Оглавление

[**C# *Джозеф Албахари и Бен Албахари*** 2](#_Toc182856533)

[Глава 2. Основы языка C# 3](#_Toc182856534)

[Глава 3. Создание типов 31](#_Toc182856535)

[Глава 17. Сериализация 52](#_Toc182856536)

# **C# *Джозеф Албахари и Бен Албахари***

## Глава 2. Основы языка C#

1. Какие из нижеследующих описаний метода Main, как точки входа в приложение, являются корректными:

* static void Main(),
* private void Main(),
* private int Main(),
* void Main(string[] args).

Метод по имени Main распознается в С# как признак стандартной точки входа в поток выполнения. Метод Main может необязательно возвращать целочисленное значение (вместо void) с целью его передачи исполняющей среде (при этом ненулевое значение обычно указывает на ошибку). Кроме того, метод Main может необязательно принимать в качестве параметра массив строк (который будет заполняться любыми аргументами, передаваемыми исполняющему файлу).

1. Что такое сборка (assembly):

* это единица упаковки и развертывания в .NET,
* набор \*.cs файлов одного проекта в MS Visual Studio,
* набор \*.cs файлов одного solution в MS Visual Studio,
* один \*.cs файл.

Компилятор С# транслирует исходный код, указываемый в виде набора файлов с расширением . cs, в сборку. Сборка (assembly) - это единица упаковки и развертывания в .NET. Сборка может быть либо приложением, либо библиотекой. Нормальное консольное или Windоws-приложение имеет метод Main и является файлом .ехе.

Библиотека представляет собой файл .dll и эквивалентна файлу .ехе без точки входа. Она предназначена для вызова (ссылки) приложением или другими библиотеками. Инфраструктура .NET Framework - это набор библиотек.

Идентификаторы - это имена, которые программисты выбирают для своих классов, методов, переменных и т.д. Идентификатор должен быть целостным словом, которое по существу состоит из символов Unicode и начинается с буквы или символа подчеркивания.

Идентификаторы С# чувствительны к регистру символов. По соглашению для параметров, локальных переменных и закрытых полей должен применяться "верблюжий" стиль (скажем, myVariable), а для всех остальных идентификаторов - стиль Pascal (вроде MyMethod).

Ключевые слова представляют собой имена, которые имеют для компилятора особый смысл (например, class). Большинство ключевых слов являются зарезервированными, а это означает, что их нельзя использовать в качестве идентификаторов.

Некоторые ключевые слова являются контекстными, а это значит, что их можно использовать также в качестве идентификаторов - без символа@. Неоднозначность с контекстными ключевыми словами не может возникать внутри контекста, в котором они применяются.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

int i = 13;

string hello = "Hello, world!";

int[] array = { 1, 2, 3, 4, 5 };

}

}

Какие пространства имен необходимо подключить, чтобы приложение было работоспособным:

* никакие,
* System,
* System.Text,
* System и System.Text.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

int i = 13;

string hello = "Hello, world!";

DateTime date = new DateTime(2021, 12, 31);

}

}

Какие пространства имен необходимо подключить, чтобы приложение было работоспособным:

* никакие,
* System,
* System.Text,
* System и System.Text.

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 -231 до 231 -1.

Предопределенные типы в С# (также называемые встроенными типами) распознаются по ключевым словам С# (например, int, char, string, true). Пространство имен System в .NET Framework содержит много важных типов, которые не являются предопределенными в языке С# (скажем, DateTime).

Предопределенные типы в С# классифицируются следующим образом.

Типы значений

• Числовой

* целочисленный со знаком (sbyte, short, int, long)
* целочисленный без знака (byte, ushort, uint, ulong)
* вещественный (float, double, decimal)

• Логический (bool)

• Символьный (char)

Cсылочные типы:

• Строка (string)

• Объект (object)

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

int i = 5;

System.Int32 i = 5;

Набор предопределенных типов значений, исключая decimal, известен в CLR как примитивные типы. Примитивные типы называются так потому, что они поддерживаются непосредственно через инструкции в скомпилированном коде, которые обычно транслируются в прямую поддержку внутри имеющегося процессора.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

Console.Write("Input number: ");

string input = Console.ReadLine();

int number;

if (int.TryParse(input, out number))

{

ShowNumber(number);

}

}

private void ShowNumber(int number)

{

Console.WriteLine($"Your number is {number}.");

}

}

Возникнет ли синтаксическая ошибка в результате работы этого кода?

Данные-члены и функции-члены, которые оперируют на экземпляра типа, называются членами экземпляра.

Данные-члены и функции-члены, которые не оперируют на экземпляре типа, а вместо этого имеют дело с самим типом, должны помечаться как static. Примерами статических методов являются Main и Console.WriteLine. Класс Console в действительности представляет собой статический класс, а это значит, что все его члены являются статическими. Создавать экземпляры класса Console никогда не придется - одна консоль совместно используется всем приложением.

1. Дана следующая переменная:

int i = 13045;

Какие из следующих строк записаны корректно:

1. long x = i;
2. short y = i;
3. double z = i;

* 1) и 3),
* 1) и 2),
* 2) и 3),
* все записаны корректно,
* все записаны некорректно.

В С# возможны преобразования между экземплярами совместимых типов.

Преобразование всегда приводит к созданию нового значения из существующего.

Преобразования могут быть либо неявными, либо явными: неявные преобразования происходят автоматически, в то время как явные преобразования требуют приведения.

Неявные преобразования разрешены, когда удовлетворяются перечисленные ниже условия:

• компилятор может гарантировать, что они всегда проходят успешно;

• в результате преобразования никакая информация не теряется.

Целочисленные преобразования являются неявными, когда целевой тип в состоянии представить каждое возможное значение исходного типа. В противном случае требуется явное преобразование.

1. С помощью какого ключевого слова можно определить специальный тип значения:

* struct,
* int,
* value,
* тип значения может быть только встроенным и его никак нельзя определить.

1. Где хранятся переменные типа значения:

* в стеке (stack),
* в куче (heap).

Все типы С# делятся на следующие категории:

• типы значений;

• ссылочные типы;

• параметры типа в обобщениях;

• типы указателей.

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

1. Задана следующая структура:

public struct Point

{

public int Х;

public int Y;

}

Затем созданы два экземпляра этой структуры:

Point p1 = new Point();

p1.X = 2;

p1.Y = 3;

Point р2 = p1;

Затем значение для p2 было изменено:

p2.X = 5;

Что будет выведено на экран при выполнении следующей функции

Console.WriteLine(p1.X + “, ” + p2.X)?

* 2, 5;
* 2, 3;
* 2, 2.

Присваивание экземпляра типа значения всегда приводит к копированию этого экземпляра.

1. Где хранятся переменные ссылочного типа:

* в куче (heap),
* в стеке (stack).

1. Задан следующий класс:

public class Point

{

public int Х;

public int Y;

}

Затем созданы два экземпляра этого класса:

Point p1 = new Point();

p1.X = 2;

p1.Y = 3;

Point р2 = p1;

Затем значение для p2 было изменено:

p2.X = 5;

Что будет выведено на экран при выполнении следующей функции

Console.WriteLine(p1.X + “, ” + p2.X)?

* 5, 5;
* 2, 5;
* 2, 2.

Ссылочный тип сложнее типа значения из-за наличия двух частей: объекта и ссылки на этот объект. Содержимым переменной или константы ссылочного типа является ссылка на объект, который содержит значение.

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

1. Заданы следующие структура и класс:

public struct Point

{

public int Х;

public int Y;

}

public class Coordinate

{

public int Х;

public int Y;

}

Какая из записей корректна:

1. Point p = null;
2. Coordinate c = null;

* запись 1) корректна,
* запись 2) корректна,
* обе записи корректны.

Ссылке может быть присвоен литерал null, который отражает тот факт, что ссылка не указывает на какой-либо объект. В противоположность этому тип значения обычно не может иметь значение null (за исключением специальной конструкции под названием «типы, допускающие значение null»).

1. Заданы следующие структура и класс:

public struct Point

{

public int Х;

public int Y;

}

public class Coordinate

{

public int Х;

public int Y;

}

Затем созданы два массива этих типов:

Point[] points = new Points[100];

Coordinate[] coordinates = new Coordinate[100];

Какой из этих массивов будет занимать больше памяти?

* массив points будет занимать больше памяти,
* массив coordinates будет занимать больше памяти,
* объем памяти, занимаемый обоими массивами, будет примерно одинаков.

Экземпляры типов значений занимают в точности столько памяти, сколько требуется для хранения их полей.

Ссылочные типы требуют раздельного выделения памяти для ссылки и объекта. Объект потребляет столько памяти, сколько необходимо его полям, плюс дополнительный объем на административные нужды. Точный объем накладных расходов по существу зависит от реализации исполняющей среды .NET, но составляет минимум восемь байтов, которые используются для хранения ключа к типу объекта, а также временной информации, такой как его состояние блокировки для многопоточной обработки и флаг для указания, является ли объект закрепленным от перемещения сборщиком мусора. Каждая ссылка на объект требует дополнительных четырех или восьми байтов в зависимости от того, на какой платформе функционирует исполняющая среда .NET - 32- или 64-разрядной.

1. Являются ли значения, записанные в строках ниже, эквивалентными:

int i = 19;

int i = 0x13;

Целочисленные литералы могут использовать десятичную или шестнадцатеричную форму записи; шестнадцатеричная форма записи предусматривает применение префикса 0х.

1. Являются ли значения, записанные в строках ниже, эквивалентными:

double d = 102.5;

double d = 1.025E02;

Вещественные литералы могут использовать десятичную и/или экспоненциальную форму записи.

1. Какие из следующих выражение записаны некорректно:
2. float f = 4.5;
3. double d = 1.025E02;
4. decimal d = -1.2З;

* корректно записано 2),
* корректно записаны 1) и 2),
* корректно записаны 2) и 3),
* все записаны корректно,
* все записаны некорректно.

Числовые суффиксы явно определяют тип литерала. Суффиксы могут записываться либо строчными, либо прописными буквами; все они перечислены ниже:

F float

D double

M decimal

U uint

L long

UL ulong

Суффикс D формально является избыточным из-за того, что все литералы с десятичной точкой выводятся в double. И к числовому литералу всегда можно добавить десятичную точку: double х = 4.0;

Суффиксы F и М наиболее полезны и всегда должны применяться при указании литералов float или decimal. Без суффикса F следующая строка кода не скомпилируется, т.к. значение 4.5 выводится в тип double, для которого не существует неявного преобразования в float:

float f = 4.5F;

Тот же принцип справедлив для десятичных литералов (не будет компилироваться без суффикса М):

decimal d = -1.2ЗМ;

1. Даны следующие переменные:

int i = 147;

float f = 1.045F;

Какие из следующих строк записаны корректно:

1. float x = d; int y = d;
2. float x = i; double y = i;
3. decimal x = i; decimal y = f;

* 1) и 3),
* 1) и 2),
* 2) и 3),
* все записаны корректно,
* все записаны некорректно.

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

Все целочисленные типы могут быть неявно преобразованы во все типы с плавающей точкой, но обратное преобразование должно быть явным.

Когда производится приведение числа с плавающей точкой к целому, дробная часть отбрасывается; никакого округления не производится.

Все целочисленные типы могут быть неявно преобразованы в тип decimal, поскольку decimal способен представлять любое возможное целочисленное значение в С#. Все остальные числовые преобразования в и из типа decimal должны быть явными.

Небольшое предостережение: очень высокие значения long после преобразования в double теряют в точности.

1. Что будет выведено на экран в результате работы следующего кода:

int x = 0;

int y = 0;

Console.Write(х++);

Console.Write(х);

Console.Write(“-”);

Console.Write(++у);

Console.Write(y);

* 01-11,
* 11-11,
* 11-12,
* 01-01.

Операции инкремента и декремента ( ++, --) увеличивают и уменьшают значения числовых типов на 1. Эти операции могут находиться перед или после имени переменной в зависимости от того, когда требуется обновить значение переменной – до или после вычисления выражения.

1. В каком случае будет обязательно сгенерировано исключение:

checked

{

int min = int.MinValue;

min--;

}

int min = int.MinValue;

min--;

* в первом случае,
* во втором случае,
* в обоих случаях,
* ни в одном случае.

Операция checked сообщает исполняющей среде о том, что вместо молчаливого переполнения она должна генерировать исключение OverflowException, когда целочисленное выражение или оператор выходит за арифметические пределы этого типа. Операция checked воздействует на выражения с операциями ++, --, +, - (бинарной и унарной), \*, / и явными преобразованиями между целочисленными типами.

Операция checked не оказывает никакого влияния на типы double и float (которые получают при переполнении специальные значения "бесконечности") и на тип decimal (который проверяется всегда).

Операцию checked можно использовать либо с выражением, либо с блоком операторов.

1. Что будет выведено в результате работы этого кода:

double d = 1.0 / 0.0;

if (double.IsPositiveInfinity(d))

{

Console.WriteLine("It is positive infinity");

}

else

{

Console.WriteLine("It is not positive infinity");

}

* It is positive infinity,
* It is not positive infinity,
* возникнет синтаксическая ошибка.

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

1. Что будет выведено в результате работы этого кода:

float tenth = 0.1f;

float one = 1f;

float result = one - tenth \* 10f;

if (result == 0)

{

Console.WriteLine("It is zero");

}

else

{

Console.WriteLine("It is not zero");

}

* It is zero,
* It is not zero,
* возникнет синтаксическая ошибка.

Типы float и double внутренне представляют числа в двоичной форме. По этой причине точно представляются только числа, которые могут быть выражены в двоичной системе счисления. На практике это означает, что большинство литералов с дробной частью (которые являются десятичными) не будут представлены точно. Именно поэтому типы float и double не подходят для финансовых вычислений.

В противоположность им тип decimal работает в десятичной системе счисления и потому способен точно представлять дробные числа вроде 0.1, выразимые в десятичной системе (а также в системах счисления с основаниями-множителями 10- двоичной и пятеричной). Поскольку вещественные литералы являются десятичными, тип decimal может точно представлять такие числа, как 0.1. Тем не менее, ни double, ни decimal не могут точно представить дробное число, десятичное представление которого является периодическим, например, 1 / 3.

1. Заданы следующие переменные:

int a = 5;

int b = 7;

bool compare = a > b;

Какие строки записаны корректно:

1. string result = compare == 1 ? "more" : "less";
2. string result = compare == 0 ? "less" : "more";
3. string result = compare ? " more" : " less";

* 1) и 3),
* 1) и 2),
* 3),
* все записаны корректно,
* все записаны некорректно.

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

Хотя для хранения булевского значения (тип bool – это псевдоним типа System.Boolean) достаточно только одного бита, исполняющая среда будет использовать один байт памяти, т.к. это минимальная порция, с которой исполняющая среда и процессор могут эффективно работать. Во избежание непродуктивных затрат пространства в случае массивов платформа .NET Framework предлагает в пространстве имен System.Collections класс BitArray, который позволяет задействовать по одному биту для каждого булевского значения в массиве.

Никакие приведения и преобразования из типа bool в числовые типы и наоборот не разрешены.

Тернарная операция (она единственная принимает три операнда) имеет вид q ? а : b, где результатом является а, если условие q равно true, и b - в противном случае.

1. Корректно ли следующее выражение:

int a = 15 + ‘Z’;

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

1. Что будет выведено в результате работы этого кода:

string hello = "Hello";

string goodbye = "Goodbye";

if (hello == goodbye)

{

Console.WriteLine("These strings are equal");

}

if (hello < goodbye)

{

Console.WriteLine($"String { hello} less ");

}

if (hello > goodbye)

{

Console.WriteLine($"String {hello} less ");

}

* It is zero,
* It is not zero,
* возникнет синтаксическая ошибка.

1. Возникнет ли ошибка в результате работы следующего кода:

int i = 17;

Console.WriteLine($"This is a number {i}");

String - это ссылочный тип, а не тип значения. Тем не менее, его операции эквивалентности следуют семантике типов значений.

Тип string не поддерживает операции < и > для сравнений. Вместо них должен применяться метод CompareTo типа string.

В С# разрешены дословные строковые литералы. Дословный строковый литерал снабжается префиксом @ и не поддерживает управляющие последовательности.

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

Внутри скобок может быть указано любое допустимое выражение С# произвольного типа, и С# преобразует это выражение в строку, вызывая ToString или эквивалентный метод этого типа.

1. Возникнет ли ошибка в результате работы следующего кода:

int age = 17;

Console.WriteLine("I’m " + i + " years old.");

1. Возникнет ли ошибка в результате работы следующего кода:

int x = 15;

long y = x;

double z = x;

Console.WriteLine(x + y + z);

Операция + выполняет конкатенацию двух строк: string s = "а" + "b";

Один из операндов может быть нестроковым значением; в этом случае для него будет вызван метод ToString.

1. Возникнет ли ошибка в результате работы следующего кода:

int[] array = { 1, 2, 3, 4, 5 };

array.Length = 10;

Массив представляет фиксированное количество переменных (называемых элементами) заданного типа. Элементы массива всегда хранятся в непрерывном блоке памяти, обеспечивая высокоэффективный доступ.

1. Возникнет ли ошибка в результате работы следующего кода (или что будет выведено на консоль):

int[] array = new int[10];

Console.WriteLine(array[5]);

1. Возникнет ли ошибка в результате работы следующего кода (или что будет выведено на консоль):

struct Point { public X, Y; }

Point[] points = new Point[10];

Console.WriteLine(points[5].X);

1. Возникнет ли ошибка в результате работы следующего кода (или что будет выведено на консоль):

class Point { public X, Y; }

Point[] points = new Point[10];

Console.WriteLine(points[5].X);

Значительное влияние на производительность оказывает то, какой тип имеют элементы массива - тип значения или ссылочный тип. Если элементы относятся к типу значения, то пространство под значение каждого элемента выделяется как часть массива.

В ситуации, когда Point является классом, создание массива приводит просто к выделению пространства под 1000 ссылок null.

1. Возникнет ли ошибка в результате работы следующего кода:

struct Point { public X, Y; }

Point[] points = null;

int[] array = null;

Независимо от типа элементов массив сам по себе всегда является объектом ссылочного типа.

Переменная представляет ячейку в памяти, которая содержит изменяемое значение. Переменная может быть локальной переменной, параметром (value, ref либо out), полем (экземпляра либо статическим) или элементом массива.

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

Стек представляет собой блок памяти для хранения локальных переменных и параметров. Стек логически увеличивается при входе в функцию и уменьшается после выхода из нее.

Куча - это блок памяти, в котором располагаются объекты (т.е. экземпляры ссылочного типа). Всякий раз, когда создается новый объект, он размещается в куче с возвращением ссылки на этот объект. Во время выполнения программы куча начинает заполняться по мере создания новых объектов. В исполняющей среде предусмотрен сборщик мусора, который периодически освобождает объекты из кучи, так что программа не столкнется с ситуацией нехватки памяти. Объект становится пригодным для освобождения, если на него отсутствуют ссылки.

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.

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

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static int y;

static void Main(string[] args)

{

int x;

1. Console.WriteLine(x);
2. Console.WriteLine(y);

int[] numbers = new int[5];

1. Console.WriteLine(numbers[4]);
2. Console.WriteLine(numbers[5]);

}

}

Возникнут ли синтаксические ошибки в строках 1) – 4)?

* возникнет в строках 1) и 4),
* возникнет в строке 1),
* возникнет в строках 1) и 2),
* возникнет в строках 1), 2) и 4),
* не возникнет ни в одной строке,
* возникнут во всех четырех строках.

В С# принудительно применяется политика определенного присваивания. Определенное присваивание приводит к трем последствиям:

* + локальным переменным должны быть присвоены значения перед тем, как их можно будет читать.
  + при вызове метода должны быть предоставлены аргументы функции (если только они не помечены как необязательные).
  + все остальные переменные (такие как поля и элементы массивов) автоматически инициализируются исполняющей средой стандартными значениями для своих типов.

Получить стандартное значение для любого типа можно с помощью ключевого слова default (на практике оно применяется при работе с обобщениями):

decimal d = default (decimal);

Стандартное значение в специальном типе значения (т.е. struct) - это то же самое, что и стандартные значения для всех полей, определенных этим специальным типом.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

int number = 13;

ShowIncrementNumber(number);

Console.WriteLine($"Original number is {number}.");

}

static void ShowIncrementNumber(int number)

{

number++;

Console.WriteLine($"Increment number is {number}.");

}

}

Какое число будет выведено в строке increment number и какое в строке original number?

* 14 и 13,
* 14 и 14,
* 13 и 14,
* 13 и 13.

По умолчанию аргументы в С# передаются по значению, что общепризнанно является самым распространенным случаем. Это означает, что при передаче методу создается копия значения.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

string hello = "Hello";

ShowMessage(hello);

Console.WriteLine(hello);

}

static void ShowMessage(string message)

{

message += ", world!";

Console.WriteLine(message);

}

}

Что будет выведено на экран?

* "Hello, world!" и "Hello",
* "Hello, world!" и "Hello, world!",
* "Hello" и "Hello, world!".

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

StringBuilder hello = new StringBuilder("Hello");

ShowMessage(hello);

Console.WriteLine(hello);

}

static void ShowMessage(StringBuilder message)

{

message.Append(", world!");

Console.WriteLine(message);

}

}

Что будет выведено на экран?

* "Hello, world!" и "Hello, world!",
* "Hello, world!" и "Hello",
* "Hello" и "Hello, world!".

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

StringBuilder hello = new StringBuilder("Hello");

ResetString(hello);

Console.WriteLine(hello);

}

static void ResetString(StringBuilder message)

{

message.Append(", world!");

message = null;

}

}

Что будет выведено на экран?

* "Hello, world!",
* "Hello",
* пустая строка.

Передача по значению аргумента ссылочного типа приводит к копированию ссылки, но не объекта.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

int number = 13;

ShowIncrementNumber(ref number);

Console.WriteLine($"Original number is {number}.");

}

static void ShowIncrementNumber(ref int number)

{

number++;

Console.WriteLine($"Increment number is {number}.");

}

}

Какое число будет выведено в строке increment number и какое в строке original number?

* 14 и 14,
* 14 и 13,
* 13 и 14,
* 13 и 13.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

static void Main(string[] args)

{

StringBuilder hello = new StringBuilder("Hello");

ResetString(ref hello);

Console.WriteLine(hello);

}

static void ResetString(ref StringBuilder message)

{

message = null;

}

}

Что будет выведено на экран?

* "Hello",
* пустая строка,
* ничего.

Для передачи по ссылке в С# предусмотрен модификатор параметра ref. Передавая аргумент по ссылке, вы устанавливаете псевдоним для ячейки памяти, в которой находится существующая переменная, а не создаете новую ячейку.

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

1. Есть следующий метод:

private void SplitName(string fullName, out string firstName, out string lastName)

{

int i = fullName.LastIndexOf(' ');

if (index != -1)

{

firstName = fullName.Substring(0, index);

lastName = fullName.Substring(index + 1);

}

}

Что произойдет при вызове

SplitName("John Lennon", out firstName, out lastName);

* возникнет синтаксическая ошибка,
* firstName = "John Lennon", lastName = "Lennon".

Аргумент out похож на аргумент ref за исключением следующих аспектов:

• он не нуждается в присваивании значения перед входом в функцию;

• ему должно быть присвоено значение перед выходом из функции.

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

Как и параметр ref, параметр out передается по ссылке.

1. Есть следующий метод:

private int Sum(params int[] numbers)

{

int sum = 0;

for (int i = 0; i < numbers.Length; i++)

{

sum += numbers[i];

}

return sum;

}

Какой способ вызова этого метода правильный (что вернет функция в каждом случае):

* int total = Sum(1, 2, 3, 4, 5);
* int total = Sum(0);
* int total = Sum('z');
* int total = Sum(new int[] { 1, 2, 3, 4, 5 });
* int total = Sum(new int[5]);
* int total = Sum(new int[0]);.

Модификатор params может быть указан для последнего параметра метода, чтобы позволить методу принимать любое количество аргументов заданного типа. Тип параметра должен быть объявлен как массив.

1. Есть следующий метод:

private void ShowMessage(string message, string header = "Header")

{

Console.WriteLine(header);

Console.WriteLine(message);

}

Какой способ вызова этого метода правильный:

* ShowMessage("John Lennon", "The Beatles");
* ShowMessage("John Lennon");
* оба способа правильны.

Начиная с версии С# 4.0, в методах, конструкторах и индексаторах можно объявлять необязательные параметры. Параметр является необязательным, если в его объявлении указано стандартное значение.

Стандартное значение необязательного параметра должно быть указано в виде константного выражения или вызова конструктора без параметров для типа значения.

Необязательные параметры не могут быть помечены посредством ref или out.

Обязательные параметры должны находиться перед необязательными параметрами в объявлении метода и при его вызове (исключением являются аргументы params, которые всегда располагаются в конце).

1. Есть следующий метод:

private void Polynomial (int а = 0, int b = 0, int c = 0, int d = 0)

{

…

}

Какой способ вызова этого метода правильный:

* Polynomial(d: 5, c: 3);
* Polynomial(a: 7);
* Polynomial();
* все способы правильны.

Вместо распознавания аргумента по позиции его можно идентифицировать по имени. Именованные аргументы могут указываться в любом порядке. Именованные и позиционные аргументы можно смешивать. Тем не менее, существует одно ограничение: позиционные аргументы должны находиться перед именованными аргументами. Именованные аргументы особенно удобны в сочетании с необязательными параметрами.

1. Что будет выведено на экран в результате выполнения следующего кода:

int x = 13 \* (x = 4);

Console.WriteLine(x);

* 52;
* 13;
* 4;
* возникнет синтаксическая ошибка.

Выражение присваивания - это не пустое выражение. На самом деле оно заключает в себе присваиваемое значение и потому может встраиваться в другое выражение.

Бинарные операции (кроме операций присваивания, лямбда-операции и операций объединения с null) являются левоассоциативными; другими словами, они вычисляются слева направо (например, 8 / 4 / 2).

Операции присваивания, лямбда-операция, операция объединения с null и условная операция являются правоассоциативными; другими словами, они вычисляются справа налево. Правая ассоциативность позволяет успешно компилировать множественное присваивание, например, х = у = 3.

1. Чему равно значение line:

string name = null;

string line = name ?? "NoName";

* NoName;
* null;
* возникнет синтаксическая ошибка.

1. Чему равно значение line:

string name1 = "NoName";

string name2 = null;

string line = name1 ?? name2.Substring(0, 1);

* NoName;
* null;
* возникнет синтаксическая ошибка.

Операция обоединения с null (null coalescing) обозначается как «??». Она выполняется следующим образом: если операнд не равен null, то возвратить eгo значение; в противном случае возвратить стандартное значение.

Если левостороннее выражение не равно null, то правостороннее выражение никогда не вычисляется.

1. Чему равно значение line:

string name = null;

string line = name?.Substring(0, 1).ToUpper();

* null;
* возникнет синтаксическая ошибка.

1. Чему равно значение line:

string name = null;

int length = name?.Substring(0, 1).Length;

* null;
* 0;
* возникнет синтаксическая ошибка.

Null-условнaя (null-conditional) операция или элвис-операция обозначается как «?.» и является нововведением версии С# 6. Она позволяет вызывать методы и получать доступ к членам подобно стандартной операции точки, но с той разницей, что если находящийся слева операнд равен null, то результатом выражения будет null, а не генерация исключения NullReferenceException.

1. Что будет выведено на экран в результате работы следующего кода, если сегодня 1 сентября 2021 года:

DateTime today = DateTime.Now;

DateTime date1 = new DateTime(2021, 01, 01);

DateTime date2 = new DateTime(2020, 09, 01);

switch (today)

{

case date1:

Console.WriteLine("Happy New Year!");

break;

case date2:

Console.WriteLine("Make sure you study well!");

break;

default:

Console.WriteLine("Good luck!");

break;

}

* Make sure you study well!;
* Good luck!;
* возникнет синтаксическая ошибка.

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

В операторе switch можно указывать только выражение с типом, который может быть оценен статически, что ограничивает его встроенными целочисленными типами, типом bool, типами enum (а также их версиями, допускающими) и типом string.

1. Что будет выведено на экран в результате работы следующего кода:

for (int i = 0; i < 4 | i % 2 == 0; i++)

{

Console.Write(i);

}

* 01234;
* 0123;
* возникнет синтаксическая ошибка.

1. Может ли возникнуть конфликт имен namespace, если их объявить в разных модулях (файлах), но в одном проекте?

Пространство имен - это область, предназначенная для имен типов. Ключевое слово namespace определяет пространство имен для типов внутри данного блока. Типы обычно организуются в иерархические пространства имен, упрощая их поиск и устраняя возможность конфликтов. С помощью точек отражается иерархия вложенных пространств имен.

Пространства имен не зависят от сборок, которые являются единицами развертывания, такими как . ехе или . dll.

Пространства имен также не влияют на видимость членов – public, internal, private и т.д.

Директива using импортирует пространство имен, позволяя ссылаться на типы без указания их полностью заданных имен. Все имена типов во время компиляции преобразуются в полностью заданные имена. Неполные или частично заданные имена в коде на промежуточном языке (lntermediate Language - IL) отсутствуют.

Глобальное пространство имен - корень всех пространств имен (идентифицируется контекстным ключевым словом global). Для указания псевдонима пространства имен используется маркер «::» .

В версии С# 6 появилась возможность импортировать не только пространство имен, но и отдельный тип с помощью директивы using static. После этого все статические члены данного типа могут использоваться без их снабжения именем типа.

## Глава 3. Создание типов

1. Что такое поле:

* переменная, которая является членом класса,
* переменная, которая является членом структуры,
* оба утверждения верны.

Поле - это переменная, которая является членом класса или структуры. С полями разрешено применять следующие модификаторы:

* + статический модификатор static,
  + модификаторы доступа: public, internal, private, protected.
  + модификатор наследования new,
  + модификатор небезопасного кода unsafe,
  + модификатор доступа только для чтения readonly,
  + модификатор многопоточности volatile.

1. При каком определении класса возникнет синтаксическая ошибка:

class MyClass

{

private readonly int versionNumber = 1;

}

class MyClass

{

private readonly int versionNumber = 1;

public MyClass(int version)

{

versionNumber = version;

}

}

class MyClass

{

private readonly int versionNumber = 1;

public MyClass(int version)

{

versionNumber = version;

}

public void SetVersionNumber(int version)

{

versionNumber = version;

}

}

* 1,
* 1 и 2,
* 2 и 3,
* 3,
* все объявления класса синтаксически верны.

Модификатор readonly предотвращает изменение поля после его создания. Присваивать значение полю, допускающему только чтение, можно только в его объявлении или внутри конструктора типа, в котором оно определено.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

class MyClass

{

public int X;

}

static void Main(string[] args)

{

MyClass myClass = new MyClass();

Console.WriteLine(myClass.X);

}

}

Что будет выведено на экран?

* 0,
* пустая строка,
* возникнет синтаксическая ошибка.

Инициализация полей является необязательной. Неинициализированное поле получает свое стандартное значение (0, \0, null, false). Инициализаторы полей выполняются перед конструкторами.

1. В рамках одного типа был объявлен метод: void Foo(int x). Какие из следующих методов являются допустимыми для объявления их в том же классе:
2. void Foo(double x) {}
3. void Foo(string x) {}
4. void Foo(int y) {}
5. int Foo(int x) {}
6. void Foo(int x, int y) {}
7. void Foo(params x) {}
8. void Foo(int x, params int[] y) {}
9. void Foo(ref int x) {}

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

С методами разрешено применять следующие модификаторы:

* + - статический модификатор static.
    - модификаторы доступа: public, internal, private, protected.
    - модификаторы наследования: new, virtual, abstract, override, sealed.
    - модификатор частичного метода partial.
    - модификаторы неуправляемого кода: unsafe, extern.
    - модификатор асинхронного кода async.

Тип может перегружать методы (иметь несколько методов с одним и тем же именем) при условии, что их сигнатуры будут отличаться. Возвращаемый тип и модификатор params не входят в состав сигнатуры метода.

Способ передачи параметра - по значению или по ссылке - также является частью сигнатуры. Например, Foo (int) может сосуществовать вместе с Foo ( ref int) или Foo (out int). Однако Foo (ref int) и Foo (out int) сосуществовать не могут.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

class MyClass

{

private int x;

public MyClass(int x)

{

this.x = x;

}

private void ShowX()

{

Console.WriteLine(x);

}

}

static void Main(string[] args)

{

MyClass myClass = new MyClass();

myClass.Show();

}

}

Что будет выведено на экран?

* 0,
* пустая строка,
* возникнет синтаксическая ошибка.

1. Возникнет ли синтаксическая ошибка при выполнении следующего консольного приложения?

class Program

{

class MyClass { }

static void Main(string[] args)

{

MyClass myClass = new MyClass();

}

}

Конструкторы выполняют код инициализации класса или структуры. Класс или структура может перегружать конструкторы. Во избежание дублирования кода один конструктор может вызывать другой конструктор, используя ключевое слово this.

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

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

class MyClass

{

private int x = 13;

public MyClass(int y)

{

if (y > 100)

{

x = y;

}

}

}

static void Main(string[] args)

{

MyClass myClass = new MyClass(50);

Console.WriteLine(myClass.x);

}

}

Что будет выведено на экран?

* 13,
* 50,
* возникнет синтаксическая ошибка.

Инициализация полей происходит *перед* выполнением конструктора в порядке их объявления.

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

Для упрощения инициализации объекта любые его доступные поля и свойства могут быть установлены с помощью ***инициализатора объекта*** непосредственно после создания, например, Point point = new Point { x = 5, y = 7 } – такая инициализация использует конструктор без параметров.

Вместо применения инициализаторов объектов мы могли бы обеспечить прием конструктором класса Point необязательных параметров: public Point(int x = 5, int y = 7).

Тонкая проблема заключается в том, что когда мы изменяем значение одного из необязательных параметров, то вызывающий код в других сборках продолжит использовать старое необязательное значение до тех пор, пока эти сборки не будут перекомпилированы. Таким образом, если необходимо поддерживать двоичную совместимость между версиями сборок, то с необязательными параметрами в открытых функциях следует проявлять осторожность.

Ссылка this указывает на сам экземпляр. Ссылка this допустима только внутри нестатических членов класса или структуры.

1. Чем является X в следующем коде:

MyClass myClass = new MyClass();

myClass.X = 13;

* полем,
* свойством,
* может быть и полем, и свойством.

Снаружи свойства выглядят похожими на поля, но внутренне они содержат логику подобно методам. 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 выбрать любое необходимое внутреннее представление, не демонстрируя внутренние детали пользователю свойства.

1. Как можно упростить следующий код:

class MyClass

{

private int[] array;

public int Length

{

get

{

return array. Length;

}

}

}

Начиная с версии С# 6, свойство, допускающее только чтение, можно объявлять более кратко как свойство, сжатое до выражения (expression-bodied property). Все фигурные скобки, а также ключевые слова get и return заменяются комбинацией =>.

public int Length => array.Length;

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

Средства доступа к свойствам С# внутренне компилируются в методы с именами get\_XXXи set\_XXX.

Простые невиртуальные средства доступа к свойствам встраиваются компилятором JIT, устраняя любую разницу в производительности между доступом к свойству и доступом к полю. Встраивание - это разновидность оптимизации, при которой вызов метода заменяется телом этого метода.

1. Что будет выведено на консоль в результате работы следующего кода:

string s = null;

Console.WriteLine(s?[0]);

* ничего,
* возникнет синтаксическая ошибка.

Индексаторы подобны свойствам, но предусматривают доступ через аргумент индекса, а не имя свойства.

Синтаксис использования индексаторов похож на синтаксис работы с массивами за исключением того, что аргумент (аргументы) индекса может быть любого типа (типов).

Индексаторы имеют те же модификаторы, что и свойства , и могут вызываться null-условным образом за счет помещения вопросительного знака перед открывающей квадратной скобкой.

Индексаторы внутренне компилируются в методы с именами get\_Item и set\_Item.

Сделать пример на создание индексатора с типом, отличным от int.

1. Какие из следующих объявлений констант ошибочны:
2. const int i = 15;
3. const bool b = 0;
4. const char c = 'z';
5. const int[] array = {1, 2, 3};
6. const string s = "Hello";
7. const string[] weekend = {"Saturday", "Sunday"};
8. enum Weekend { Saturday, Sunday };

const Weekend sunday = Wekkend.Sunday;

1. struct Point { public int X, Y; }

const Point origin = new Point();

1. const DateTime today = DateTime.Now;

Константа - это статическое поле, значение которого никогда не может изменяться. Константа оценивается статически на этапе компиляции и компилятор литеральным образом подставляет ее значение всегда, когда она используется. Константа может относиться к любому из встроенных числовых типов, bool, char, string или перечислению.

1. Сколько статических конструкторов может быть у типа:

* 1,
* 0,
* много.

1. Могут ли быть параметры у статического конструктора?

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

Исполняющая среда автоматически вызывает статический конструктор прямо перед тем, как тип начинает применяться. Этот вызов инициируется двумя действиями:

• создание экземпляра типа;

• доступ к статическому члену типа.

Для статических конструкторов разрешены только модификаторы unsafe и extern.

1. Чему равны значения следующих статических полей :

class MyClass

{

public static int X = Y;

public static int Y = 13;

}

* X = 0, Y = 13;
* X = 13, Y = 13;
* возникнет синтаксическая ошибка.

Инициализаторы статических полей запускаются непосредственно перед вызовом статического конструктора. Если тип не имеет статического конструктора, то инициализаторы полей будут выполняться перед тем, как тип начнет использоваться - или в любой момент раньше по прихоти исполняющей среды. (Это означает, что присутствие статического конструктора может привести к тому, что инициализаторы полей выполнятся в программе позже, чем было бы в противном случае.)

Инициализаторы статических полей выполняются в порядке объявления полей.

1. Можно ли создать подкласс на основе статического класса?

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

Частичные типы позволяют расщеплять определение типа - обычно на несколько файлов. Распространенный сценарий предполагает автоматическую генерацию частичного класса из какого-то источника (например, шаблона Visual Studio) и последующее его дополнение вручную написанными методами. Каждый участник должен иметь объявление partial.

Частичный тип может содержать частичные методы. Частичный метод состоит из двух частей: определение и реализация. Определение обычно записывается генератором кода, а реализация - вручную. Если реализация не предоставлена, то определение частичного метода при компиляции удаляется (вместе с кодом, который его вызывает). Это дает автоматически сгенерированному коду большую свободу в предоставлении точек привязки, не заставляя беспокоиться по поводу эффекта раздувания кода. Частичные методы должны быть void, и они неявно являются private.

1. Есть простое консольное приложение, состоящее из следующего кода:

class Program

{

public static int X = 13;

static void Main(string[] args)

{

Console.WriteLine(nameof(X) + X);

}

}

Что будет выведено на экран?

* X13,
* 13,
* возникнет синтаксическая ошибка.

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

Преимущество использования этой операции по сравнению с простым указанием строки связано со статической проверкой типов. Инструменты, подобные Visual Studio, способны воспринимать символические ссылки, поэтому переименование любого символа приводит к переименованию также и всех ссылок на него. Для указания имени члена типа, такого как поле или свойство, необходимо включать тип члена. Это применимо к статическим членам и членам экземпляра.

1. Объявлены два класса:

class Person

{

…

}

class Musician : Person

{

…

}

Какие из следующих переменных объявлены некорректно?

1. Person person = new Person();
2. Musician musician = new Musician();
3. Person person = new Musician();
4. Musician musician = new Person();

Класс может быть унаследован от другого класса с целью расширения или настройки исходного класса. Наследование от класса позволяет повторно использовать функциональность этого класса вместо ее построения с нуля. Класс может наследоваться только от одного класса, но сам может быть унаследован множеством классов, формируя тем самым иерархию классов.

Ссылки являются полиморфными. Это значит, что переменная типа х может ссылаться на объект, относящийся к подклассу х. В основе работы полиморфизма лежит тот факт, что подклассы обладают всеми характеристиками своего базового класса. Однако обратное утверждение не верно.

1. Объявлены два класса:

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }}

class Musician : Person

{

public string Band { get; set; }

}

Чему равно значение переменной equal?

Musician musician = new Musician;

Person person = musician;

bool equal = musician == person;

* true,
* false.

1. Объявлены два класса:

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }}

class Musician : Person

{

public string Band { get; set; }

}

Что будет выведено на консоль в результате работы следующего кода (как зовут Леннона)?

Musician lennon = new Musician() {

FirstName = "John",

LastName = "Lennon",

Band = "Beatles" };

Person person = lennon;

lennon.FirstName = "Paul";

Console.WriteLine(person.FirstName);

* Paul,
* John,
* возникнет синтаксическая ошибка.

Ссылка на объект может быть:

• неявно приведена вверх к ссылке на базовый класс;

• явно приведена вниз к ссылке на подкласс.

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

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

После приведения вверх переменная по-прежнему ссылается на тот же самый объект. Сам объект, на который имеются ссылки, не изменяется и не преобразуется.

1. Объявлены два класса:

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }}

class Musician : Person

{

public string Band { get; set; }

}

Что будет выведено на консоль в результате работы следующего кода?

Person lennon = new Person() {

FirstName = "John",

LastName = "Lennon"

};

Musician musician = (Musician)lennon;

Console.WriteLine(lennon.FirstName);

* John,
* пустая строка,
* возникнет синтаксическая ошибка.

Операция приведения вниз создает ссылку на подкласс из ссылки на базовый класс. Как и в случае приведения вверх, затрагиваются только ссылки, но не лежащий в основе объект. Приведение вниз требует явного указания, потому что потенциально оно может не достигнуть успеха во время выполнения. Когда приведение вниз терпит неудачу, генерируется исключение InvalidCastException.

1. Объявлены два класса:

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }}

class Musician : Person

{

public string Band { get; set; }

}

Чему равно значение musician?

Person lennon = new Person() {

FirstName = "John",

LastName = "Lennon"

};

Musician musician = lennon as Musician;

* null,
* John Lennon,
* возникнет синтаксическая ошибка.

Операция as выполняет приведение вниз, которое в случае отказа вычисляется как null (вместо генерации исключения).

Посредством операции приведения вы сообщаете компилятору о том, что уверены в типе заданного значения; если это не так, значит, в коде присутствует ошибка, поэтому нужно сгенерировать исключение. С другой стороны, в случае операции as вы не уверены в типе значения и хотите организовать ветвление в соответствие с результатом во время выполнения.

1. Объявлены два класса:

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

class Musician : Person

{

public string Band { get; set; }

}

Чему равно значение musician?

Person lennon = new Person() {

FirstName = "John",

LastName = "Lennon"

};

var musician = lennon is Musician;

* null,
* true,
* false,
* John Lennon,
* возникнет синтаксическая ошибка.

Операция is проверяет, будет ли преобразование ссылки успешным; другими словами, является ли объект производным от указанного класса (или реализует ли он какой-то интерфейс). Она часто используется при проверке перед приведением вниз.

Операция is также дает в результате true, если может успешно выполниться распаковывающее преобразование. Однако она не принимает во внимание специальные или числовые преобразования.

1. Объявлены два класса:

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public virtual int Experience { get; set; }

}

class Musician : Person

{

public string Band { get; set; }

}

Что будет выведено на консоль?

Musician lennon = new Musician() {

FirstName = "John",

LastName = "Lennon",

Band = "Beatles"

};

Console.WriteLine(lennon.Experience);

* 0,
* пустая строка,
* возникнет синтаксическая ошибка.

1. Объявлены два класса:

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public virtual int Experience { get; set; }

}

class Musician : Person

{

private int experience;

public string Band { get; set; }

public override int Experience => experience;

public Musician(int experience)

{

this.experience = experience;

}

}

Что будет выведено на консоль?

Person lennon = new Musician() {

FirstName = "John",

LastName = "Lennon",

Band = "Beatles"

};

lennon.Experience = 25;

Console.WriteLine(lennon.Experience);

* 21,
* 25,
* возникнет синтаксическая ошибка.

Функция, помеченная как виртуальная (virtual), может быть переопределена в подклассах, где требуется предоставление ее специализированной реализации. Объявлять виртуальными можно методы, свойства, индексаторы и события.

Виртуальный метод переопределяется в подклассе с применением модификатора override.

1. Объявлены два класса:

abstract class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public abstract int Experience { get; }

}

class Musician : Person

{

private int experience;

public string Band { get; set; }

public override int Experience => experience;

public Musician(int experience)

{

this.experience = experience;

}

}

Что будет выведено на консоль?

Person lennon = new Musician() {

FirstName = "John",

LastName = "Lennon",

Band = "Beatles",

Experience = 21

};

Person mcCartney = new Person() {

FirstName = "Paul",

LastName = "McCartney",

Band = "Beatles"

};

Console.WriteLine

(lennon.FirstName + " and " + mcCartney.FirstName);

* John and Paul,
* John,
* возникнет синтаксическая ошибка.

Класс, объявленный как абстрактный (abstract), не разрешает создавать свои экземпляры. Вместо этого можно создавать только экземпляры его конкретных подклассов.

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

1. Объявлены два класса с одинаковым свойством Name.

class Person

{

public virtual string Name { get; set; }

}

class Musician : Person

{

public new string Name { get; set; }

}

Что будет выведено на консоль?

Musician lennon = new Musician();

lennon.Name = "John Lennon";

Person person = lennon;

Console.WriteLine(lennon.Name);

Console.WriteLine(person.Name);

* John Lennon, пустая строка;
* John Lennon, John Lennon;
* возникнет синтаксическая ошибка.

В базовом классе и подклассе могут быть определены идентичные члены.

Говорят, что поле в подклассе скрывает поле в базовом классе. Обычно это происходит случайно, когда член добавляется к базовому типу после того, как идентичный член был добавлен к подтипу. В таком случае компилятор генерирует предупреждение.

Иногда необходимо скрыть какой-то член преднамеренно; тогда к члену в подклассе можно применить ключевое слово new. Модификатор new не делает ничего сверх того, что просто подавляет выдачу компилятором соответствующего предупреждения. Модификатор new сообщает компилятору - и другим программистам - о том, что дублирование члена произошло не случайно.

С помощью ключевого слова sealed переопределенная функция может запечатывать свою реализацию, предотвращая ее переопределение другими подклассами. Можно также запечатать весь класс, неявно запечатав все его виртуальные функции, путем применения модификатора sealed к самому классу. Запечатывание классов встречается чаще, чем запечатывание отдельных функций-членов.

Хотя можно запечатывать, предотвращая переопределение, нет возможности запечатывать с целью предотвращения сокрытия.

1. Объявлены два класса:

class Person

{

public string Name { get; set; }

public Person()

{}

public Person(string name)

{

Name = name;

}

}

class Musician : Person

{}

Что будет выведено на консоль?

Musician lennon = new Musician("John Lennon");

Console.WriteLine(lennon.Name);

* John Lennon,
* пустая строка,
* возникнет синтаксическая ошибка.

1. Объявлены два класса:

class Person

{

public string Name { get; set; }

public Person()

{

Name = "John Doe";

}

}

class Musician : Person

{}

Что будет выведено на консоль?

Musician musician = new Musician();

Console.WriteLine(musician.Name);

* John Doe,
* пустая строка,
* возникнет синтаксическая ошибка.

В подклассе должны быть объявлены собственные конструкторы. Конструкторы базового класса доступны в производном классе, но они никогда автоматически не наследуются.

Следовательно, в подклассе должны быть "повторно определены" любые конструкторы, которые необходимо открыть. Однако при этом можно вызывать любой конструктор базового класса с применением ключевого слова base.

Ключевое слово base похоже на ключевое слово this. Оно служит двум важным целям:

• доступ к функции-члену базового класса при ее переопределении в подклассе;

• вызов конструктора базового класса.

Конструкторы базового класса всегда выполняются первыми; это гарантирует выполнение базовой инициализации перед специализированной инициализацией.

Если в конструкторе подкласса опустить ключевое слово base, будет неявно вызываться конструктор без параметров базового класса. Если базовый класс не имеет доступного конструктора без параметров, то в конструкторах подклассов придется использовать ключевое слово base.

class A

{

public int X = 13;

public A(int x) {...}

}

class Musician : Person

{}

Когда объект создан, инициализация происходит в указанном ниже порядке.

1. От подкласса к базовому классу:

а) инициализируются поля;

б) оцениваются аргументы для вызова конструкторов базового класса.

2. От базового класса к подклассу:

а) выполняются тела конструкторов.

**----- стр.122**

## Глава 17. Сериализация

1. Какие техники исполнения сериализации доступны в .NET Framework:

* сериализатор контрактов данных (data contract serializer),
* двоичный сериализатор (binary serializer),
* сериализатор ХМL (Xml Serializer),
* интерфейс IXmlSerializable,
* интерфейс ISerializable,
* двоичный форматер (binary formatter).

Сериализация и десериализация - механизмы, с помощью которых объекты могут быть представлены в простой текстовой или двоичной форме. Все типы находятся в следующих пространствах:

System.Runtime.Serialization

System.Xml.Serialization

Сериализация - это действие по превращению находящегося в памяти объекта или графа объектов (набора объектов, ссылающихся друг на друга) в плоское представление в виде потока байтов или ХМL-узлов, который может быть сохранен или передан по сети. Десериализация работает в противоположном направлении, получая поток данных и восстанавливая его в объект или граф объектов в памяти. Сериализация и десериализация обычно используются для решения следующих задач:

• передача объектов по сети или за границы приложения;

• сохранение представлений объектов внутри файла или базы данных.

Еще одно менее распространенное применение связано с глубоким копированием объектов. Контракты данных и механизмы сериализации ХМL могут также использоваться в качестве универсальных инструментов для загрузки и сохранения XML-файлов с известной структурой.

В .NET Framework доступны четыре техники исполнения сериализации:

• сериализатор контрактов данных;

• двоичный сериализатор (в настольных приложениях);

• сериализатор (на основе атрибутов) ХМL (XmlSerializer);

• интерфейс IXmlSerializable.

Первые три из них - это "механизмы" сериализации, которые делают большую часть или всю работу самостоятельно. Последняя техника представляет собой просто прием выполнения сериализации с применением классов XmlReader и XmlWriter.

Двоичный сериализатор хорошо автоматизирован: довольно часто единственный атрибут - это все, что требуется для того, чтобы сделать сложный тип полностью сериализируемым. Тем не менее, он тесно связывает внутреннюю структуру типа с форматом сериализированных данных, приводя в результате к плохой переносимости версий.

Вывод сериализатора контрактов данных и двоичного сериализатора оформляется с помощью подключаемого форматера. Роль форматера одинакова в обоих механизмах сериализации, хотя для выполнения работы они используют совершенно разные классы. Форматер приводит форму финального представления в соответствие с конкретной средой или контекстом сериализации.

Механизм сериализации ХМL может генерировать только ХМL, и по сравнению с другими механизмами менее функционален при сохранении и восстановлении сложного графа объектов (он не позволяет восстанавливать разделяемые объектные ссылки). Однако он обладает наибольшей гибкостью из трех механизмов в следовании произвольной структуре ХМL. Например, можно выбирать, во что должны сериализироваться свойства - в элементы или в атрибуты, и обрабатывать внешний ХМL элемент коллекции. Механизм сериализации ХМL также поддерживает великолепную переносимость версий.

Реализация интерфейса IXmlSerializable означает самостоятельное выполнение сериализации с помощью классов XmlReader и XmlWriter.

1. Какие существуют способы сериализации и десериализации:

* явная,
* неявная,
* оба варианта верны.

Сериализация и десериализация могут быть инициированы двумя способами. Первый способ предусматривает явное запрашивание сериализации и десериализации конкретного объекта. При явной сериализации или десериализации выбирается механизм и форматер. В отличие от явной неявная сериализация запускается платформой .NET Framework. Это происходит, когда сериализатор рекурсивно сериализирует дочерний объект.

1. Какими из нижеперечисленных атрибутов можно пометить метод для того, чтобы он выполнятся до/после сериализации/десериализации:

* [OnSerializing],
* [OnSerialized],
* [OnDeserializing],
* [OnDeserialized],
* [OnSerializable],
* [OnSerialize].

Можно затребовать, чтобы до или после сериализации выполнялся специальный метод, пометив его одним из перечисленных ниже атрибутов.

[OnSerializing] – указывает метод для вызова перед сериализацией.

[OnSerialized] – указывает метод для вызова после сериализации.

Аналогичные атрибуты поддерживаются и для десериализации.

[OnDeserializing] – указывает метод для вызова перед десериализацией.

[OnDeserialized] – указывает метод для вызова после десериализации.

Специальный метод должен иметь единственный параметр типа StreamingContext. Этот параметр обязателен для обеспечения согласованности с механизмом двоичной сериализации; сериализатор контрактов данных его не использует.

Атрибуты [OnSerializing] и [OnDeserialized] пригодны для обработки членов, которые выходят за рамки возможностей механизма сериализации контрактов данных, таких как коллекция, несущая дополнительную полезную нагрузку или не реализующая стандартные интерфейсы.

1. Каким способом можно сделать тип поддерживающим двоичную сериализацию:

* пометить тип атрибутом [Serializable],
* реализовать интерфейс ISerialiable,
* оба варианта верны.

Сделать тип поддерживающим двоичную сериализацию можно двумя путями. Первый из них основан на атрибутах, а второй предусматривает реализацию интерфейса ISerializable. Добавление атрибутов проще, но реализация ISerializable предлагает более высокую гибкость.

Тип делается сериализируемым с помощью единственного атрибута: [Serializable].

Атрибут [Serializable] инструктирует сериализатор о необходимости включения всех полей в данном типе. Это касается как закрытых, так и открытых полей (но не свойств). Каждое поле само должно допускать сериализацию, иначе сгенерируется исключение. Примитивные типы .NET, такие как string и int, поддерживают сериализацию (подобно многим другим типам .NET).

Атрибут Serializable не наследуется, так что подклассы не будут автоматически сериализируемыми, если их не пометить этим атрибутом.

Для применения с механизмом двоичной сериализации предназначены два форматера: BinaryFormatter и SoapFormatter.

Указанные два форматера применяются одинаково. Следующий код выполняет сериализацию объекта Person с помощью BinaryForrnatter:

Person р = new Person ()

{ Name = "George", Age = 25 };

IFormatter formatter = new BinaryFormatter();

using (FileStream s = File.Create ("serialized.bin"))

formatter.Serialize (s, р);

Все данные, необходимые для реконструирования объекта Person, записываются в файл serialized.bin. Метод Deserialize восстанавливает объект:

using (FileStream s = File.OpenRead ("serialized.bin"))

{

Person р2 = (Person) formatter.Deserialize (s);

Console.WriteLine (p2.Name + " " + p.Age);

}

Сериализированные данные включают полные сведения о типе и сборке, поэтому если попытаться привести результат десериализации к совпадающему типу Person из другой сборки, то возникнет ошибка. Десериализатор восстанавливает объектные ссылки полностью в их исходном состоянии. Это касается коллекций, которые трактуются просто как сериализируемые объекты, подобные любым другим (все коллекции, определенные в пространствах имен System.Collections.\*, помечены как сериализируемые).

При сериализации полей, механизм двоичной сериализации реализует политику отключения (opt-out). Поля, которые не должны сериализироваться, такие как используемые для временных вычислений или для хранения файловых либо оконных дескрипторов, потребуется явно пометить с помощью атрибута [NonSerialized].

1. Для реализации интерфейса ISerializable необходимо реализовать метод void GetObjectData (SerializationInfo info, StreamingContext context). Что представляет собой параметр info:

* словарь пар "имя/значение",
* список полей для сериализации,
* хранит параметры сериализуемого типа (номер сборки и т.п.).

Реализация интерфейса ISerializable предоставляет типу полный контроль над прохождением его двоичной сериализации и десериализации.

Ниже показано определение интерфейса ISerializable:

public interface ISerializable

{

void GetObjectData (SerializationInfo info, StreamingContext context);

}

Метод GetObjectData запускается при сериализации; его работа заключается в наполнении объекта SerializationInfo (словарь пар "имя/значение") данными из всех полей, которые необходимо сериализировать. Вот как можно реализовать метод GetObjectData, сериализирующий два поля с именами Name и DateOfBirth:

public virtual void GetObjectData (Serializationinfo info, StreamingContext context)

{

info.AddValue ("Name", Name);

info.AddValue ("DateOfBirth", DateOfBirth);

}

В приведенном примере мы решили именовать каждый элемент согласно соответствующему ему полю. Это вовсе не обязательно; может применяться любое имя при условии, что при десериализации используется такое же точно имя. Сами значения могут иметь любой сериализируемый тип; при необходимости платформа .NET Framework будет выполнять рекурсивную сериализацию. В словаре разрешено хранить значения null.

Неплохо объявить метод GetObjectData как virtual - если только класс не является sealed. Это позволит подклассам расширять сериализацию без необходимости в повторной реализации интерфейса ISerializable.

1. Каким способом можно организовать механизм XML сериализации:

* использовать атрибуты, определенные в System.Xml.Serialization,
* реализовать интерфейс IXmlSerialiable,
* оба варианта верны.

Платформа .NET Framework предлагает в пространстве имен System.Xml.Serialization отдельный под названием XmlSerializer. Он подходит для сериализации типов .NET в ХМL·файлы и также неявно применяется веб-службами ASMX.

Как и в случае механизма двоичной сериализации, на выбор доступны два подхода:

• добавить к типам атрибуты (определенные в System.Xml.Serialization);

• реализовать интерфейс IXmlSerializable.

Однако в отличие от механизма двоичной сериализации реализация интерфейса (т.е. IXmlSerializable) полностью избегает использования механизма, оставляя на разработчика самостоятельное написание кода сериализации с участием классов XmlReader и XmlWriter.

1. Объявлен класс Person:

public class Person

{

public string Name {get; set;}

public int Age {get; set;}

}

Что нужно сделать для того, чтобы этот класс и его свойства были доступны для XML сериализации:

* пометить класс атрибутом [Serializable],
* пометить атрибутом [Serializable] свойства класса,
* ничего не надо делать.

Класс XmlSerializer способен сериализировать типы, не имеющие ни одного атрибута. По умолчанию он сериализирует все открытые поля и свойства типа.

Десериализатор является переносимым в плане версий: он не жалуется, если элементы или атрибуты отсутствуют либо встречаются лишние данные.

1. Объявлен класс Person:

public class Person

{

public string Name {get; set;}

public int Age {get; set;}

}

Что нужно сделать для того, чтобы при использовании XML сериализации получился следующий результат:

<Person CurrentAge="30">

<Name>John</Name>

</Person>

* пометить свойство Age атрибутом [XmlAttribute("CurrentAge")],
* изменить имя свойства Age на CurrentAge,
* такого результата добиться нельзя.

По умолчанию поля и свойства сериализируются в ХМL-элементы. Потребовать, чтобы взамен применялся ХМL-атрибут, нужно применить атрибут [XmlAttribute].

1. Объявлен класс Person, который содержит в себе ссылку на класс Address:

public class Person

{

public string Name {get; set;}

public int Age {get; set;}

public Address HomeAddress {get; set;}

}

Что произойдет в результате XML сериализации:

* сериализация класса Person пройдет успешно, так как все вложенные классы сериализуются автоматически,
* для успешной сериализации необходимо отдельно сериализовать класс Address,
* возникнет синтаксическая ошибка.

Класс XmlSerializer автоматически рекурсивно обрабатывает объектные ссылки (то есть сериализует дочерние объекты).

1. Объявлен класс Person, который содержит список объектов Phone:

public class Person

{

public string Name {get; set;}

public int Age {get; set;}

public List<Phone> Phones {get; set;}

}

Что произойдет в результате XML сериализации:

* сериализация класса Person пройдет успешно, так как типы коллекций сериализуются автоматически,
* возникнет синтаксическая ошибка.

Класс XmlSerializer распознает и сериализирует конкретные типы коллекций, не требуя какого-то вмешательства.

1. В каких из перечисленных ситуаций может потребоваться реализовать интерфейс IXmlSerializable:

* для сериализации неоткрытых членов,
* для добавления ловушек сериализации,
* для сериализации дочерних объектов,
* для сериализации коллекций.

Хотя сериализация ХМL на основе атрибутов обладает гибкостью, с ней связаны ограничения. Например, невозможно добавлять ловушки сериализации, равно как и сериализировать неоткрытые члены. Ее также неудобно использовать, если ХМL может представлять один и тот же элемент или атрибут несколькими разными путями.

Что касается последней проблемы, то границы можно несколько раздвинуть, передавая объект XmlAttributeOverrides конструктору XmlSerializer. Тем не менее, наступает момент, когда проще принять императивный подход. За эту работу отвечает интерфейс IXmlSerializable. Реализация интерфейса IXmlSerializable обеспечивает полный контроль над читаемым или записываемым ХМL.