делегаты 131
События 137
Лямбда-выражения 143
Анонимные методы 148
Инструкции `try` и исключения 149
Перечисление и итераторы 157
Типы-значения, допускающие `null` 162
Расширяющие методы 170
Анонимные типы 172

Кортежи 173
Записи (C# 9) 175
Сопоставление с образцом 182
LINQ 186
Динамическое связывание 212
Перегрузка операторов 220
Атрибуты 224
Атрибуты информации о вызывающем компоненте 228
Асинхронные функции 229
Небезопасный код и указатели 241
Директивы препроцессора 246
XML-документация 248

Предметный указатель 253

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; }
```

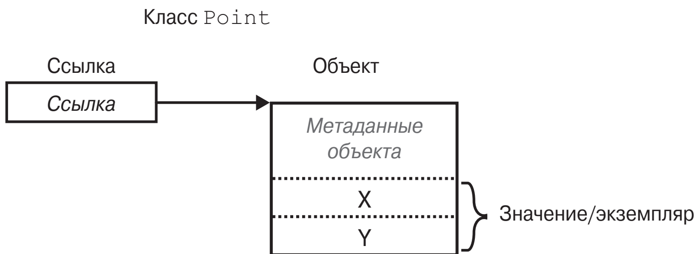

Рис. 3. *Экземпляр ссылочного типа в памяти*

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.

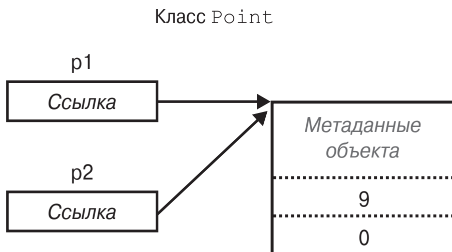

Рис. 4. *Экземпляр ссылочного типа в памяти*

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
```