Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ЯЗЫК C# И ОСНОВЫ ПЛАТФОРМЫ .NET

Курс лекций
для студентов специальности I-31 03 04 Информатика
всех форм обучения

Минск 2011

СОДЕРЖАНИЕ

1. ЯЗЫК C# И ОСНОВЫ ПЛАТФОРМЫ .NET 4

1.1. ОБЗОР ПЛАТФОРМЫ .NET 4

1.2. ОБЩИЕ КОНЦЕПЦИИ СИНТАКСИСА C# 5

1.3. СИСТЕМА ТИПОВ CLR И ЯЗЫКА C# 7

1.4. ИДЕНТИФИКАТОРЫ, КЛЮЧЕВЫЕ СЛОВА И ЛИТЕРАЛЫ 9

1.5. ВЫРАЖЕНИЯ И ОПЕРАЦИИ 11

1.6. ОПЕРАТОРЫ 13

Операторы объявления 13

Операторы выражений 14

Операторы перехода 15

Операторы выбора 15

Операторы циклов 16

Прочие операторы 17

1.7. НАЧАЛЬНЫЕ СВЕДЕНИЯ О МАССИВАХ 18

1.8. КЛАССЫ 20

Допустимые элементы класса 20

Модификаторы доступа для элементов и типов 21

Разделяемые классы 22

Использование класса 22

1.9. МЕТОДЫ 23

Описание метода 23

Вызов метода 26

Разделяемые методы 27

1.10. СВОЙСТВА И ИНДЕКСАТОРЫ 27

1.11. СТАТИЧЕСКИЕ ЭЛЕМЕНТЫ И МЕТОДЫ РАСШИРЕНИЯ 31

Статические элементы 31

Статические классы 31

Методы расширения 32

1.12. КОНСТРУКТОРЫ И ИНИЦИАЛИЗАЦИЯ ОБЪЕКТОВ 33

1.13. НАСЛЕДОВАНИЕ КЛАССОВ **35**

1.14. КЛАСС SYSTEM.OBJECT И ИЕРАРХИЯ ТИПОВ **39**

1.15. СТРУКТУРЫ **41**

1.16. ПЕРЕЧИСЛЕНИЯ **42**

1.17. ИНТЕРФЕЙСЫ **44**

1.18. УНИВЕРСАЛЬНЫЕ ШАБЛОНЫ **46**

Универсальные классы и структуры **46**

Ограничения на параметры шаблонов **49**

Ковариантность и контравариантность **50**

Универсальные методы **51**

1.19. ИСПОЛЬЗОВАНИЕ УНИВЕРСАЛЬНЫХ ШАБЛОНОВ **52**

Кортежи **52**

Типы, допускающие значение null **53**

Прочие примеры универсальных шаблонов **54**

1.20. ДЕЛЕГАТЫ **55**

1.21. АНОНИМНЫЕ МЕТОДЫ И ЛЯМБДА-ВЫРАЖЕНИЯ **58**

1.22. СОБЫТИЯ **60**

1.23. ПЕРЕГРУЗКА ОПЕРАЦИЙ **64**

1.24. АНОНИМНЫЕ ТИПЫ **67**

1.25. ПРОСТРАНСТВА ИМЁН **68**

1.26. ГЕНЕРАЦИЯ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ **69**

1.27. ПРЕПРОЦЕССОРНЫЕ ДИРЕКТИВЫ **72**

1.28. ДОКУМЕНТИРОВАНИЕ ИСХОДНОГО КОДА **73**

1. ЯЗЫК C# И ОСНОВЫ ПЛАТФОРМЫ .NET

1.1. ОБЗОР ПЛАТФОРМЫ .NET

В середине 2000 года корпорация Microsoft объявила о работе над новой платформой для создания приложений, которая получила имя *платформа .NET* (*.NET Framework*). Платформа .NET образует каркас, включающий библиотеку классов, технологии для доступа к данным и построения оконных и веб-приложений. Основным инструментом разработки для платформы .NET является интегрированная среда Microsoft Visual Studio.

База платформы .NET - это *общеязыковая среда исполнения* (*Common Language Runtime*, *CLR*). CLR является «прослойкой» между операционной системой и приложением. Приложения для платформы .NET состоят из *управляемого кода* (*managed code*), который является результатом компиляции исходных текстов. Скомпилированные файлы называются *сборками* (*assembly*) и включают следующие части:

1. *Манифест* (*manifest*) – описание сборки: версия, ограничения безопасности, список необходимых внешних сборок.
2. *Метаданные* – специальное описание всех пользовательских типов, размещённых в сборке.
3. *Код на языке Microsoft Intermediate Language* (*MSIL*, или просто *IL*). Данный код является независимым от операционной системы и типа процессора. В процессе работы приложения он компилируется в машинно-зависимый код специальным JIT-компилятором (*Just-in-Time compiler*).

Основная задача CLR – это манипулирование сборками: загрузка, JIT-компиляция, создание окружения для выполнения сборок. Важной функцией CLR является управление памятью при работе приложения и выполнение *автоматической сборки мусора*, то есть фонового освобождения неиспользуемой памяти. Кроме этого, CLR реализует в приложениях для .NET проверку типов, управление политиками безопасности при доступе к коду и другие функции.

В состав платформы .NET входит обширная библиотека классов *Framework Class Library* (*FCL*). Элементом этой библиотеки является базовый набор классов *Base Class Library* (*BCL*). В BCL входят классы для работы со строками и коллекциями данных, для поддержки многопоточности и множество других классов. Частью FCL являются компоненты, поддерживающие различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML и базами данных, для создания пользовательских интерфейсов.

В стандартную поставку платформы .NET включено несколько компиляторов. Это компиляторы языков C#, F#, Visual Basic .NET, C++/CLI. Благодаря открытым спецификациям компиляторы для .NET предлагаются различными сторонними производителями. Необходимо подчеркнуть, что любой язык для платформы .NET является верхним элементом архитектуры. Имена элементов

библиотеки FCL не зависят от языка программирования. Специфичной частью языка остается только синтаксис. Этот факт упрощает межъязыковое взаимодействие, перевод текста программы с одного языка на другой. Конечно, в синтаксических элементах любого языка программирования для .NET неизбежно находит свое отражение тесная связь с CLR.

Для поддержки межъязыкового взаимодействия служат две спецификации платформы .NET. *Общая система типов* (*Common Type System*, *CTS*) описывает набор типов, который должен поддерживаться любым языком программирования для .NET. *Общеязыковая спецификация* (*Common Language Specification*, *CLS*) – это общие правила поведения для всех .NET-языков.

В заключение рассмотрим историю версий платформы .NET.

- Февраль 2002 года - первая официальная версия платформы .NET.
- Апрель 2003 года - опубликована версия 1.1 (по сути, это пакет обновлений для версии 1.0).
- Ноябрь 2005 года – вышла версия 2.0, содержащая обновленную CLR с поддержкой универсальных шаблонов (generics). В синтаксис языков C# и VB.NET были внесены существенные изменения. Переработаны и улучшены технологии ASP.NET и ADO.NET.
- Ноябрь 2006 года – выпуск версии 3.0, которая содержит набор технологий Windows Presentation Foundation, Windows Communication Foundation, Workflow Foundation.
- Ноябрь 2007 года - вышла версия 3.5, основными особенностями которой являются реализация технологии LINQ и новые версии компиляторов для C# и VB.NET.
- Август 2008 года - опубликован пакет обновлений для версии 3.5.
- Апрель 2010 года - выпущена четвёртая версия платформы .NET, которая содержит переработанную CLR, а также интегрирует множество новых технологий, существовавших ранее в виде отдельных проектов (например, Parallel Task Library, DLR, ASP.NET MVC).

1.2. ОБЩИЕ КОНЦЕПЦИИ СИНТАКСИСА C#

Специально для платформы .NET был разработан новый язык программирования C#. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и Java, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного и простого в сопровождении кода. Здесь и далее рассматривается синтаксис четвёртой версии языка C#, доступной в составе .NET Framework 4.

Ключевыми структурными понятиями в языке C# являются *программы*, *сборки*, *пространства имен*, *пользовательские типы* и *элементы типов*. Исходный код программы на языке C# размещается в одном или нескольких текстовых файлах, имеющих стандартное расширение `.cs`. В программе объявляются пользовательские типы, которые состоят из элементов. Примерами поль-

зовательских типов являются классы и структуры, а примером элемента типа - метод класса. Типы могут быть логически сгруппированы в пространства имен, а физически (после компиляции) – в сборки, представляющие собой файлы с расширением `.exe` или `.dll`.

Исходный текст программы на языке C# - это набор *операторов* (*statements*[1]) и *комментариев*. Комментарии игнорируются при компиляции и бывают трёх видов:

1. *Строчный комментарий* – это комментарий, начинающийся с последовательности `//` и продолжающийся до конца строки.
2. *Блочный комментарий* – все символы, заключенные между `/*` и `*/`.
3. *Комментарий для документации* – напоминает строчный комментарий, но начинается с последовательности `///` и содержит специальные теги.

В C# различаются строчные и прописные символы при записи идентификаторов и ключевых слов. Количество пробелов в начале строки, в конце строки и между элементами строки значения не имеет. Это позволяет улучшить структуру исходного текста программы – операторы одного уровня вложенности обычно сопровождаются одинаковым начальным отступом.

Рассмотрим простейшую программу на языке C#, которая переводит расстояние в милях в километры.

```
using System;

class FirstProgram
{
    static void Main()
    {
        Console.Write("Input miles: ");
        string s = Console.ReadLine();
        double miles = double.Parse(s);
        Console.Write("In kilometers: ");
        Console.WriteLine(miles * 1.609);
    }
}
```

Программа представляет собой описание пользовательского типа – класса с именем `FirstProgram`. Необязательная директива `using` в первой строке программы служит для ссылки на пространство имен `System`, группирующее базовый набор классов. Использование `using System` позволяет вместо полного имени класса `System.Console` записать короткое имя `Console`.

Любая исполняемая программа на C# должна иметь специальную *точку входа*, с которой начинается выполнение приложения. Такой точкой входа всегда является метод `Main()` с модификатором `static`, объявленный в некотором пользовательском типе программы (в данном случае – в классе `FirstProgram`).

[1] Термин *statement* переводится в данном пособии как «оператор», а термин *operator* – как «операция».

Метод `Main()` начинается с вызова метода `Write()` класса `Console`. Методы `Console.WriteLine()` и `Console.Write()` выводят информацию на экран, а метод `Console.ReadLine()` ожидает ввод пользователя и возвращает введённые данные как строку. Информация сохраняется в локальной строковой переменной `s`. Метод `double.Parse()` выполняет преобразование строки в вещественный тип.

Если программа содержится в файле `FirstProgram.cs`, то она может быть скомпилирована при помощи *компилятора командной строки* `csc.exe`. При этом допустимо указание различных параметров – имени скомпилированного файла, ссылок на необходимые сборки и так далее.

```
csc.exe FirstProgram.cs
```

После компиляции будет получена сборка `FirstProgram.exe`, готовая для запуска на любом компьютере с установленной платформой .NET.

Скомпилированные программы для платформы .NET допускают *декомпиляцию*, то есть восстановление исходного кода программы. Для этой цели можно использовать такие инструменты как `ILDasm` (от Microsoft) или `Reflector` (автор - Lutz Roeder).

1.3. СИСТЕМА ТИПОВ CLR И ЯЗЫКА C#

Основой CLR является развитая система типов. Все типы C# соответствуют определенным типам CLR. Имя типа в C# - это псевдоним типа из CLR (например, тип `int` в C# - псевдоним типа `System.Int32`).

Система типов допускает несколько вариантов классификации. С точки зрения размещения переменных в памяти все типы можно разделить на *типы значений* и *ссылочные типы*. Переменная типа значения непосредственно содержит данные и размещается в стеке. К типам значений относятся *структуры* и *перечисления*. Структуры, в свою очередь, делятся на *числовые типы*, *тип* `bool` и *пользовательские структуры*. Переменная ссылочного типа, далее называемая *объектом*, содержит ссылку на данные, которые размещены в управляемой динамической памяти. Ссылочные типы – это *класс*, *интерфейс*, *строка*, *массив*, *делегат* и *тип* `object`.

Другой подход к классификации типов предполагает деление на *примитивные типы* и *пользовательские типы*. Числовые типы, а также типы `bool`, `string` и `object` принято относить к примитивным типам, так как они встроены в CLR. Пользовательские типы перед применением должны быть описаны при помощи особых синтаксических конструкций. Любая программа на языке C# представляет собой набор определенных пользовательских типов.

Числовые типы делятся на *целочисленные типы*, *типы с плавающей запятой* и *тип* `decimal`. Информация о числовых типах представлена в табл. 1.

Таблица 1

Числовые типы C# и CLR

Категория	Размер (бит)	Тип C#	Имя типа в CLR	Диапазон/Точность
Целочисленные типы	8	`sbyte`	`System.SByte`	–128..127
	16	`short`	`System.Int16`	–32 768..32 767
	32	`int`	`System.Int32`	–2 147 483 648..2 147 483 647
	64	`long`	`System.Int32`	-9 223 372 036 854 775 808.. 9 223 372 036 854 775 807
	8	`byte`	`System.Byte`	0..255
	16	`ushort`	`System.UInt16`	0..65535
	16	`char`	`System.Char`	Символ в кодировке Unicode
	32	`uint`	`System.UInt32`	0..4 294 967 295
	64	`ulong`	`System.UInt64`	0..18 446 744 073 709 551 615
Типы с плавающей запятой	32	`float`	`System.Single`	Точность: от 1.5×10^{-45} до 3.4×10^{38}, 7 цифр
	64	`double`	`System.Double`	Точность: от 5.0×10^{-324} до 1.7×10^{308}, 14-15 цифр
Тип `decimal`	128	`decimal`	`System.Decimal`	Точность: от 1.0×10^{-28} до 7.9×10^{28}, 28 цифр

Отметим, что типы `sbyte`, `ushort`, `uint`, `ulong` не соответствуют Common Language Specification. Это означает, что данные типы не следует использовать в интерфейсах межъязыкового взаимодействия. Тип `char`, хотя формально и относится к целочисленным, представляет символ в 16-битной Unicode-кодировке. Тип `decimal` удобен для проведения финансовых вычислений.

Так как язык C# - это язык со строгой типизацией, необходимо соблюдать соответствие типов при присваивании и вызове методов. В случае несоответствия выполняется *преобразование типов*, которое бывает явным и неявным. Для *явного преобразования* (*explicit conversion*) служит *операция приведения* в форме `(<целевой тип>)<выражение>`. При этом ответственность за корректность преобразования возлагается на программиста. *Неявное преобразование* (*implicit conversion*) не требует особых синтаксических конструкций и осуществляется компилятором. Подразумевается, что неявное преобразование безопасно, то есть, например, для целочисленных типов не происходит переполнения. Для числовых типов определено неявное преобразование типа `A` в тип `B`, если на схеме 1 существует путь из `A` в `B`.

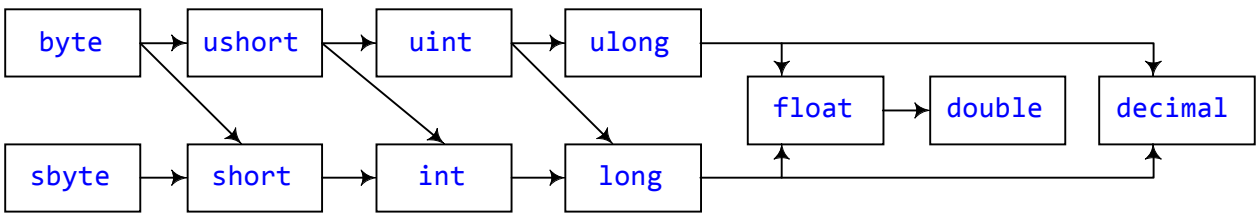

Рис. 1. Схема неявного преобразования числовых типов.

Тип `char` преобразуется в типы `sbyte`, `short`, `byte` явно, а в остальные числовые типы – неявно. Преобразование числового типа в тип `char` может быть выполнено только в явной форме.

Тип `bool` (`System.Boolean`) служит для хранения булевых значений. Переменные данного типа могут принимать значения `true` или `false`. Невозможны никакие преобразования из `bool` в числовые типы и обратно.

Тип `string` (`System.String`) используется для работы со строками и является последовательностью Unicode-символов.

Тип `object` (`System.Object`) - это ссылочный тип, переменной которого можно присвоить любое значение.

Опишем функциональность, которой обладают пользовательские типы.

1. Класс – тип, поддерживающий всю функциональность объектно-ориентированного программирования, включая наследование и полиморфизм.
2. Структура – тип значения, обеспечивающий инкапсуляцию данных, но не поддерживающий наследование. Синтаксически, структура похожа на класс.
3. Интерфейс – абстрактный тип, реализуемый классами и структурами для обеспечения оговоренной функциональности.
4. Массив – пользовательский тип для представления упорядоченного набора значений.
5. Перечисление – тип, содержащий в качестве членов именованные целочисленные константы.
6. Делегат – пользовательский тип, инкапсулирующий метод.

1.4. ИДЕНТИФИКАТОРЫ, КЛЮЧЕВЫЕ СЛОВА И ЛИТЕРАЛЫ

Идентификатор – это пользовательское имя для переменной, константы, метода или типа. В C# идентификатор – это произвольная последовательность букв, цифр и символов подчеркивания, начинающаяся с буквы, символа подчеркивания, либо символа `@`. Идентификатор должен быть уникальным внутри области видимости. Он не может совпадать с ключевым словом языка, за исключением того случая, когда используется специальный *префикс* `@` (не являющийся частью идентификатора). Примеры допустимых идентификаторов: `Temp`, `_variable`, `_`, `@class` (используется префикс `@`, так как `class` – ключевое слово).

Ключевые слова - это предварительно определенные зарезервированные идентификаторы, имеющие специальные значения для компилятора. Их нельзя использовать в программе в качестве идентификаторов. Далее приведены два списка ключевых слов. В первом списке перечислены ключевые слова, являющиеся зарезервированными идентификаторами в любой части программы C#:

```
abstract    do          in          protected   true
as          double      int         public      try
base        else        interface   readonly    typeof
bool        enum        internal    ref         uint
break       event       is          return      ulong
byte        explicit    lock        sbyte       unchecked
case        extern      long        sealed      unsafe
```

```
catch       false       namespace   short       ushort
char        finally     new         sizeof      using
checked     fixed       null        stackalloc  virtual
class       float       object      static      volatile
const       for         operator    string      void
continue    foreach     out         struct      while
decimal     goto        override    switch
default     if          params      this
delegate    implicit    private     throw
```

Во втором списке перечислены *контекстные ключевые слова*. Они имеют особое значение только в ограниченном программном контексте и могут использоваться в качестве идентификаторов за пределами этого контекста, хотя так поступать не рекомендуется:

```
add         global      let         select      where
dynamic     group       orderby     set         yield
from        into        partial     value
get         join        remove      var
```

Литерал – это последовательность символов, которая может интерпретироваться как значение одного из примитивных типов. Так как C# является языком со строгой типизацией, часто необходимо явно указать, к какому типу относится последовательность символов, определяющая данные.

Рассмотрим правила записи некоторых литералов. Для ссылочных типов определен литерал `null`, который указывает на неинициализированную ссылку. В языке C# два булевых литерала: `true` и `false`. Целочисленные литералы могут быть записаны в десятичной или шестнадцатеричной форме. Признаком шестнадцатеричного литерала является префикс `0x`. Конкретный тип целочисленного литерала определяется следующим образом:

- Если литерал не имеет суффикса, то его тип – это первый из типов `int`, `uint`, `long`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `U` или `u`, его тип – это первый из типов `uint`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `L` или `l`, то его тип – это первый из типов `long`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, `lu`, его тип – `ulong`[1].

Если в числе с десятичной точкой не указан суффикс, то подразумевается тип `double`. Суффикс `f` (или `F`) используется для указания на тип `float`, суффикс `d` (или `D`) используется для явного указания на тип `double`, суффикс `m` (или `M`) определяет литерал типа `decimal`. Число с плавающей точкой может быть записано в научном формате: `3.5E-6`, `-7E10`, `.6E+7`.

[1] При записи целочисленных литералов не рекомендуется использовать суффикс `l` (строчная `L`), так как его легко перепутать с единицей.

Символьный литерал обычно записывают как единичный символ в кавычках (`'a'`). Альтернативным способом записи символьного литерала является использование шестнадцатеричного кода Unicode, заключенного в одинарные кавычки (`'\x005C'` или `'\u005C'` – это символ \). Кроме этого, для представления некоторых специальных символов используются следующие пары:

`\'` – одинарная кавычка
`\"` – двойная кавычка
`\\` – обратный слеш
`\0` – символ с кодом `'\x0000'`
`\a` – звуковой сигнал
`\b` – забой
`\f` – новая страница
`\n` – новая строка
`\r` – возврат каретки
`\t` – горизонтальная табуляция
`\v` – вертикальная табуляция

Для строковых литералов в языке C# существуют две формы. Обычно строковый литерал записывается как последовательность символов в двойных кавычках. Среди символов строки могут быть и управляющие последовательности (`"This is \t tabbed string"`). *Дословная форма* (*verbatim form*) строкового литерала – это запись строки в кавычках с использованием префикса `@` (`@"There is \t no tab"`). В этом случае управляющие последовательности воспринимаются как обычные пары символов. Дословная форма может занимать несколько строк.

1.5. ВЫРАЖЕНИЯ И ОПЕРАЦИИ

Любое выражение в языке C# состоит из операндов и операций. Следующий список содержит допустимые операции. Они разбиты на группы. Порядок групп соответствует приоритету операций (операции в каждой группе имеют одинаковый приоритет, за исключением группы логических операций, где они расположены в порядке убывания приоритета).

1. **Основные (первичные) операции**

`x.m`	Доступ к элементу типа
`x(...)`	Вызов методов и делегатов
`x[...]`	Доступ к элементу массива или индексатора
`x++`	Постинкремент
`x--`	Постдекремент
`new T(...)`	Создание объекта или делегата типа `T`
`new T[...]`	Создание массива с элементами типа `T`
`typeof(T)`	Получение для типа `T` объекта `System.Type`
`checked(x)`	Вычисление в контролируемом контексте
`unchecked(x)`	Вычисление в неконтролируемом контексте

2. **Унарные операции**

`+x`	Идентичность
`-x`	Отрицание
`!x`	Логическое отрицание
`~x`	Битовое отрицание

`++x`	Пре-инкремент
`--x`	Пре-декремент
`(T)x`	Явное преобразование `x` к типу `T`
`true(x)`	Возвращает логическое значение `true` для операнда
`false(x)`	Возвращает логическое значение `false` для операнда
`sizeof(T)`	Размер в байтах для примитивного типа значения `T`

3. **Мультипликативные операции**

`x * y`	Умножение
`x / y`	Деление
`x % y`	Вычисление остатка

4. **Аддитивные операции**

`x + y`	Сложение чисел или конкатенация строк
`x - y`	Вычитание

5. **Операции сдвига**

`x << y`	Битовый сдвиг влево
`x >> y`	Битовый сдвиг вправо

6. **Операции отношения и проверки типов**

`x < y`	Меньше
`x > y`	Больше
`x <= y`	Меньше или равно
`x >= y`	Больше или равно
`x is T`	Возвращает `true`, если `x` приводим к типу `T`
`x as T`	Возвращает `x`, приведённый к типу `T`, или `null`

7. **Операции равенства**

`x == y`	Равно
`x != y`	Не равно

8. **Логические и условные операции**

`x & y`	Целочисленное битовое AND, логическое AND
`x ^ y`	Целочисленное битовое XOR, логическое XOR
`x | y`	Целочисленное битовое OR, логическое OR
`x && y`	Вычисляется `y`, только если `x == true`
`x || y`	Вычисляется `y`, только если `x == false`
`x ? y : z`	Если `x == true`, вычисляется `y`, иначе `z`

9. **Операции присваивания**

`x = y`	Присваивание
`x` *`op`*`= y`	Составное присваивание, поддерживаются операции `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

10. **Операция проверки на `null`**

`x ?? y`	Возвращает `x`, если `x` не равно `null`. Иначе возвращает `y`

11. **Лямбда-выражение**

`x => code`	Выполняет блок кода *`code`*

Поясним использование некоторых операций. Для контроля значений, получаемых при работе с числовыми выражениями, в C# предусмотрено использование контролируемого и неконтролируемого контекстов. *Контролируемый контекст* объявляется в форме `checked` *<операторный блок>*, либо как операция `checked`(*<выражение>*). Если при вычислении в контролируемом контексте получается значение, выходящие за пределы целевого типа, то генерируется либо ошибка компиляции (для константных выражений), либо обрабатываемое исключение (для выражений с переменными). *Неконтролируемый контекст* объявляется в форме `unchecked` *<операторный блок>*, либо как операция `unchecked`(*<выражение>*). При использовании неконтролируемого контекста выход за пределы целевого типа ведет к автоматическому «урезанию» результата либо путем отбрасывания бит (целые типы), либо путем округления (вещественные типы). Неконтролируемый контекст применяется в вычислениях по умолчанию.

Арифметические операции `+`, `-`, `*`, `/`, `%` определены для всех числовых типов, за исключением 8- и 16-битовых целых типов. Для коротких целых типов компилятор выполняет неявное преобразование типов (при этом операция с целыми числами должна остаться операцией с целыми числами). Арифметические операции для типов с плавающей запятой не генерируют исключительных ситуаций при переполнении, потере точности или делении на ноль. В результате таких операций получаются особые значения, определённые в виде констант `double.NaN`, `double.NegativeInfinity`, `double.PositiveInfinity` (т.е. «не число», «минус бесконечность», «плюс бесконечность»).

1.6. ОПЕРАТОРЫ

Методы пользовательских типов состоят из операторов, которые выполняются последовательно. Часто используется *операторный блок* – последовательность операторов, заключённая в фигурные скобки.

Операторы объявления

К *операторам объявления* относятся *операторы объявления переменных* и *операторы объявления констант*. Для объявления локальных переменных метода применяется оператор следующего формата:

```
<тип> <имя переменной> [= <начальное значение>];
```

Здесь *<тип>* – тип переменной, *<имя переменной>* – допустимый идентификатор, необязательное *<начальное значение>* – литерал или выражение, соответствующее типу переменной. Локальные переменные методов не могут использоваться в вычислениях, не будучи инициализированы.

Если необходимо объявить несколько переменных одного типа, то идентификаторы переменных можно перечислить через запятую после имени типа. При этом для каждой переменной можно выполнить инициализацию.

```
int a;                  // простейший вариант объявления
```

```
int a = 20;             // объявление с инициализацией
int a, b, c;            // объявление однотипных переменных
int a = 20, b = 10;     // инициализация нескольких переменных
```

Локальная переменная может быть объявлена без указания типа, с использованием ключевого слова `var`. В этом случае компилятор выводит тип переменной из обязательного выражения инициализации.

```
var x = 3;
var y = "Student";
var z = new Student();
```

Не стоит воспринимать переменные, объявленные с `var`, как некие универсальные контейнеры для данных любого типа. Все эти переменные строго типизированы. Так, переменная `x` в приведённом примере имеет тип `int`.

Оператор объявления константы имеет следующий синтаксис:

```
const <тип константы> <имя константы> = <значение>;
```

Здесь *<тип константы>* – это примитивный тип или перечисление, *<значение>* может быть литералом соответствующего типа или результатом действий с другими константами. Примеры объявления констант:

```
const double Pi = 3.1415926;
const double Pi_Plus_2 = Pi + 2;
```

Область доступа к переменной или константе ограничена операторным блоком, содержащим объявление:

```
{
    int i = 10;
}
Console.WriteLine(i); // ошибка компиляции, переменная i не доступна
```

Если операторные блоки вложены друг в друга, то внутренний блок не может содержать объявлений переменных, идентификаторы которых совпадают с переменными внешнего блока:

```
{
    int i = 10;
    {
        int i = 20;     // ошибка компиляции
    }
}
```

Операторы выражений

Операторы выражений – это выражения, одновременно являющиеся допустимыми операторами:

- операция присваивания (включая инкремент и декремент);
- операция вызова метода или делегата;
- операция создания объекта.

Приведём несколько примеров:

```
x = 1 + 2;              // присваивание
x++;                    // инкремент
Console.Write(x);       // вызов метода
new StringBuilder();    // создание объекта
```

Заметим, что при вызове конструктора или метода, возвращающего значение, результат их работы использовать не обязательно.

Операторы перехода

К *операторам перехода* относятся `break`, `continue`, `goto`, `return`, `throw`. Оператор `break` используется для выхода из операторного блока циклов и оператора `switch`. Оператор `break` выполняет переход на оператор за блоком. Оператор `continue` располагается в теле цикла и применяется для запуска новой итерации цикла. Если циклы вложены, то запускается новая итерация того цикла, в котором непосредственно располагается `continue`.

Оператор `goto` передаёт управление на помеченный оператор. Обычно данный оператор употребляется в форме `goto` *<метка>*, где *<метка>* – это допустимый идентификатор. Метка должна предшествовать помеченному оператору и заканчиваться двоеточием, отдельно описывать метки не требуется:

```
goto label;
. . .
label:
A = 100;
```

Оператор `goto` и помеченный оператор должны располагаться в одном операторном блоке. Возможно использование оператора `goto` в одной из следующих форм:

```
goto case <константа>;
goto default;
```

Данные формы обсуждаются при рассмотрении оператора `switch`.

Оператор `return` служит для завершения методов. Оператор `throw` генерирует исключительную ситуацию[1].

Операторы выбора

Операторы выбора – это операторы `if` и `switch`. Оператор `if` в языке C# имеет следующий формат:

```
if (<условие>)
    <оператор или операторный блок 1>
[else
    <оператор или операторный блок 2>]
```

[1] Работа с методами и исключительными ситуациями рассматривается далее.

Здесь *<условие>* – это некоторое булево выражение. Ветвь `else` является необязательной.

Оператор `switch` выполняет одну из групп инструкций в зависимости от значения тестируемого выражения. Синтаксис оператора `switch`:

```
switch (<выражение>)
{
    case <константное выражение>:
       <операторы>
       <оператор перехода>
    case <константное выражение 2>:
       <операторы>
       <оператор перехода>
    . . .
    [default:
       <операторы>
       <оператор перехода>]
}
```

Тестируемое *<выражение>* должно возвращать значение целочисленного типа (включая `char`), булево значение, строку или элемент перечисления. При совпадении тестируемого и константного выражений выполняется соответствующая ветвь `case`. Если совпадения не обнаружено, то выполняется ветвь `default` (если она есть). *<оператор перехода>* – это один из следующих операторов: `break`, `goto`, `return`, `throw`. Оператор `goto` используется с указанием либо ветви `default` (`goto default`), либо определенной ветви `case` (`goto case` *<константное выражение>*).

Хотя после `case` может быть указано только одно константное выражение, при необходимости несколько ветвей `case` можно сгруппировать следующим образом:

```
switch (n)
{
    case 0:
    case 1:
    case 2:
        . . .
}
```

Операторы циклов

К *операторам циклов* относятся операторы `for`, `while`, `do-while`, `foreach`. Для циклов с известным числом итераций используется оператор `for`:

```
for ([<инициализатор>]; [<условие>]; [<итератор>]) <блок (оператор)>
```

Здесь *<инициализатор>* задаёт начальное значение счетчика (или счетчиков) цикла. Для счетчика может использоваться существующая переменная или объявляться новая переменная, время жизни которой будет ограничено циклом (при этом вместо типа переменной допустимо указать `var`). Цикл выполняется,

пока булево *<условие>* истинно, а *<итератор>* определяет изменение счетчика цикла на каждой итерации.

Простейший пример использования цикла `for`:

```
for (int i = 0; i < 10; i++)  // i доступна только в цикле for
    Console.WriteLine(i);     // вывод чисел от 0 до 9
```

В инициализаторе можно объявить и задать начальные значения для нескольких счетчиков одного типа. В этом случае итератор может представлять собой последовательность из нескольких операторов, разделенных запятой:

```
// цикл выполнится 5 раз, на последней итерации i = 4, j = 6
for (int i = 0, j = 10; i < j; i++, j--)
    Console.WriteLine("i = {0}, j = {1}", i, j);
```

Если число итераций цикла заранее неизвестно, можно использовать цикл `while` или цикл `do-while`. Данные циклы имеют схожий синтаксис:

```
while (<условие>) <блок (оператор)>

do
    <блок (оператор)>
while (<условие>);
```

В обоих оператора цикла тело цикла выполняется, пока булево *<условие>* истинно. В цикле `while` условие проверяется в начале очередной итерации, а в цикле `do-while` – в конце. Таким образом, цикл `do-while` всегда выполнится, по крайней мере, один раз. Обратите внимание, *<условие>* должно присутствовать обязательно. Для организации бесконечных циклов на месте условия можно использовать литерал `true`:

```
while (true) Console.WriteLine("Endless loop");
```

Для перебора элементов объектов перечисляемых типов (например, массивов) в C# существует специальный цикл `foreach`:

```
foreach (<тип> <идентификатор> in <коллекция>) <блок (оператор)>
```

В заголовке цикла объявляется переменная, которая будет последовательно принимать значения элементов коллекции. Вместо указания типа этой переменной можно использовать ключевое слово `var`. Присваивание переменной новых значений не отражается на элементах коллекции.

Прочие операторы

К группе *прочих операторов* относятся операторы `lock` и `using`. Первый связан с синхронизацией потоков выполнения, второй – с процессом освобождения локальной переменной[1]. Подробно синтаксис и примеры использования данных операторов рассмотрены в соответствующих разделах.

---

[1] В C# имеется директива `using` для импорта пространств имён. Следует различать *директиву* `using` и *оператор* `using`.

1.7. НАЧАЛЬНЫЕ СВЕДЕНИЯ О МАССИВАХ

Массивы – это ссылочные пользовательские типы. Объявление массива в C# схоже с объявлением переменной, но после указания типа размещается пара квадратных скобок – признак массива:

```
int[] data;
```

Массив является ссылочным типом, поэтому перед началом работы любой массив должен быть создан в памяти. Для этого используется конструктор в форме `new <тип>[<количество элементов>]`.

```
int[] data;
data = new int[10];
```

Создание массива можно совместить с его объявлением:

```
int[] data = new int[10];
```

Созданный массив автоматически заполняется значениями по умолчанию для своего базового типа (ссылочные типы – `null`, числа – `0`, тип `bool` – `false`).

Для доступа к элементу массива указывается имя массива и индекс в квадратных скобках: `data[0] = 10`. Элементы массива нумеруются с нуля, в C# не предусмотрено синтаксических конструкций для указания особого значения нижней границы массива. При выходе индекса массива за допустимый диапазон генерируется исключительная ситуация.

В C# существует способ задания всех элементов массива при создании. Для этого используется список значений в фигурных скобках. При этом можно не указывать количество элементов, а также полностью опустить указание на тип и ключевое слово `new`:

```
int[] data_1 = new int[4] { 1, 2, 3, 5 };
int[] data_2 = new int[] { 1, 2, 3, 5 };
int[] data_3 = new[] { 1, 2, 3, 5 };
int[] data_4 = { 1, 2, 3, 5 };
```

Первые три примера инициализации допускают указание вместо типа переменной ключевого слова `var` (`var data_3 = new[] { 1, 2, 3, 5 }`). Компилятор вычислит тип массива автоматически.

При необходимости можно объявить массивы, имеющие несколько размерностей. Для этого в квадратных скобках после имени типа помещают запятые, «разделяющие» размерности:

```
// двумерный массив d
int[,] d;
d = new int[10, 2];

// трехмерный массив Cube
int[,,] Cube = new int[3, 2, 5];

// объявим двумерный массив и инициализируем его
```

```
int[,] c = new int[2, 4] {
                           {1, 2, 3, 4},
                           {10, 20, 30, 40}
                         };

// то же самое, но немного короче:
int[,] c = { {1, 2, 3, 4}, {10, 20, 30, 40} };
```

В приведённых примерах объявлялись массивы из нескольких размерностей. Такие массивы всегда являются прямоугольными. Можно объявить *массив массивов*, используя следующий синтаксис[1,2]:

```
int[][] table;           // table – массив одномерных массивов
table = new int[2][];    // в table будет 2 одномерных массива
table[0] = new int[2];   // в первом массиве будет 2 элемента
table[1] = new int[20];  // во втором – 20 элементов
table[1][3] = 1000;      // работаем с элементами table:

// совместим объявление и инициализацию массива массивов
int[][] T = { new[] { 10, 20 }, new[] { 1, 2, 3 } };
```

При работе с массивом можно использовать цикл `foreach`, перебирающий все элементы. В цикле `foreach` возможно перемещение по массиву в одном направлении – от начала к концу, при этом попытки присвоить значение элементу массива игнорируются. В следующем фрагменте кода производится суммирование элементов массива:

```
int[] data = { 1, 3, 5, 7, 9 };
var sum = 0;
foreach (var element in data)
    sum += element;
```

В заключение рассмотрим вопрос о приведении типов массивов. Массивы *ковариантны* для ссылочных типов. Это означает, что если ссылочный тип `A` неявно приводим к ссылочному типу `B`, массив с элементами типа `A` может быть присвоен массиву с элементами типа `B`. При этом количество элементов в массиве роли не играет, но массивы должны иметь одинаковую размерность.

```
public class Student { . . . }   // объявление класса

Student[] students = new Student[10];
object[] array = students;       // ковариантность массивов
```

[1] Объявление `int[,][] c` задаёт двумерный массив, состоящий из одномерных массивов.
[2] IL содержит специальные инструкции для работы с одномерными массивами, индексированными с нуля. Поэтому массив массивов обрабатывается быстрее, чем двумерный массив.

Все массивы в платформе .NET могут рассматриваться как классы, являющиеся потомками класса `System.Array`. Описание возможностей этого класса дано в разделе, рассказывающем о работе с коллекциями.

1.8. КЛАССЫ

Класс является основным пользовательским типом. Синтаксис объявления класса в C# следующий:

```
<модификаторы> class <имя класса>
{
    [<элементы класса>]
}
```

Допустимые элементы класса

1. Поле. Синтаксис объявления поля класса совпадает с синтаксисом оператора объявления переменной[1] (тип поля должен всегда быть указан явно, использование `var` не допускается). Если для поля не указано начальное значение, то поле принимает значение по умолчанию для соответствующего типа (для числовых типов - `0`, для типа `bool` - `false`, для ссылочных типов - `null`). Для полей возможно применение модификатора `readonly`, который запрещает изменение поля после его начальной установки.

```
class Person
{
    readonly int _age = 20;
    string _name = "None";
}
```

Поля с модификатором `readonly` похожи на константы, но имеют следующие отличия:

- Тип поля может быть любым.
- Значение поля может быть установлено при объявлении и в конструкторе класса.
- Значение поля вычисляется в момент выполнения, а не компиляции.

2. Константа. Синтаксис объявления константы в классе аналогичен синтаксису, применяемому при объявлении константы в теле метода.

Следующие элементы класса будут подробно рассмотрены в дальнейшем.

3. Метод. Методы описывают функциональность класса.

4. Свойство. Свойства класса призваны предоставить защищенный доступ к полям.

5. Индексатор. Индексатор – это свойство-коллекция, отдельный элемент которого доступен по индексу.

---

[1] Как правило, идентификаторы полей снабжаются неким оговоренным префиксом.

6. Конструктор. Задача конструктора – начальная инициализация объекта или класса.

7. Финализатор. Финализатор автоматически вызывается сборщиком мусора и содержит завершающий код для объекта.

8. Событие. События представляют собой механизм рассылки уведомлений различным объектам.

9. Операция. Язык C# допускает перегрузку некоторых операций для объектов класса.

10. Вложенный пользовательский тип. Описание класса может содержать описание другого пользовательского типа – класса, структуры, перечисления, интерфейса, делегата. Обычно вложенные типы выполняют вспомогательные функции и явно вне основного типа не используются.

Модификаторы доступа для элементов и типов

Для поддержания принципа инкапсуляции элементы класса могут снабжаться специальными *модификаторами доступа*:

- `private`. Элемент с данным модификатором доступен только в том типе, в котором определён. Например, поле доступно только в содержащем его классе.
- `protected`. Элемент виден в типе, в котором определен, и в наследниках этого типа (даже если наследники расположены в других сборках). Данный модификатор может применяться только в типах, поддерживающих наследование, то есть в классах.
- `internal`. Элемент доступен без ограничений, но только в той сборке, где описан.
- `protected internal`. Комбинация модификаторов `protected` и `internal`. Элемент виден в содержащей его сборке без ограничений, а вне сборки – только в наследниках типа (т.е. `protected` *или* `internal`[1]).
- `public`. Элемент доступен без ограничений как в той сборке, где описан, так и в других сборка, к которым подключается сборка с элементом.

По умолчанию (без указания) для всех элементов типа применяется модификатор `private`. Для локальных переменных методов и операторных блоков модификаторы доступа не используются.

При описании самостоятельного класса допустимо указать для него модификаторы `public` или `internal` (`internal` применяется по умолчанию). Если же класс вложен в другой пользовательский тип, то такой класс можно объявить с любым модификатором доступа. Заметим, что у `internal`-класса `public`-элементы за пределами сборки не видны.

[1] В CLR имеется модификатор доступа, соответствующий `protected` **и** `internal`. При помощи языка C# такой уровень доступа описать нельзя.

Разделяемые классы

Хорошей практикой программирования считается размещение каждого класса в отдельном файле. Однако иногда классы получаются настолько большими, что указанный подход становится непрактичным. Это часто справедливо при использовании средств автоматической кодогенерации. *Разделяемые классы* (*partial classes*) – это классы, разбитые на несколько фрагментов, описанных в отдельных файлах с исходным кодом[1].

Для объявления разделяемого класса используется модификатор `partial`:

```
// файл part1.cs
partial class BrokenClass
{
    private int someField;
    private string anotherField;
}

// файл part2.cs
partial class BrokenClass
{
    public void Method() { }
}
```

Все фрагменты разделяемого класса должны быть доступны во время компиляции, так как «сборку» типа выполняет компилятор. Еще одно замечание касается использования модификаторов, применяемых к классу. Модификаторы доступа должны быть одинаковыми у всех фрагментов. Если же к одному из фрагментов применяется модификатор `sealed` или `abstract`, то эти модификаторы считаются применёнными ко всем фрагментам, то есть к классу в целом.

Использование класса

Чтобы использовать класс после объявления (то есть, получить доступ к его открытым экземплярным элементам[2]), необходима переменная класса – *объект*. Объект объявляется как обычная переменная:

```
<имя класса> <имя объекта>;
```

Так как класс – ссылочный тип, то объекты должны быть инициализированы до непосредственного использования. Для инициализации объекта используется операция `new` - вызов конструктора класса. Если конструктор не описывался, применяется предопределенный конструктор без параметров с именем класса:

```
<имя объекта> = new <имя класса>();
```

---

[1] Разделяемыми могут быть не только классы, но структуры и интерфейсы.

[2] Для доступа к открытой константе класса применяется синтаксис *<имя класса>.<имя константы>*.

Инициализацию объекта можно совместить с его объявлением:

```
<имя класса> <имя объекта> = new <имя класса>();
```

Доступ к экземплярным элементам класса через объект осуществляется по синтаксису *<имя объекта>.<имя элемента>*.

1.9. МЕТОДЫ

Методы в языке C# являются неотъемлемой частью описания таких пользовательских типов как класс или структура. В C# не существует глобальных методов – любой метод должен быть членом класса или структуры.

Описание метода

Рассмотрим общий синтаксис описания метода:

```
<модификаторы> <тип> <имя метода>([<параметры>]) <тело метода>
```

Здесь *<тип>* – это тип возвращаемого методом значения. Допустимо использование любого примитивного или пользовательского типа. В C# формально не существует процедур – любой метод является функцией, возвращающей значение. Для «процедуры» в качестве типа указывается специальное ключевое слово `void`. После имени метода всегда следует пара круглых скобок, в которых указывается список формальных параметров метода (если этот список не пуст).

Список формальных параметров метода – это набор элементов, разделенных запятыми. Каждый элемент имеет следующий формат:

```
[<модификатор>] <тип> <имя формального параметра> [= <значение>]
```

Существуют четыре вида параметров, которые специфицируются модификатором:

1. *Параметры-значения* – объявляются без модификатора;
2. *Параметры, передаваемые по ссылке* – используют модификатор `ref`;
3. *Выходные параметры* – объявляются с модификатором `out`;
4. *Параметры-списки* – применяется модификатор `params`.

Параметры, передаваемые по ссылке и по значению, ведут себя аналогично тому, как это происходит в других языках программирования. *Выходные параметры* подобны ссылочным, то есть при работе с ними в теле метода не создается копия фактического параметра. Компилятор отслеживает, чтобы выходным параметрам в теле метода обязательно было присвоено значение.

Параметры-списки позволяют передать в метод любое количество аргументов. Метод может иметь не более одного параметра-списка, который обязательно должен быть последним в списке формальных параметров. Тип параметра-списка объявляется как массив, и работа с таким параметром происходит в методе как с массивом. Каждый аргумент из передаваемого в метод списка ведет себя как параметр, переданный по значению.

При объявлении метода для параметров-значений допустимо указать значение параметра по умолчанию. Так определяется *опциональный параметр*. Опциональные параметры должны быть указаны в конце списка формальных параметров метода.

Для выхода из метода служит оператор `return` или оператор `throw`. Если тип метода не `void`, то после `return` обязательно указывается возвращаемое значение (тип этого значения должен совпадать с типом метода или неявно приводится к нему). Кроме этого, оператор `return` или оператор `throw` должны встретиться в таком методе во всех ветвях кода, по крайней мере, один раз.

Рассмотрим несколько примеров объявления методов.

1. Простейшее объявление метода-процедуры без параметров:

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

2. Метод без параметров, возвращающий целое значение:

```
int ReturnInt()
{
    Console.WriteLine("Hello!");
    return 5;
}
```

3. Функция `Add()` выполняет сложение двух аргументов:

```
int Add(int a, int b)
{
    return a + b;
}
```

4. Функция `ReturnTwoValues()` возвращает `10` как результат своей работы, кроме этого значение параметра `a` устанавливается равным `100`:

```
int ReturnTwoValues(out int a)
{
    a = 100;
    return 10;
}
```

5. Метод `PrintList()` использует параметр-список:

```
void PrintList(params int[] list)
{
    foreach(int item in list)
        Console.WriteLine(item);
}
```

6. Метод `AddWithOptional()` показывает использование опционального параметра:

```
int AddWithOptional(int x, int y = 5)
{
    return x + y;
}
```

В экземплярных методах доступен параметр `this` (в методах класса – только для чтения, в методах структуры – и для чтения, и для записи). Это ссылка на текущий экземпляр. Данную ссылку можно применять для устранения конфликта имен (если имя элемента типа совпадает с именем параметра метода):

```
class Pet
{
    private int age;
    private string name;

    public void SetAge(int age)
    {
        this.age = age;
    }
}
```

C# позволяет выполнить *перегрузку методов* в пользовательских типах. Перегруженные методы имеют одинаковое имя, но разную сигнатуру. *Сигнатура* – это упорядоченный набор из модификаторов и типов формальных параметров. Если две версии перегруженного метода различаются только модификатором `params`, они считаются не различимыми:

```
// код не компилируется – методы Foo() различить нельзя!
void Foo(params int[] a) { . . . }
void Foo(int[] a) { . . . }
```

Если одна версия метода как признак отличия содержит модификатор `ref`, а другая - `out`, то методы также не различимы с точки зрения компилятора:

```
// код не компилируется – методы Foo() различить нельзя!
void Foo(out int a) { . . . }
void Foo(ref int a) { . . . }
```

Однако если одна версия метода содержит модификатор `ref` или `out`, а другая нет, то методы различимы:

```
// код компилируется
void Foo(out int a) { . . . }
void Foo(int a) { . . . }
```

Вызов метода

При вызове метода на место формальных параметров помещаются фактические аргументы. Соответствие между параметром и аргументом устанавливается либо по позиции, либо используя синтаксис *именованных аргументов*:

```
<имя формального параметра> : <выражение для аргумента>
```

Рассмотрим примеры вызова метода `Add()`, содержащего три параметра:

```
int Add(int x, int y = 3, int z = 5)
{
    return x + y + z;
}

int res_1 = Add(10, 20, 30);         // передача по позиции
int res_2 = Add(x:10, z:20, y:30);   // именованные параметры
int res_3 = Add(10, z:20, y:30);     // комбинирование двух способов
```

Использование именованных аргументов зачастую необходимо, если метод содержит опциональные параметры:

```
int res_4 = Add(10, z:20);
```

Метод с параметром-списком можно вызвать несколькими способами. Можно передать методу произвольное количество аргументов указанного типа или массив целых значений:

```
// передаем два аргумента
PrintList(10, 20);
// теперь передаем четыре аргумента
PrintList(1, 2, 3, 4);
// создаем и передаем массив целых чисел
PrintList(new[] { 10, 20, 30, 40 });
// можем вообще ничего не передавать
PrintList();
```

Если при описании параметра использовались модификаторы `ref` или `out`, то они должны быть указаны и при вызове. Кроме этого, фактические аргументы с такими модификаторами должны быть представлены переменными, а не литералами или выражениями. В случае параметров-значений тип аргумента должен совпадать или неявно приводится к типу формального параметра. При согласовании типов в случае возникновения двусмысленности делается выбор числового типа из той же группы знаковости. Например, пусть имеются перегруженные методы `M(uint x)` и `M(int x)`, а переменная `y` имеет тип `ushort`. Тогда вызов `M(y)` означает вызов версии с формальным параметром типа `uint`. Для `ref`- и `out`-параметров требуется абсолютное совпадение типов.

Разделяемые методы

Разделяемые классы и структуры могут содержать *разделяемые методы*. Разделяемый метод состоит из двух частей: заголовка и реализации. Обычно эти части размещаются в различных частях разделяемого типа. Выглядит это следующим образом:

```
public partial class Student
{
    partial void M(int x);
}

public partial class Student
{
    partial void M(int x)
    {
        Console.WriteLine("M body");
    }
}
```

Разделяемые методы подчиняются следующим правилам:

- Объявление метода начинается с модификатора `partial`;
- Метод обязан возвращать значение `void`;
- Метод может иметь параметры, но `out`-параметры запрещены;
- Метод неявно объявляется как `private`. Поэтому он не может быть виртуальным;
- Разделяемые методы могут быть статическими или универсальными;
- Вызов разделяемого метода нельзя инкапсулировать в делегат.

Отметим еще одну особенность разделяемого метода: его реализация может быть опущена. В этом случае компилятор даже не генерирует код вызовов разделяемого метода.

1.10. СВОЙСТВА И ИНДЕКСАТОРЫ

Свойства класса призваны предоставить защищенный доступ к полям. Как и в большинстве объектно-ориентированных языков, в C# непосредственная работа с полями не приветствуется. Поля класса обычно объявляются как `private`-элементы, а для доступа к ним используются свойства.

Рассмотрим базовый синтаксис описания свойства:

```
<модификаторы> <тип свойства> <имя свойства>
{
    get {<блок кода>}
    set {<блок кода>}
}
```

Синтаксис описания заголовка свойства напоминает синтаксис описания обычного поля. Тип свойства обычно совпадает с типом того поля, для обслу-

живания которого свойство создается. У свойства присутствует специальный блок, содержащий методы для доступа к свойству. Данный блок состоит из `get`-части и `set`-части, далее называемых *аксессор* и *мутатор* соответственно. Одна из частей может отсутствовать, так получается *свойство только для чтения* или *свойство только для записи*. Аксессор отвечает за возвращаемое свойством значение и работает как функция. Мутатор работает как процедура, устанавливающая значение свойства. Считается, что параметр, передаваемый в мутатор, имеет специальное имя `value`.

Рассмотрим пример класса, имеющего свойства:

```
public class Student
{
    private int _age;
    private string _name;

    public int Age
    {
        get { return _age; }
        set
        {
            // проверка корректности
            _age = value < 0 ? 0 : value;
        }
    }

    public string Name
    {
        get { return "My name is " + _name; }
        set { _name = value; }
    }
}
```

Свойства транслируются при компиляции в вызовы методов. В скомпилированный код класса добавляются методы со специальными именами `get_`*`Name`*`()` и `set_`*`Name`*`()`, где *`Name`* – это имя свойства. Побочным эффектом такой трансляции является тот факт, что пользовательские методы с данными именами допустимы в классе, только если они имеют сигнатуру, отличающуюся от методов, соответствующих свойству.

Как правило, свойства открыты, то есть снабжаются модификатором доступа `public`. Однако иногда логика класса требует разделения права доступа чтения и записи свойства. Например, чтение позволено всем, а запись – только из методов того класса, где свойство объявлено. В C# разрешено при описании свойства указывать модификаторы доступа для аксессоров и мутаторов. При этом действуют два правила. Во-первых, модификатор может быть только у одной из частей. Во-вторых, он должен понижать видимость части по сравнению с видимостью всего свойства:

```
public class SomeClass
{
    public int Prop
    {
        get { return 0; }
        private set { }
    }
}
```

Достаточно часто свойство содержит только простейший код доступа к полю. Вот фрагмент класса с таким свойством:

```
public class Person
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

Чтобы облегчить описание таких свойств-«обёрток», в C# имеются *автосвойства* (*auto properties*). Используя автосвойства, приведённый фрагмент кода можно переписать следующим образом:

```
public class Person
{
    public string Name { get; set; }
}
```

В этом случае компилятор сам сгенерирует необходимое поле класса, связанное со свойством. Обратите внимание: в автосвойстве должны присутствовать и часть `get`, и часть `set`. При необходимости получить аналог классических свойств только для чтения необходимо использовать модификаторы доступа для частей:

```
public class Person
{
    public string Name { get; private set; }
}
```

Кроме скалярных свойств язык C# поддерживает *индексаторы*. При помощи индексаторов осуществляется доступ к коллекции данных, содержащихся в объекте, с использованием привычного синтаксиса для доступа к элементам массива – пары квадратных скобок и индекса.

Объявление индексатора напоминает объявление обычного свойства:

```
<модификаторы> <тип> this[<параметры>] { <get и set блоки> }
```

Параметры индексатора служат для описания типа и имён индексов, применяемых для доступа к данным. Параметры индексатора могут быть описаны как параметры-значения или как параметр-список (с использованием `params`). Также допустимо использование опциональных параметров.

Рассмотрим пример класса, содержащего индексатор. Пусть данный класс описывает студента с набором оценок:

```
public class Student
{
    private readonly int[] _marks = new int[5];

    public int this[int i]
    {
        get { return Belongs(i, 0, 4) ? _marks[i] : 0; }
        set
        {
            if (Belongs(i, 0, 4) && Belongs(value, 1, 10))
                _marks[i] = value;
        }
    }

    private bool Belongs(int x, int min, int max)
    {
        return (x >= min) && (x <= max);
    }
}
```

При использовании индексатора указывается имя объекта и значение индекса (или индексов) в квадратных скобках. Допустимы *именованные индексы* (по аналогии с именованными аргументами метода). Если необходимо использовать индексатор в пределах класса, применяют синтаксис `this[<значение>]`.

```
var student = new Student();
student[1] = 8;
student[3] = 4;
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(student[i]);
}
```

В одном классе нельзя объявить два индексатора, у которых совпадают типы параметров. Можно объявить индексаторы, у которых параметры имеют разный тип или количество параметров различается[1].

[1] Индексаторы транслируются в методы с именами `get_Item()` и `set_Item()`. Изменить имена методов можно, используя атрибут `[IndexerName]`.

1.11. СТАТИЧЕСКИЕ ЭЛЕМЕНТЫ И МЕТОДЫ РАСШИРЕНИЯ

Статические элементы

Поля, методы и свойства классов, которые рассматривались в предыдущих параграфах, использовались при помощи объекта класса. Такие элементы называются *экземплярными*. *Статические* элементы предназначены для работы не с объектами, а с классом. Статические поля хранят информацию, общую для всех объектов, статические методы либо вообще не используют поля, либо работают только со статическими полями.

Чтобы объявить статический элемент, применяется модификатор `static`:

```
public class Account
{
    private static double _tax = 0.1;

    public static double GetTax()
    {
        return _tax * 100;
    }
}
```

Для вызова статических элементов требуется использовать имя класса:

```
Console.WriteLine(Account.GetTax());
```

Подчеркнём, что статическими могут быть сделаны поля, методы и обычные свойства. Открытая константа, описанная в классе, уже работает как статический элемент. Индексатор класса не может быть статическим[1].

Статические классы

Если класс содержит только статические элементы, то при объявлении класса можно указать модификатор `static`. Так определяется *статический класс*:

```
public static class ApplicationSettings
{
    public static string BaseDir { }
    public static string GetRelativeDir() { }
}
```

Экземпляр статического класса не может быть создан или даже объявлен в программе. Все открытые элементы статического класса доступны только с использованием имени класса.

Упомянем некоторые полезные статические классы из пространства имен `System`. Для преобразования данных различных типов удобно использовать

[1] В отличие от языка C#, CLR позволяет создавать статические индексаторы.

класс `Convert`. Класс `Math` содержит большой набор различных математических функций. Класс `Console` предназначен для чтения и записи информации на консоль, а также для настройки консоли. Класс `Environment` содержит полезные свойства, описывающие окружение запуска приложения (например, `Version` – текущая версия платформы .NET).

Методы расширения

В C# 3.0 была представлена концепция *метода расширения* (*extension method*). Методы расширения позволяют «добавлять» методы в существующие типы без создания нового производного типа, перекомпиляции или иного изменения исходного типа. Методы расширения являются особым видом статического метода, но они вызываются, как если бы они были методами экземпляра в расширенном типе. Для клиентского кода нет видимого различия между вызовом метода расширения и вызовом методов, фактически определенных в типе.

Рассмотрим следующий пример. Пусть разрабатывается программный проект, в различных частях которого требуется подсчет суммы элементов целочисленного массива. Для реализации данной задачи создан специальный вспомогательный класс, содержащий статический метод подсчёта:

```
public static class ArrayHelper
{
    public static int Sum(int[] array)
    {
        var result = 0;
        foreach (var item in array)
        {
            result += item;
        }
        return result;
    }
}

// использование
int[] m = { 3, 4, 6 };
Console.WriteLine(ArrayHelper.Sum(m));
```

Превратим `Sum()` в метод расширения. Для этого достаточно добавить ключевое слово `this` перед первым аргументом метода:

```
public static class ArrayHelper
{
    public static int Sum(this int[] array)
    { . . . }
}
```

Теперь метод `Sum()` можно вызывать как традиционным способом, так и как экземплярный метод массива:

```
int[] m = { 3, 4, 6 };
Console.WriteLine(m.Sum());
```

Подчеркнём, что методами расширения могут быть только статические методы статических классов. Количество параметров такого метода произвольно (один и более), но только первый можно указать с модификатором `this`. Соответственно, метод расширит тип первого параметра.

Методы расширения применимы к типу, как только импортируется пространство имен, содержащее класс с этими методами расширения. Если выполняется импорт нескольких пространств имен, содержащих классы с одинаковыми методами расширения для одного типа, то возникает ошибка компиляции. В этом случае методы расширения должны использоваться как обычные статические методы некоего вспомогательного класса.

1.12. КОНСТРУКТОРЫ И ИНИЦИАЛИЗАЦИЯ ОБЪЕКТОВ

Конструктор выполняет начальную инициализацию объекта или класса. Синтаксис описания конструктора напоминает синтаксис описания метода. Однако имя конструктора всегда совпадает с именем класса, а любое указание на тип возвращаемого значения отсутствует (даже `void`).

Задача *экземплярных конструкторов* – создание и инициализация объекта. Любой экземплярный конструктор в начале свой работы выполняет размещение объекта в динамической памяти и инициализацию полей объекта. Различают два вида экземплярных конструкторов – *конструкторы по умолчанию* и *пользовательские конструкторы*.

Конструктор по умолчанию автоматически создаётся компилятором, если программист не описал в классе собственный конструктор. Конструктор по умолчанию – это всегда конструктор без параметров.

```
// класс Pet не содержит описания конструктора
public class Pet
{
    public int Age;
    public string Name = "I'm a pet";
}

var dog = new Pet();            // вызов конструктора по умолчанию
Console.WriteLine(dog.Age);     // выводит 0
Console.WriteLine(dog.Name);    // выводит I'm a pet
```

Пользовательский конструктор описывается программистом. Класс может содержать несколько пользовательских конструкторов, однако они обязаны различаться сигнатурой. Если в классе определён хотя бы один пользовательский конструктор, конструктор по умолчанию уже не создается.

```
// класс Pet содержит два пользовательских конструктора
public class Pet
{
    public int Age;
    public string Name = "I'm a pet";

    public Pet()
    {
        Age = 0;
        Name = "Pet";
    }

    public Pet(int age, string name)
    {
        Age = age;
        Name = name;
    }
}
```

Пользовательские конструкторы могут применяться для начальной инициализации `readonly`-полей (`readonly`-поля доступны для записи, но только в конструкторе). Пользовательский конструктор может вызвать другой конструктор того же класса, но только в начале своей работы. Для этого при описании конструктора используется синтаксис, аналогичный приведённому в следующем примере[1]:

```
public Pet() : this(10, "Pet") { . . . }
```

Для вызова экземплярных конструкторов используется операция `new`, которая возвращает созданный объект. У объекта нельзя вызвать конструктор как метод (т.е. в виде *<имя объекта>.<имя конструктора>*).

```
// вызов конструкторов для создания объекта
var cat = new Pet();
var dog = new Pet(5, "Buddy");
```

Статические конструкторы используются для начальной инициализации статических полей класса. Статический конструктор всегда объявляется с модификатором `static` и без параметров. Область видимости у статических конструкторов не указывается. В теле статического конструктора возможна работа только со статическими полями и методами класса. Статический конструктор не может вызывать экземплярные конструкторы в начале своей работы.

```
public class Account
{
    private static double _tax;
```

[1] Использование при описании конструктора опциональных параметров уменьшает необходимость подобных вызовов.

```
    static Account()
    {
        _tax = 0.1;
    }
}
```

Статические конструкторы вызываются не программистом, а общеязыковой средой исполнения в следующих случаях:

- перед созданием первого объекта класса или при первом обращении к элементу класса, не унаследованному от предка;
- перед первым обращением к статическому полю, не унаследованному от предка.

При работе с объектами достаточно типичным является следующий код: вначале объект создается, а затем настраивается путём установки свойств.

```
Student s = new Student();
s.Name = "Mr. Spiderman";
s.Age = 25;
```

C# позволяет совместить создание объекта с его настройкой. Для этого после параметров конструктора в фигурных скобках перечисляются требуемые `public`-элементы класса (обычно это свойства) и их значения (если конструктор без параметров, можно не указывать круглые скобки после его имени):

```
Student s = new Student { Name = "Mr. Spiderman", Age = 25 };
```

Инициализация объектов действует и для классов-коллекций[1]. Предполагается, что такой класс реализует интерфейс `IEnumerable` и имеет `public`-метод `Add()`. Именно этот метод вызывает компилятор, когда обрабатывает код инициализации.

```
var x = new List<int> { 2, 3, 10 };
var y = new List<string> { "There", "is", "no", "spoon" };
```

Инициализация коллекций работает, даже если у метода `Add()` несколько параметров. В таком случае эти параметры записываются в фигурных скобках:

```
var cars = new Dictionary<int, string> {{1, "Ford" }, {2, "Opel" }};
```

1.13. НАСЛЕДОВАНИЕ КЛАССОВ

Язык C# полностью поддерживает объектно-ориентированную концепцию наследования для классов. Чтобы указать, что один класс является наследником другого, используется следующий синтаксис:

```
class <имя класса-наследника> : <имя класса-предка> {<тело класса>}
```

[1] Стандартные классы для представления коллекций, такие как `List<T>` и `Dictionary<K, V>`, будут рассмотрены в соответствующем разделе.

Наследование от двух и более классов в C# запрещено. Наследник обладает всеми полями, методами и свойствами предка, но элементы предка с модификатором `private` не доступны в наследнике. Конструкторы класса-предка не переносятся в класс-наследник. При наследовании также нельзя расширить область видимости класса: `internal`-класс может наследоваться от `public`-класса, но не наоборот.

Для объектов класса-наследника определено неявное преобразование к типу класса-предка. C# содержит две специальные операции, связанные с контролем типов при наследовании. Выражение `x is T` возвращает значение `true`, если тип объекта `x` это `T` или наследник класса `T`. Выражение `x as T` возвращает объект, приведенный к типу `T`, если это возможно, и `null` в противном случае.

Для обращения к методам непосредственного предка класс-наследник может использовать ключевое слово `base` в форме `base.`*`<метод базового класса>`*. Если конструктор наследника должен вызвать конструктор предка, то для этого также используется ключевое слово `base`:

`<конструктор наследника>``([`*`<параметры>`*`]): base([`*`<параметры_2>`*`])`

Для конструкторов производного класса справедливо следующее замечание: вначале работы конструктор должен совершить вызов другого конструктора своего или базового класса. Если вызов конструктора базового класса отсутствует, компилятор автоматически подставляет в заголовок конструктора вызов `: base()`. Если в базовом классе нет конструктора без параметров, происходит ошибка компиляции.

Для классов можно указать два модификатора, связанных с наследованием. Модификатор `sealed` задает класс, от которого запрещено наследование. Модификатор `abstract` задает *абстрактный класс*, у которого обязательно должны быть наследники. Объект абстрактного класса создать нельзя, хотя статические элементы такого класса можно вызвать:

```
sealed class FinishedClass { }
abstract class AbstractClass { }
```

Класс-наследник может дополнять базовый класс новыми элементами, а может замещать элементы базового класса. Для замещения достаточно указать в новом классе элемент с прежним именем и, возможно, новой сигнатурой:

```
public class Pet
{
    public void Speak() { Console.WriteLine("I'm a pet"); }
}

public class Dog : Pet
{
    public void Speak() { Console.WriteLine("I'm a dog"); }
}
```

При компиляции данного фрагмента будет получено предупреждающее сообщение о том, что метод `Dog.Speak()` закрывает метод базового класса `Pet.Speak()`. Чтобы подчеркнуть, что метод класса-наследника сознательно замещает метод базового класса, используется ключевое слово `new`:

```
public class Dog : Pet
{
    public new void Speak() { Console.WriteLine("I'm a dog"); }
}
```

При замещении методов с изменением типов параметров метод базового класса вызвается только в том случае, если компилятор не может подобрать метод производного класса, выполняя неявное приведение типов:

```
class A
{
    public void Do(int x) { Console.WriteLine("A.Do()"); }
}

class B : A
{
    public void Do(double x) { Console.WriteLine("B.Do()"); }
}

B x = new B();
x.Do(3);                // печатает "B.D()"
```

Замещение методов класса не является полиморфным по умолчанию. Следующий фрагмент кода печатает две одинаковые строки:

```
Pet pet = new Pet(), dog = new Dog();
pet.Speak();            // печатает "I'm a pet"
dog.Speak();            // так же печатает "I'm a pet"
```

Для организации полиморфного вызова методов применяется пара модификаторов - `virtual` и `override`: `virtual` указывается для метода базового класса, который мы хотим сделать полиморфным, `override` – для методов производных классов. Эти методы должны совпадать по имени, типу и сигнатуре с перекрываемым методом класса-предка.

```
public class Pet
{
    public virtual void Speak() { Console.WriteLine("I'm a pet"); }
}

public class Dog : Pet
{
    public override void Speak() { Console.WriteLine("I'm a dog"); }
}
```

```
Pet pet = new Pet(), dog = new Dog();
pet.Speak();          // печатает "I'm a pet"
dog.Speak();          // печатает "I'm a dog"
```

При описании метода возможно совместное указание модификаторов `new` и `virtual`. Такой приём создаёт новую полиморфную цепочку замещения.

```
class A
{
    public virtual void Do() { Console.WriteLine("A.Do()"); }
}

class B : A
{
    public override void Do() { Console.WriteLine("B.Do()"); }
}

class C : A
{
    public new virtual void Do() { Console.WriteLine("C.Do()"); }
}

A[] x = { new A(), new B(), new C() };
x[0].Do();            // печатает "A.Do()"
x[1].Do();            // печатает "B.Do()"
x[2].Do();            // печатает "A.Do()"
```

Если на некоторой стадии построения иерархии классов требуется запретить дальнейшее переопределение виртуального метода в производных классах, этот метод помечается ключевым словом `sealed`:

```
public class Dog : Pet
{
    public override sealed void Speak() { }
}
```

Для методов абстрактных классов (классов с модификатором `abstract`) можно задать модификатор `abstract`, который говорит о том, что метод не реализуется в классе, а должен обязательно переопределяться в наследнике (в такой ситуации модификатор `abstract` эквивалентен модификатору `virtual`).

```
abstract class AbstractClass
{
    // реализации метода в классе нет
    public abstract void AbstractMethod();
}
```

Отметим, что наряду с виртуальными методами в классе можно описать виртуальные свойства, индексаторы и события. Статические элементы класса не могут быть виртуальными.

1.14. КЛАСС SYSTEM.OBJECT И ИЕРАРХИЯ ТИПОВ

Диаграмма, показанная на рис. 2, связывает базовые типы платформы .NET с точки зрения отношения наследования.

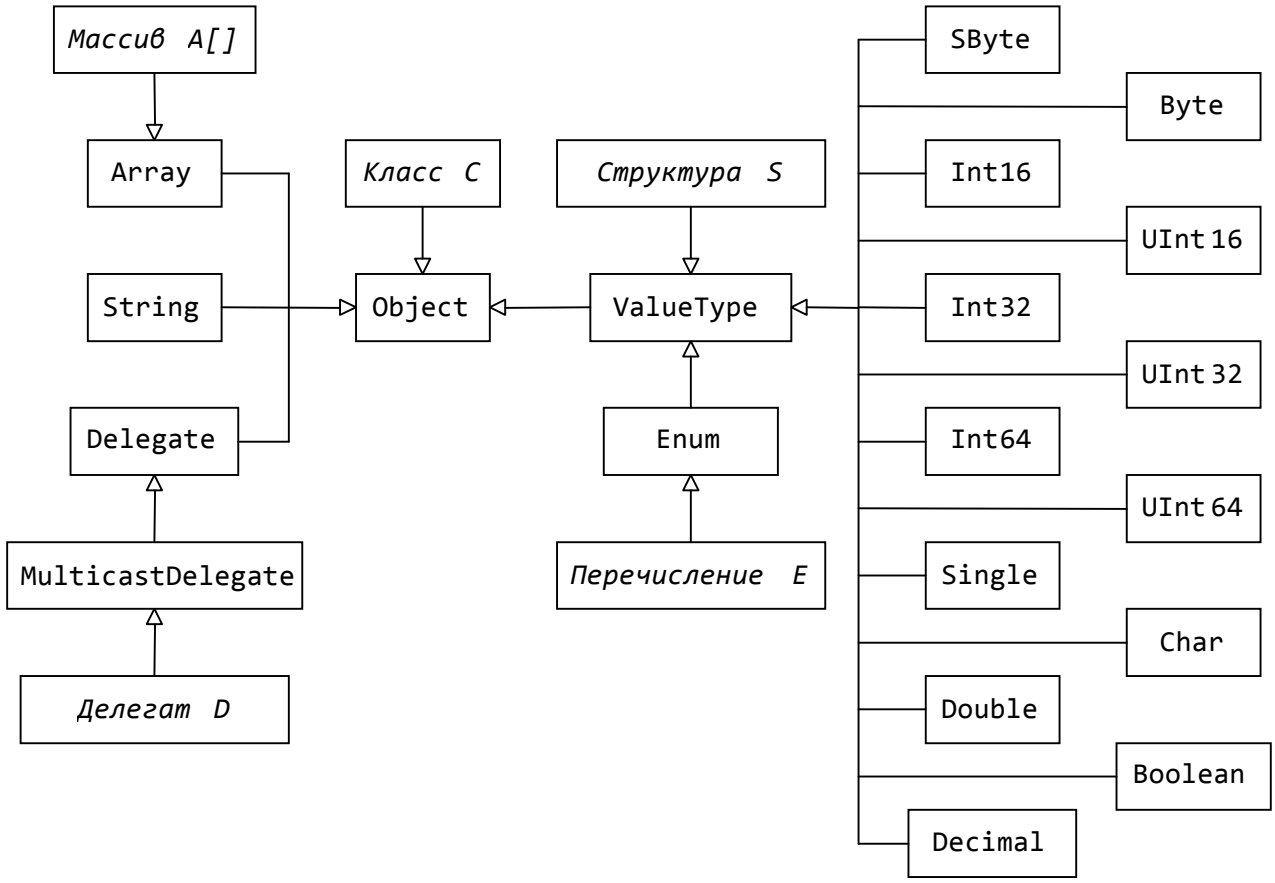

Рис. 2. Иерархия базовых типов платформы .NET

Все типы в .NET Framework наследуются (прямо или косвенно) от класса `System.Object`[1] (в C# для этого типа используется псевдоним `object`). Тип `System.ValueType` является предком всех типов значений (включая числовые типы, пользовательские структуры и перечисления). Массивы наследуются от класса `System.Array`, а класс `System.Delegate` является предком всех делегатов.

Рассмотрим элементы класса `System.Object` в алфавитном порядке.

1. `public virtual bool Equals(object obj)`

Данный метод определяет, равен ли объект `obj` текущему объекту. Реализация `Equals()` по умолчанию обеспечивает равенство ссылок для ссылочных типов и побитовое равенство для типов значений. Пользовательский тип может переопределять метод `Equals()`. При этом должны выполняться такие правила:

[1] Формально, от `object` не наследуются типы-указатели, используемые в неуправляемом коде (например, `int*`), а также интерфейсы (но интерфейсы приводятся к `object`).

- `x.Equals(x) == true`.
- `x.Equals(y) == y.Equals(x)`.
- `(x.Equals(y) && y.Equals(z)) == true` ⟺ `x.Equals(z) == true`.
- Вызовы метода `x.Equals(y)` возвращают одинаковое значение до тех пор, пока объекты `x` и `y` остаются неизменными.
- `x.Equals(null) == false`, если `x != null`.
- Метод `Equals()` не должен генерировать исключений.

Типы, переопределяющие метод `Equals()`, должны также переопределять метод `GetHashCode()` (и наоборот); в противном случае коллекции-словари могут работать неправильно.

Если применяется перегрузка операции равенства для заданного типа, то этот тип должен переопределять метод `Equals()`. Реализация метода `Equals()` должна возвращать те же результаты, что и перегруженная операция равенства.

2. `public static bool Equals(object a, object b)`

Метод определяет, считаются ли равными экземпляры `a` и `b`. Для этого вызывается экземплярный метод `Equals()` у объекта `a` и объекта `b`.

3. `protected virtual void Finalize()`

Метод `Finalize()` позволяет объекту попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект будет утилизирован в процессе сборки мусора.

4. `public virtual int GetHashCode()`

Метод `GetHashCode()` играет роль хэш-функции для определенного типа. Этот метод можно использовать в алгоритмах хэширования и таких структурах данных, как хэш-таблицы.

Реализация метода `GetHashCode()` по умолчанию не гарантирует уникальность возвращаемых кодов. Пользовательские типы могут переопределять данный метод для эффективного вычисления хэш-функции. Если два объекта при сравнении оказались равны, методы `GetHashCode()` этих объектов должны возвращать одинаковые значения. Однако если при сравнении оказалось, что объекты не равны, методы `GetHashCode()` не обязательно должны возвращать разные значения.

5. `public Type GetType()`

Данный метод возвращает объект `System.Type` для текущего экземпляра. Объект `System.Type` представляет метаданные, связанные с классом текущего экземпляра.

6. `protected object MemberwiseClone()`

Метод `MemberwiseClone()` применяется для создания неполной копии объекта. Метод создает новый объект, а затем копирует в него нестатические поля текущего объекта. Если поле относится к типу значения, выполняется побитовое копирование полей. Если поле относится к ссылочному типу, копируются ссылки, а не объекты, на которые они указывают; следовательно, ссылки в исходном объекте и его клоне указывают на один и тот же объект.

7. `public static bool ReferenceEquals(object a, object b)`

Этот статический метод возвращает значение `true`, если параметр `a` соответствует тому же экземпляру, что и параметр `b`, или же оба они равны `null`; в противном случае метод возвращает `false`.

8. `public virtual string ToString()`

Метод `ToString()` возвращает строку, которой представлен текущий объект. Метод может быть переопределен в производном классе для возврата адекватных значений для данного типа.

Так как `System.Object` является предком любого типа, переменной типа `object` можно присвоить любую переменную. Если для ссылочных типов при этом происходит только присваивание указателей, для типов значений выполняется специальная операция, называемая *операцией упаковки* (*boxing*)[1]. При упаковке в динамической памяти создается объект, содержащий значение переменной и информацию о её типе. Упакованный объект может быть в дальнейшем подвергнут обратному преобразованию – *операции распаковки* (*unboxing*). По форме распаковка выглядит как приведение типов.

```
int i = 123;
object o = i;          // упаковка
int j = (int)o;        // распаковка
```

1.15. СТРУКТУРЫ

Структура – это пользовательский тип значения, поддерживающий всю функциональность класса, кроме наследования. Пользовательская структура в простейшем случае позволяет инкапсулировать несколько полей различных типов. Но элементами структуры могут быть не только поля, а и методы, свойства, события, константы. Структуры также могут реализовывать интерфейсы.

Синтаксис определения структуры следующий:

```
<модификаторы> struct <имя структуры>
{
    <элементы структуры>
}
```

При описании полей структуры следует учитывать, что они не могут быть инициализированы при объявлении. Как и класс, структура может содержать конструкторы. Однако в структуре можно объявить только экземплярный конструктор с параметрами[2], причём в теле конструктора необходимо инициализировать все поля структуры. Еще одно отличие структуры от класса - в структуре указатель на экземпляр `this` доступен не только для чтения, но и для записи.

Рассмотрим пример структуры для представления точки в пространстве:

---

[1] Операция упаковки выполняется и в случае, когда переменной типа интерфейс присваивается переменная типа значения. Этот аспект будет разобран при рассмотрении интерфейсов.

[2] В структуре можно объявить статический конструктор, но он никогда не вызывается CLR.

```
public struct Point3D
{
    public double X, Y, Z;

    public Point3D(double x, double y)
    {
        X = x;
        Y = y;
        Z = 0.0;
    }

    public Point3D(Point3D point)
    {
        this = point;
    }
}
```

Если в типе объявляется поле-структура, все элементы структуры получат значения по умолчанию. Аналогичная ситуация будет при объявлении локальной переменной-структуры и вызове конструктора структуры без параметров[1]. Без вызова конструктора поля переменной-структуры не инициализированы.

```
// поля p1 не инициализированы, их надо установить до использования
Point3D p1;
// поля p2 инициализированы значениями 0.0
Point3D p2 = new Point3D();
// поля p3 инициализированы значениями 2.0, 3.0, 0.0
Point3D p3 = new Point3D(2.0, 3.0);
```

Напомним, что переменные структур размещаются в стеке приложения. Структурные переменные можно присваивать друг другу, при этом выполняется копирование данных структуры на уровне полей. Все структуры наследуются от класса `System.ValueType`. Этот класс переопределяет некоторые методы класса `System.Object`. В частности, переопределяется метод `Equals()` для обеспечения сравнения объектов путем сравнения их полей.

1.16. ПЕРЕЧИСЛЕНИЯ

Перечисление – это тип, содержащий в качестве элементов именованные целочисленные константы. Рассмотрим синтаксис определения перечисления:

```
<модификаторы> enum <имя перечисления> [: <тип перечисления>]
{
    <элемент перечисления 1> [= <значение элемента>],
    . . .
    <элемент перечисления N> [= <значение элемента>]
}
```

[1] В отличие от классов, в структуре конструктор без параметров присутствует даже при объявлении пользовательского конструктора.

Перечисление может предваряться модификатором доступа. Если задан тип перечисления, то он определяет тип каждого элемента перечисления. Типами перечислений могут быть только целочисленные типы, включая `char`. По умолчанию применяется тип `int`. Для элементов перечисления область видимости указать нельзя. Значением элемента перечисления должна быть целочисленная константа. Если значение не указано, элемент будет на единицу большее предыдущего элемента (первый элемент принимает значение `0`). Заданные значения элементов перечисления могут повторяться.

Приведем примеры перечислений:

```
enum Seasons { Winter, Spring, Summer, Autumn }

public enum ErrorCodes : byte
{
    First = 1,
    Fourth = 4
}
```

После описания перечисления можно объявить переменную соответствующего типа:

```
Seasons s = Seasons.Spring;
Console.WriteLine(s); // выводит на печать Spring
```

При помощи явного преобразования типов переменной перечисления можно присвоить значение, которое в перечислении не описано:

```
Seasons s = (Seasons)30;
```

Структура `System.Enum` является базовой для всех перечислений. Табл. 2 содержит описание некоторых методов структуры `System.Enum`.

Таблица 2

Некоторые методы `System.Enum`

Имя метода	Категория	Описание
`GetName()`	статический	Возвращает строку с именем элемента для указанного типа и значения перечисления
`GetNames()`	статический	Возвращает массив строк с именами элементов для указанного типа перечисления
`GetUnderlyingType()`	статический	Возвращает тип перечисления
`GetValues()`	статический	Возвращает массив значений элементов для указанного типа перечисления
`HasFlag()`	экземплярный	Возвращает `true`, если перечисление содержит заданные флаги (т.е. набор значений)
`IsDefined()`	статический	Возвращает `true`, если указанный элемент содержится в заданном типе перечисления
`Parse()`	статический	Конвертирует строку с именем элемента в переменную перечисления
`TryParse<Enum>()`	статический	Делает попытку конвертирования строки в переменную перечисления

1.17. ИНТЕРФЕЙСЫ

В языке C# запрещено множественное наследование классов. Тем не менее, существует концепция, позволяющая имитировать множественное наследование. Эта концепция *интерфейсов*. Интерфейс представляет собой *набор объявлений* свойств, индексаторов, методов и событий. Класс или структура могут *реализовывать* интерфейс. В этом случае они берут на себя обязанность предоставить полную реализацию элементов интерфейса (хотя бы пустыми методами).

Для объявления интерфейса используется ключевое слово `interface`. Интерфейс содержит только заголовки методов, свойств и событий. Для свойства указываются только ключевые слова `get` и (или) `set`. При объявлении элементов интерфейса не могут использоваться следующие модификаторы: `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, `static`. Считается, что все элементы интерфейса имеют `public`-уровень доступа:

```
public interface IFlyable
{
    void Fly();                     // метод
    double Speed { get; set; }      // свойство
}
```

Чтобы указать, что тип реализует некий интерфейс, используется синтаксис *<имя типа> : <имя интерфейса>* при записи заголовка типа. Если класс является производным от некоторого базового класса, то имя базового класса указывается перед именем реализуемого интерфейса.

Элементы интерфейса допускают явную и неявную реализацию. При *неявной реализации* тип должен содержать открытые экземплярные элементы, имена и сигнатура которых соответствуют элементам интерфейса. При *явной реализации* элемент типа называется по форме *<имя интерфейса>.<имя элемента>*, а указание любых модификаторов для элемента при этом запрещается.

```
public class Falcon : IFlyable
{
    // неявная реализация интерфейса IFlyable
    public void Fly() { Console.WriteLine("Falcon flies"); }
    public double Speed { get; set; }
}

public class Eagle : IFlyable
{
    // обычный метод
    public void PrintType() { Console.WriteLine("Eagle"); }

    // явная реализация интерфейса IFlyable
    void IFlyable.Fly() { Console.WriteLine("Eagle flies"); }
    double IFlyable.Speed { get; set; }
}
```

Если тип реализует некоторые элементы интерфейса явно, то такие элементы будут недоступны через переменную типа. Допустимо объявить переменную интерфейса, которая может содержать значение любого типа, реализующего интерфейс (для структур будет выполнена операция упаковки). Через переменную интерфейса можно вызывать только элементы интерфейса.

```
Eagle eagle = new Eagle(); // обычное создание объекта
eagle.PrintType();         // у объекта доступен только этот метод
IFlyable x = eagle;        // переменная интерфейса
x.Fly();                   // получили доступ к элементам интерфейса
x.Speed = 60;
```

Все неявно реализуемые элементы интерфейса по умолчанию помечаются в классе как `sealed`. А значит, наследование классов не ведёт к прямому наследованию реализаций:

```
public interface ISimple
{
    void M();
}

public class Base : ISimple
{
    public void M() { Console.Write("Base.M()"); }
}

public class Descendant : Base
{
    public void M() { Console.Write("Descendant.M()"); }
}

Base x = new Base();
Descendant y = new Descendant();
ISimple xi = x, yi = y;

x.M();            // печатает "Base.M()"
y.M();            // печатает "Descendant.M()"
xi.M();           // печатает "Base.M()"
yi.M();           // печатает "Base.M()"
```

Чтобы осуществить наследование реализаций, требуется при реализации использовать модификаторы `virtual` и `override`[1]:

```
public class Base : ISimple
{
    public virtual void M() { Console.Write("Base.M()"); }
}
```

[1] При явной реализации использование модификаторов невозможно.

```
public class Descendant : Base
{
    public override void M() { Console.Write("Descendant.M()"); }
}
```

Подобно классам, интерфейсы могут наследоваться от других интерфейсов. При этом наследование интерфейсов может быть множественным. Один класс может реализовывать несколько интерфейсов - имена интерфейсов перечисляются после имени класса через запятую.

Интерфейсы сходны с абстрактными классам, что иногда порождает проблему выбора «интерфейс или абстрактный класс». Табл. 3 содержит сравнение возможностей этих пользовательских типов, для того, чтобы помочь сделать правильный выбор между ними.

Таблица 3

Сравнение абстрактных классов и интерфейсов

Абстрактные классы	Интерфейсы
Не могут быть созданы напрямую, но могут содержать конструктор, который вызывается в классе-наследнике	Не могут содержать конструктор
Абстрактный класс может быть так дополнен элементами, что это не повлияет на его классы-наследники	Если в интерфейс помещаются дополнительные элементы, все классы, которые его реализуют, должны быть дополнены
Может хранить данные в полях	Не может хранить данных
Виртуальные элементы могут содержать базовую реализацию. Допустимы невиртуальные элементы	Все элементы являются виртуальными и не включают реализацию
Класс может наследоваться от единственного абстрактного класса	Класс может реализовывать несколько интерфейсов
Класс-наследник может переопределить только некоторые элементы абстрактного класса	Класс, который реализует интерфейс, должен реализовать все элементы интерфейса
Наследование поддерживается только для классов	Интерфейс может быть реализован структурой

1.18. УНИВЕРСАЛЬНЫЕ ШАБЛОНЫ

Универсальные шаблоны (*generics*) позволяют при разработке пользовательского типа или метода указать в качестве параметра тип, который конкретизируется при использовании. Универсальные шаблоны применимы к классам, структурам, интерфейсам, делегатам и методам.

Универсальные классы и структуры

Поясним необходимость универсальных шаблонов на следующем примере. Пусть разрабатывается класс для представления структуры данных «стек». Чтобы не создавать отдельные версии стека для хранения данных определённых типов, программист выбирает базовый тип `object` как тип элемента:

```
public class Stack
{
    private object[] _items;
    public void Push(object item) { . . . }
    public object Pop() { . . . }
}
```

Класс `Stack` можно использовать для разных типов данных:

```
var stack = new Stack();
stack.Push(new Customer());
Customer c = (Customer)stack.Pop();

var stack2 = new Stack();
stack2.Push(3);
int i = (int)stack2.Pop();
```

Однако универсальность класса `Stack` имеет и отрицательные моменты. При извлечении данных из стека необходимо выполнять приведение типов. Для типов значений (например, `int`) при помещении данных в стек и при извлечении выполняются операции упаковки и распаковки, что отрицательно сказывается на производительности. И, наконец, неверный тип помещаемого в стек элемента может быть выявлен только на этапе выполнения, но не компиляции.

```
var stack = new Stack();    // планируем сделать стек чисел
stack.Push(1);
stack.Push(2);
stack.Push("three");        // вставили не число, а строку
var summa = 0;
for (var i = 0; i < 3; i++)
{
    // компилируется, но на третьей итерации
    // будет сгенерирована исключительная ситуация
    summa += (int)stack.Pop();
}
```

Необходимость устранение описанных недостатков явилась основной причиной появления универсальных шаблонов, представленных в C# 2.0.

Опишем класс `Stack` как универсальный тип. Для этого используется следующий синтаксис: после имени класса в угловых скобках указывается *параметр типа*. Этот параметр может затем использоваться при описании элементов класса (в нашем примере - методов и массива).

```
public class Stack<T>
{
    private T[] _items;
    public void Push(T item) { . . . }
    public T Pop() { . . . }
}
```

Использовать универсальный тип «как есть» в клиентском коде нельзя, так как он является не типом, а, скорее, «чертежом» типа. Для работы со `Stack<T>` необходимо использовать *сконструированный тип* (*constructed type*), указав в угловых скобках аргумент типа. Аргумент-тип может быть любым типом. Можно создать любое количество экземпляров сконструированных типов, и каждый из них может использовать разные аргументы типа.

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

Обратите внимание: при работе с типом `Stack<int>` отпала необходимость в выполнении приведения типов при извлечении элементов из стека. Кроме этого, теперь компилятор отслеживает, чтобы в стек помещались только данные типа `int`. И еще одна особенность: нет необходимости в упаковке и распаковке типа значения, а это приводит к росту производительности.

Подчеркнем некоторые особенности сконструированных типов. Во-первых, сконструированный тип не связан отношением наследования с универсальным типом. Во-вторых, даже если классы `A` и `B` связаны наследованием, сконструированные типы на их основе этой связи лишены. В-третьих, статические поля, описанные в универсальном типе, уникальны для каждого сконструированного типа.

При объявлении универсального шаблона можно использовать несколько параметров-типов. Приведём фрагмент описания класса для хранения пар «ключ-значение» с возможностью доступа к значению по ключу[1]:

```
public class Dictionary<K, V>
{
    public void Add(K key, V value) { . . . }
    public V this[K key] { . . . }
}
```

Сконструированный тип для `Dictionary<K, V>` должен быть основан на двух аргументах-типах:

```
Dictionary<int, Customer> dict = new Dictionary<int, Customer>();
```

В языке C# существует операция `default`, которая возвращает значение по умолчанию для переменной указанного типа. Эта операция может использоваться в тех методах, где возвращаемое значение задано как параметр типа:

```
public class Cache<K, V>
{
    // метод для поиска элемента по ключу
```

[1] И класс `Stack<T>`, и класс `Dictionary<K, V>` рассмотрены только как примеры. В .NET Framework уже имеются полноценные аналоги данных классов.

```
        // если элемент найден, то метод возвращается его
        // иначе метод возвращает значение по умолчанию для V
        public V LookupItem(K key)
        {
            return ContainsKey(key) ? GetValue(key) : default(V);
        }
    }
```

Ограничения на параметры шаблонов

Как правило, универсальные типы не просто хранят данные, но и вызывают методы у объекта, чей тип указан как параметр. Например, в классе `Dictionary<K, V>` метод `Add()` может использовать метод `CompareTo()` для сравнения ключей:

```
public class Dictionary<K, V>
{
    public void Add(K key, V value)
    {
        . . .
        if(key.CompareTo(x) < 0) { . . . }  // ошибка компиляции!
        . . .
    }
}
```

Ошибка компиляции в этом примере возникает по следующей причине. Так как тип параметра `K` может быть любым, то у объекта `key` можно вызывать только методы базового типа `object`. Проблему можно решить, используя приведение типов:

```
public class Dictionary<K, V>
{
    public void Add(K key, V value)
    {
        . . .
        if(((IComparable)key).CompareTo(x) < 0) { . . . }
        . . .
    }
}
```

Недостаток такого подхода – многочисленность операций приведения. К тому же, если у сконструированного типа параметр `K` не поддерживает интерфейс `IComparable`, то при работе программы будет сгенерировано исключение `InvalidCastException`.

C# допускает указание *ограничения* (*constraint*) для каждого параметра универсального типа. Только тип, удовлетворяющий ограничениям, может быть применён для записи сконструированного типа.

Ограничения объявляются с использованием ключевого слова `where`, после которого указывается параметр, двоеточие и список ограничения. Элементом списка ограничения на тип могут являться:

- Ключевое слово `class` (требование, чтобы тип был ссылочным) или ключевое слово `struct` (требование, чтобы тип был типом значения).
- Имя класса (требование, чтобы тип приводился к этому классу).
- Интерфейс или список интерфейсов (требование, чтобы тип реализовывал эти интерфейсы).
- Конструкция `new()` (требование, чтобы у типа был конструктор без параметров).

Порядок элементов в списке ограничений имеет значение. Правильный порядок соответствует порядку в списке, приведенном выше.

В следующем примере используется несколько ограничений на различные параметры универсального типа (предполагается, что интерфейс `IComparable<K>` содержит метод `CompareTo()`):

```
public class EntityTable<K, E>
    where K : IComparable<K>, IPersistable
    where E : Entity, new()
{
    public void Add(K key, E entity) {
        . . .
        if (key.CompareTo(x) < 0) { . . . }
        . . .
    }
}
```

Смысл ограничений, наложенных на параметр `E`: он должен быть приводим к классу `Entity` и иметь `public`-конструктор без параметров.

Ковариантность и контравариантность

Определим понятия ковариантности и контравариантности для сконструированных типов данных. Для этого введём отношение частичного порядка на множестве ссылочных типов:

$$T_1 \le T_2 \iff T_1 \text{ наследуется (прямо или косвенно) от } T_2.$$

Если имеется тип `C<T>`, а также типы T_1 и T_2 ($T_1 \le T_2$), то `C<T>` назовём:

- *ковариантным*, если $C<T_1> \le C<T_2>$;
- *контравариантным*, если $C<T_2> \le C<T_1>$;
- *инвариантным*, если не верно ни первое, ни второе утверждение.

Понятия частичного порядка типов, ковариантности и контравариантности связаны с приведением типов. Тот факт, что тип T_1 «меньше» типа T_2, означает возможность неявного приведения переменной типа T_1 к типу T_2. Как указыва-

лось ранее, массивы ковариантны для ссылочных типов (например, массив строк присваивается массиву объектов).

Универсальные классы и структуры инвариантны, однако, универсальные интерфейсы могут быть описаны как ковариантные или контравариантные относительно некоего параметра-типа. Чтобы указать на ковариантность относительно параметра `T`, следует использовать ключевое слово `out` при описании параметра типа. На контравариантность указывает ключевое слово `in` при описании параметра типа.

```
public interface IOutOnly<out T>
{
   T this[int index] { get; }
}
public interface IInOnly<in T>
{
   void Process(T x);
}
```

Для обеспечения безопасности типов компилятор отслеживает, чтобы ковариантные параметры всегда использовались как типы возвращаемых значений, а контравариантные параметры являлись типами аргументов. Один универсальный интерфейс может, при необходимости, содержать как ковариантные, так и контравариантные параметры.

Универсальные методы

В некоторых случаях достаточно параметризовать не весь пользовательский тип, а только отдельный метод. *Универсальные методы* (*generic methods*) объявляются с использованием параметров-типов в угловых скобках после имени метода[1]. Как и при описании универсальных типов, универсальные методы могут содержать ограничения на параметр-тип.

```
void PushMultiple<T>(Stack<T> stack, params T[] values)
{
    foreach (T value in values)
    {
        stack.Push(value);
    }
}
```

Использование универсального метода `PushMultiple<T>` позволяет работать с любым сконструированным типом на основе `Stack<T>`.

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

[1] Универсальные методы могут заменить перекрытие методов в пользовательском типе, если алгоритмы работы различных версий перекрытых методов не зависят от типов параметров.

В большинстве ситуаций компилятор способен самостоятельно сконструировать тип универсального метода на основе анализа типов фактических параметров. Это позволяет записывать вызов метода без указания типа:

```
var stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);
```

Рассмотрим следующий пример:

```
public class C
{
    public static void M(string a, string b) { . . . }
    public static void M<T>(T a, T b) { . . . }
}

int a = 10;
byte b = 20;
string s = "ABC";
C.M(a, b);                  // C.M<int>(a,b)
C.M(s, s);                  // C.M(s,s)
C.M<string>(s, s);          // C.M<string>(s,s)
```

В первом случае компилятор сконструирует метод `M<T>()` как `M<int>()`, потому что к типу `int` приводится и переменная `a`, и переменная `b`. Второй вызов показывает, что при наличии альтернатив компилятор всегда выбирает версию метода без универсального параметра, если только метод не сконструирован явно.

1.19. ИСПОЛЬЗОВАНИЕ УНИВЕРСАЛЬНЫХ ШАБЛОНОВ

В данном параграфе приведено несколько примеров использования универсальных шаблонов в элементах платформы .NET.

Кортежи

В языках программирования *кортежем* (*tuple*) называется структура данных, содержащая фиксированный набор разнотипных значений. В платформе .NET для создания кортежей доступен набор универсальных классов вида `System.Tuple<>`. Всего имеется восемь универсальных классов `Tuple<>`, которые просто различаются количеством параметров типов.

```
Tuple<string, int> t = new Tuple<string, int>("Hello", 4);
Console.WriteLine("{0} - {1}", t.Item1, t.Item2);
```

Статический класс `System.Tuple` содержит восемь перегруженных версий метода `Create()` для конструирования кортежа с заданным числом элементов:

```
Tuple<string, int> t = Tuple.Create("Hello", 4);
```

Типы, допускающие значение null

Примером универсальных типов являются *типы, допускающие значение* `null` (*nullable types*). Такие типы являются экземплярами структуры `System.Nullable<T>`, где тип `T` должен быть типом значения. Структура `Nullable<T>` имеет специальный флаг `HasValue`, указывающий на наличие значения, и свойство `Value`, содержащее значение. Попытка прочитать `Value` при `HasValue == false` ведет к генерации исключения. Также в `Nullable<T>` определён метод `GetValueOrDefault()`.

Язык C# предлагает компактную форму объявления типа, допускающего значение `null`. Для этого после имени типа указывается символ `?`.

```
int? x = 123;
int? y = null;
if (x.HasValue) Console.WriteLine(x.Value);
if (y.HasValue) Console.WriteLine(y.Value);
int k = x.GetValueOrDefault();
int p = y.GetValueOrDefault(10);
```

Для типов значений определено неявное преобразование к соответствующему типу, допускающему значение `null`. Если для типа `S` возможно приведение к типу `T`, то такая возможность имеется и для типов `S?` и `T?`. Также возможно неявное приведение типа `S` к типу `T?` и явное приведение `S?` к `T`. В последнем случае возможна генерации исключительной ситуации – если значение типа `S?` не определено.

```
int x = 10;
int? z = x;              // неявное приведение int к int?
double? w = z;           // неявное приведение int? к double?
double y = (double)z;    // явное приведение int? к double
```

Хотя для структуры `Nullable<T>` не определены арифметические операции и операции сравнения, компилятор способен «заимствовать» нужную операцию у соответствующего типа значения. При этом действуют следующие правила:

- Арифметические операции возвращают значение `null`, если хотя бы один из операндов равен `null`.
- Операции сравнения, кроме `==` и `!=`, возвращают значение `false`, если хотя бы один из операндов равен `null`.
- Операции равенства `==` и `!=` считают две переменные, равные `null`, равными между собой.
- Если в операциях `&` и `|` участвуют операнды типа `bool?`, то `null | true` равняется `true`, а `null & false` равняется `false`.

```
int? x = 3, y = 5, z = null;
Console.WriteLine(x + y);        // 8
Console.WriteLine(x + z);        // null
Console.WriteLine(x < y);        // true
```

```
Console.WriteLine(x < z);         // false
Console.WriteLine(null == z);     // true
```

С типами, допускающими значение null, связана операция ??. Результатом выражения a ?? b является a, если a содержит некое значение, и b – в противном случае. Таким образом, b – это значение, которое следует использовать, если a не определено. Тип выражения a ?? b определяется типом операнда b.

```
int? x = GetNullableInt();
int? y = GetNullableInt();
int? z = x ?? y;
int i = z ?? -1;
```

Операцию ?? можно применить и для ссылочных типов:

```
string s = GetStringValue();
Console.WriteLine(s ?? "Unspecified");
```

В этом фрагменте кода на консоль выводится значение строки s, или "Unspecified", если s == null.

Прочие примеры универсальных шаблонов

Структура System.ArraySegment<T> является «обёрткой» над массивом, определяющей интервал элементов массива. Для одного массива можно создать несколько объектов ArraySegment<T>, которые могут задавать даже перекрывающиеся интервалы. Работать с «обёрнутым» массивом можно, используя свойство ArraySegment<T>.Array.

```
int[] a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
var firstSeg = new ArraySegment<int>(a, 2, 6);     // первый сегмент
var secondSeg = new ArraySegment<int>(a, 4, 3);    // второй сегмент
firstSeg.Array[3] = 10;      // изменяем четвёртый элемент массива a
```

Класс System.Lazy<T> служит для поддержки отложенной инициализации объектов. Данный класс содержит булево свойство для чтения IsValueCreated и свойство для чтения Value типа T. Использование Lazy<T> позволяет задержать создание объекта до первого обращения к свойству Value. Для создания объекта используется либо конструктор без параметров типа T, либо функция, передаваемая конструктору Lazy<T>.

```
Lazy<Student> lazy = new Lazy<Student>();
Console.WriteLine(lazy.IsValueCreated);           // false
Student s = lazy.Value;
Console.WriteLine(lazy.IsValueCreated);           // true
```

1.20. ДЕЛЕГАТЫ

Делегат – это пользовательский тип, который инкапсулирует метод. В C# делегат объявляется с использованием ключевого слова `delegate`. При этом указывается имя делегата, тип и сигнатура инкапсулируемого метода:

```
public delegate double Function(double x);
public delegate void Subroutine(int i);
```

Делегат – самостоятельный пользовательский тип, он может быть как вложен в другой пользовательский тип (класс, структуру), так и объявлен отдельно. Так как делегат – это тип, то нельзя объявить в одной области видимости два делегата с одинаковыми именами, но разной сигнатурой.

После объявления делегата можно объявить переменные этого типа:

```
Function F;
Subroutine S;
```

Переменные делегата инициализируются конкретными методами при использовании *конструктора делегата* с одним параметром – именем метода (или именем другого делегата). Если делегат инициализируется статическим методом, требуется указать имя класса и имя метода, для инициализации экземплярным методом указывается объект и имя метода. При этом метод должен обладать подходящей сигнатурой:

```
F = new Function(ClassName.SomeStaticFunction);
S = new Subroutine(obj.SomeInstanceMethod);
```

Для инициализации делегата можно использовать упрощенный синтаксис – достаточно указать имя метода без применения `new()`.

```
F = ClassName.SomeStaticFunction;
S = obj.SomeInstanceMethod;
```

После того как делегат инициализирован, инкапсулированный в нем метод вызывается с указанием аргументов метода непосредственно после имени переменной-делегата: `F(0.5);`.

Приведем пример работы с делегатами. Создадим класс, содержащий метод расширения для трансформации массива целых чисел:

```
public static class ArrayHelper
{
    public static int[] Transform(this int[] data, Transformer f)
    {
        var result = new int[data.Length];
        for (int i = 0; i < data.Length; i++)
            result[i] = f(data[i]);
        return result;
    }
}
```

Тип `Transformer` является делегатом, который определяет способ преобразования отдельного числа. Он описан следующим образом:

```
public delegate int Transformer(int x);
```

Создадим класс, который использует `ArrayHelper` и `Transformer`:

```
public class MainClass
{
    public static int TimesTwo(int i) { return i * 2; }

    public int AddFive(int i) { return i + 5; }

    public static void Main()
    {
        int[] a = { 1, 2, 3 };
        Transformer t = TimesTwo;
        a = a.Transform(t);
        var c = new MainClass();
        a = a.Transform(c.AddFive);
    }
}
```

Необходимо заметить, что делегаты обладают *контравариантностью* относительно типов параметров и *ковариантностью* относительно типа результата. Это значит, что делегат может инкапсулировать метод, у которого входные параметры имеют более общий тип, а тип результата более специфичен, чем описано в делегате.

```
// два класса, связанных наследованием
public class Person { . . . }
public class Student : Person { . . . }

// делегат для обработки
public delegate Person Register(Person p);

// вспомогательный класс с методом обработки
public class C
{
    public static Student M(object o) { return new Student(); }
}

// присваивание возможно благодаря ко- и контравариантности
Register f = C.M;
```

Делегаты могут быть объявлены как универсальные типы. При этом допустимо использование ограничений на параметр типа, а также указания на ковариантность и контравариантность параметров типа.

```
public delegate TResult Transformer<in T, out TResult>(T x);

public static class ArrayHelper
{
    public static TResult[] Transform<T, TResult>(this T[] data,
                                        Transformer<T, TResult> f)
    {
        var result = new TResult[data.Length];
        for (int i = 0; i < data.Length; i++)
        {
            result[i] = f(data[i]);
        }
        return result;
    }
}
```

Ключевой особенностью делегатов является то, что они могут инкапсулировать не один метод, а несколько. Подобные объекты называются *групповыми делегатами*. При вызове группового делегата срабатывает вся цепочка инкапсулированных в нем методов.

Групповой делегат объявляется таким же образом, как и обычный. Затем создаётся несколько объектов делегата, и все они связываются с некоторыми методами. После этого используются операция + для объединения делегатов в один групповой делегат. Если требуется удалить метод из цепочки группового делегата, используются операция –. Если из цепочки удаляют последний метод, результатом будет значение `null`.

Приведём пример использования группового делегата типа `Transformer`:

```
int[] a = { 1, 2, 3 };
var c = new MainClass();
Transformer<int, int> x = TimesTwo, y = c.AddFive, z;
z = x + y;          // z – групповой делегат, содержит 2 функции
a.Transform(z);
```

Любой пользовательский делегат можно рассматривать как наследник класса `System.MulticastDelegate`, который, в свою очередь, наследуется от класса `System.Delegate`. Рассмотрим элементы `System.Delegate`:

- Перегруженный статический метод `CreateDelegate()` позволяет создавать делегаты на основе информации о типе и методе.
- Метод `Clone()` создаёт копию объекта-делегата.
- Статический метод `Combine()` объединяет в групповой делегат два объекта-делегата или массив таких объектов.
- Статические методы `Remove()` и `RemoveAll()` удаляют указанный объект-делегат из группового делегата[1].

[1] Делегаты относятся к *неизменяемым типам*. Поэтому методы `Combine()` и `Remove()` возвращают новые объекты-делегаты.

- Метод `GetInvocationList()` возвращает массив инкапсулированных объектов-делегатов.
- Свойство `Method` позволяет получить информацию о методе, инкапсулированном объектом-делегатом.
- Свойство `Target` содержит ссылку на экземпляр объекта, связанного с инкапсулированным методом (или `null` для статических методов).

В пространстве имен `System` объявлено несколько полезных универсальных делегатов. Имеются делегаты для представления функций и действий, содержащих от нуля до шестнадцати аргументов - `Func<…>` и `Action<…>`, делегаты для функций конвертирования `Converter<in TInput, out TOutput>`, сравнения `Comparison<in T>` и предиката `Predicate<in T>`.

1.21. АНОНИМНЫЕ МЕТОДЫ И ЛЯМБДА-ВЫРАЖЕНИЯ

Назначение *анонимных методов* (*anonymous methods*) заключается в том, чтобы сократить объём кода, который должен писать разработчик при использовании делегатов. Если рассмотреть примеры предыдущего параграфа, то очевидно, что даже для объектов-делегатов, содержащих минимум действий, необходимо создавать метод (а, возможно, и отдельный класс) и инкапсулировать этот метод в делегат. При применении анонимных методов формируется безымянный блок кода, который назначается объекту-делегату.

Синтаксис объявления анонимного метода включает ключевое слово `delegate` и список формальных параметров. Ковариантность и контравариантность делегатов работает и в случае применения анонимных методов. Дополнительным правилом является возможность описать анонимный метод без параметров, если параметры не используются в теле метода, а делегат не имеет `out`-параметров.

Модифицируем фрагмент кода из предыдущего параграфа, использовав анонимные методы:

```
int[] a = { 1, 2, 3 };
Transformer<int, int> t = delegate(int x) { return x * x; };
a.Transform(t);
```

Лямбда-выражения и *лямбда-операторы* – это альтернативный синтаксис записи анонимных методов. Начнём с рассмотрения лямбда-операторов. Пусть имеется анонимный метод:

```
Func<int, bool> f = delegate(int x)
                    {
                        int y = x - 100;
                        return y > 0;
                    };
```

При использовании лямбда-операторов список параметров отделяется от тела оператора символами `=>`, а ключевое слово `delegate` не указывается:

```
Func<int, bool> f = (int x) => { int y = x - 100; return y > 0; };
```

Более того, так как мы уже фактически указали тип аргумента лямбда-оператора слева при объявлении `f`, то его можно не указывать справа. В случае если у нас один аргумент, можно опустить обрамляющие его скобки [1]:

```
Func<int, bool> f = x => { int y = x - 100; return y > 0; };
```

Когда лямбда-оператор содержит в своем теле единственный оператор `return`, он может быть записан в компактной форме *лямбда-выражения*:

```
Func<int, bool> f_2 = x => x > 0;
```

Заметим, что для переменной, в которую помещается ламбда-оператор или лямбда-выражение, требуется явно указывать тип – `var` использовать нельзя.

Рассмотрим пример, демонстрирующий применение лямбда-выражений. Используем метод `Transform()` из предыдущего параграфа, чтобы получить из числового массива массив булевых значений, а из него - массив строк. Обратите внимание на компактность получаемого кода.

```
int[] a = { 1, 2, 3 };
string[] s = a.Transform(x => x > 2).Transform(y => y.ToString());
// s содержит "False", "False", "True"
```

Анонимные методы и лямбда-операторы способны захватывать внешний контекст вычисления. Если при описании тела анонимного метода применялась внешняя переменная, вызов метода будет использовать *текущее* значение переменной. Захват внешнего контекста иначе называют *замыканием* (*closure*).

```
int[] a = { 1, 2, 3 };
int external = 0;            // внешняя переменная
Transformer<int, int> t = x => x + external;        // замыкание
external = 10;               // изменили переменную после описания t
int[] b = a.Transform(t);    // прибавляет 10 к каждому элементу
```

Эффектный приём использования замыканий – функции, реализующие *мемоизацию* (*memoization*). Мемоизация – это кэширование результатов вычислений. В следующем примере описан метод расширения для строк, который возвращает функцию подсчёта частоты встречаемости символа в строке.

```
public static Func<char, int> FrequencyFunc(this string text)
{
    int[] freq = new int[char.MaxValue];
    foreach (char c in text)
    {
        freq[c]++;
    }
```

[1] Если аргументов несколько, скобки нужно указывать. Также пустые скобки указываются, когда лямбда-оператор не имеет входных аргументов.

```
    return ch => freq[ch];
}

// использование частотной функции
var f = "There is no spoon".FrequencyFunc();
Console.WriteLine(f('o'));
```

1.22. СОБЫТИЯ

События – способ описания связи одного объекта с другими по действиям. Работу с событиями можно условно разделить на три этапа:

- объявление события (*publishing*);
- регистрация получателя события (*subscribing*);
- генерация события (*raising*).

Событие можно объявить в пределах класса, структуры или интерфейса. Базовый синтаксис объявления события следующий:

```
<модификаторы> event <тип делегата> <имя события>;
```

Ключевое слово `event` указывает на объявление события. При объявлении события требуется указать делегат, описывающий метод обработки события. Этот делегат должен иметь тип возвращаемого значения `void`.

Фактически, события являются полями типа делегатов. При объявлении события компилятор добавляет в класс `private`-поле с именем *<имя события>* и типом *<тип делегата>*. Кроме этого, для обслуживания события компилятор создаёт два метода `add_`*Name*`()` и `remove_`*Name*`()`, где *Name* – имя события. Эти методы содержат код, добавляющий и удаляющий обработчик события в цепочку группового делегата, связанного с событием.

Если программиста по каким-либо причинам не устраивает автоматическая генерация методов `add_`*Name*`()` и `remove_`*Name*`()`, он может описать собственную реализацию данных методов. Для этого при объявлении события указывается блок, содержащий секции `add` и `remove`:

```
<модификаторы> event <тип делегата> <имя события>
{
    add {<блок кода>}
    remove {<блок кода>}
};
```

В блоке добавления и удаления обработчиков обычно размещается код, добавляющий или удаляющий метод в цепочку группового делегата. Поле-делегат в этом случае должно быть явно объявлено в классе.

Для генерации события в требуемом месте кода помещается вызов в формате *<имя события>*`(`*<фактические аргументы>*`)`. Предварительно обычно прове-

ряют, назначен ли обработчик события. Генерация события может происходить только в том же классе, в котором событие объявлено[1].

Приведём пример класса, содержащего объявление и генерацию события. Данный класс будет включать метод с целым параметром, устанавливающий значение поля класса. Если значение параметра отрицательно, генерируется событие, определенное в классе:

```
public delegate void Handler(int val);

public class ExampleClass
{
    private int _field;

    public int Field
    {
        get { return _field; }
        set
        {
            _field = value;
            if (value < 0)
            {
                // проверка нужна, чтобы предотвратить генерацию
                // исключительной ситуации
                if (NegativeValueSet != null)
                {
                    NegativeValueSet(value);
                }
            }
        }
    }

    public event Handler NegativeValueSet;
}
```

Рассмотрим этап регистрации получателя события. Чтобы отреагировать на событие, его надо ассоциировать с *обработчиком события*. Обработчиком может быть метод, приводимый к типу события (делегату). В качестве обработчика может выступать анонимный метод или лямбда-оператор. Назначение и удаление обработчиков события выполняется при помощи операторов `+=` и `-=`.

Используем класс `ExampleClass` и продемонстрируем назначение и удаление обработчиков событий:

```
public class MainClass
{
    public static void Reaction(int i)
```

[1] Поведение, аналогичное событиям, можно получить, используя открытие поля делегатов. Ключевое слово `event` заставляет компилятор проверять, что описание и генерация события происходят в одном классе, и запрещает для события все операции, кроме `+=` и `-=`.

```
    {
        Console.WriteLine("Negative value = {0}", i);
    }

    public static void Main()
    {
        var c = new ExampleClass();
        c.Field = -10; // нет обработчиков, нет и реакции на событие

        // назначаем обработчик
        c.NegativeValueSet += Reaction;
        c.Field = -20;    // вывод: "Negative value = -20"

        // назначаем еще один обработчик в виде лямбда-оператора
        c.NegativeValueSet += i => Console.WriteLine(i);
        c.Field = -30;    // вывод: "Negative value = -30" и "-30"

        // удаляем первый обработчик
        c.NegativeValueSet -= Reaction;
    }
}
```

Платформа .NET предлагает средства стандартизации работы с событиями. В частности, для типов событий зарезервированы следующие делегаты:

```
public delegate void EventHandler(object sender, EventArgs e);

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

Как видим, данные делегаты предполагают, что первым параметром будет выступать объект, в котором событие было сгенерировано. Второй параметр используется для передачи информации события. Это либо класс `EventArgs`, либо наследник этого класса с необходимыми полями.

Сама генерация события обычно выносится в отдельный виртуальный метод класса. В этом методе проверяется, был ли установлен обработчик события. Также можно создать копию события перед обработкой (это актуально в многопоточных приложениях)

Внесём изменения в код класса `ExampleClass`, чтобы работа с событиями соответствовала стандартам:

```
public class MyEventArg : EventArgs
{
    public int NewValue { get; private set; }

    public MyEventArg(int newValue)
    {
        NewValue = newValue;
    }
```

```
}

public class ExampleClass
{
    private int _field;

    public int Field
    {
        get { return _field; }
        set
        {
            _field = value;
            if (value < 0)
            {
                OnNegativeValueSet(value);
            }
        }
    }

    protected virtual void OnNegativeValueSet(int value)
    {
        EventHandler<MyEventArg> local = NegativeValueSet;
        if (local != null)
        {
            local(this, new MyEventArg(value));
        }
    }

    public event EventHandler<MyEventArg> NegativeValueSet;
}
```

Создадим несколько полезных классов для упрощения работы с событиями. Очень часто для передачи информации события достаточно класса с единственным свойством. В этом случае можно использовать универсальный класс `EventArgs<T>`.

```
public class EventArgs<T> : EventArgs
{
    public T EventInfo { get; private set; }

    public EventArgs(T eventInfo)
    {
        EventInfo = eventInfo;
    }
}
```

Вот пример использования `EventArgs<T>` при описании события:

```
public event EventHandler<EventArgs<int>> NegativeValueSet;
```

В класс `EventHelper` поместим два метода расширения, упрощающих генерацию событий:

```
public static class EventHelper
{
    public static void Raise<T>(
                            this EventHandler<EventArgs<T>> handler,
                            T args, object sender = null)
    {
        EventHandler<EventArgs<T>> e = handler;
        if (e != null)
        {
            e(sender, new EventArgs<T>(args));
        }
    }

    public static void Raise(this EventHandler handler,
                             EventArgs args, object sender = null)
    {
        EventHandler e = handler;
        if (e != null)
        {
            e(sender, args);
        }
    }
}
```

Вот пример использования метода расширения из класса `EventHelper`:

```
local.Raise(value, this);
```

1.23. ПЕРЕГРУЗКА ОПЕРАЦИЙ

Язык C# позволяет организовать для объектов пользовательского класса или структуры *перегрузку операций*. Могут быть перегружены унарные операции `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false` и бинарные операции `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`. При перегрузке бинарной операции автоматически перегружается соответствующая операция с присваиванием (например, при перегрузке операции `+` перегрузится и операция `+=`). Некоторые операции могут быть перегружены только парами: `==` и `!=`, `>` и `<`, `>=` и `<=`, `true` и `false`.

Для перегрузки операций используется специальный статический метод, имя которого образовано из ключевого слова `operator` и знака операции. Количество формальных параметров метода зависит от типа операции: унарная операция требует одного параметра, бинарная – двух. Метод обязательно должен иметь модификатор доступа `public`.

Рассмотрим перегрузку операций на примере. Определим класс для представления точек на плоскости с перегруженной операцией сложения:

```
public class Point
```

```
{
    public double X { get; set; }
    public double Y { get; set; }

    public override string ToString()
    {
        return String.Format("X = {0} Y = {1}", X, Y);
    }

    public static Point operator +(Point a, Point b)
    {
        return new Point { X = a.X + b.X, Y = a.Y + b.Y };
    }
}
```

Для объектов класса `Point` возможно использование следующего кода:

```
var p1 = new Point { X = 10, Y = 20 };
var p2 = new Point { X = -5, Y = 10 };
Console.WriteLine(p1);          // печатает "X = 10 Y = 20"
Console.WriteLine(p2);          // печатает "X = -5 Y = 10"
Console.WriteLine(p1 + p2);     // печатает "X = 5 Y = 30"
```

Параметры метода перегрузки должны быть параметрами, передаваемыми по значению. Тип формальных параметров и тип возвращаемого значения метода перегрузки обычно совпадает с описываемым типом, хотя это и не обязательное условие. Более того, класс или структура могут содержать версии одной операции с разным типом формальных параметров. Однако не допускается существование двух версий перегрузки операции, различающихся только типом возвращаемого значения. Также класс не может содержать перегруженной операции, у которой ни один из формальных параметров не имеет типа класса.

Дополним класс `Point`, позволив прибавлять к точке вещественное число:

```
public class Point
{
    . . .
    public static Point operator +(Point a, double delta)
    {
        return new Point { X = a.X + delta, Y = a.Y + delta };
    }
}
```

Любой класс или структура могут перегрузить операции `true` и `false`. Операции перегружаются парой, тип возвращаемого значения операций – булевский. Если выполнена подобная перегрузка, объекты могут использоваться как условия в операторах условного перехода и циклов.

Рассмотрим следующий пример. Пусть в классе `Point` перегружены операции `true` и `false`:

```
public class Point
{
    . . .
    public static bool operator true(Point a)
    {
        return (a.X > 0) || (a.Y > 0);
    }

    public static bool operator false(Point a)
    {
        // этот метод должен возвращать «true»,
        // если семантика объекта соответствует false
        return (a.X == 0) && (a.Y == 0);
    }
}
```

Теперь возможно написать такой код (обратите внимание на оператор `if`):

```
var p1 = new Point { X = 10, Y = 20 };
var p2 = new Point();
if (p2)
    Console.WriteLine("Point is positive");
else
    Console.WriteLine("Point has non-positive data");
```

Если класс или структура `T` перегружают `true`, `false` и операции `&` или `|`, то становится возможным вычисление булевых выражений по сокращенной схеме. В этом случае:

- выражение `x && y` транслируется в `T.false(x) ? x : T.&(x,y)`;
- выражение `x || y` транслируется в `T.true(x) ? x : T.|(x,y)`;

Любой класс или структура могут перегрузить операции для неявного и явного приведения типов. При этом используется следующий синтаксис:

```
public static implicit operator <целевой тип>(<приводимый тип> <имя>)

public static explicit operator <целевой тип>(<приводимый тип> <имя>)
```

Ключевое слово `implicit` используется при перегрузке неявного приведения типов, а ключевое слово `explicit` – при перегрузке операции явного приведения. Либо *<целевой тип>*, либо *<приводимый тип>* должны совпадать с типом того класса, где выполняется перегрузка операций.

Поместим две перегруженных операции приведения в класс `Point`:

```
public class Point
{
    . . .
    public static implicit operator Point(double x)
    {
```

```
            return new Point { X = x };
        }

        public static explicit operator double(Point a)
        {
            return Math.Sqrt(a.X * a.X + a.Y * a.Y);
        }
    }
```

Вот пример кода, использующего преобразование типов:

```
var p1 = new Point { X = 3, Y = 4 };
double y = 10;
double x = (double)p1;          // явное приведение типов
p1 = y;                         // неявное приведение типов
```

Хотелось бы отметить, что разработчик должен ответственно подходить к вопросу перегрузки операций. Методы перегрузки в совокупности с методами приведения типов могут породить код, который трудно сопровождать.

1.24. АНОНИМНЫЕ ТИПЫ

Анонимные типы (*anonymous types*), представленные в C# 3.0, позволяют создавать новый тип, не декларируя его заранее, а описывая непосредственно при создании переменной. Мотивом для введения анонимных типов в спецификацию языка послужила работа с коллекциями в технологии LINQ. При обработке коллекций тип элементов результата может отличаться от типа элементов исходной коллекции. Например, одна обработка набора объектов `Student` может привести к коллекции, содержащей имя студента и возраст. Другая обработка – к коллекции с именем и номером группы. В таких ситуациях в старых версиях C# нужно или заранее создать необходимое количество вспомогательных типов, или воспользоваться неким «мегатипом», содержащим все возможные поля результатов. Анонимные типы предлагают более элегантное решение.

Объявление анонимного типа использует синтаксис инициализатора объектов, предварённый ключевым словом `new`. Тип полей не указывается, а выводится из начального значения.

```
var anonymous = new {a = 3, b = true, c = "string data"};
```

Если при объявлении анонимного типа в качестве значений полей применяются не константы, а элементы известного пользовательского типа или локальные переменные, то имя поля анонимного типа можно не указывать. Будет использовано имя инициализатора.

```
int x = 10;
// у анонимного типа будут поля x (со значением 10), b и c
var anonymous = new {x, b = true, c = "string data"};
```

Анонимный тип следует рассматривать как класс, состоящий из полей только для чтения. Кроме полей, других элементов анонимный тип содержать не может. Два анонимных типа считаются эквивалентными, если у них полностью (вплоть до порядка) совпадают поля (имена и типы).

```
var anonymous = new { a = 3, b = true, c = "string data" };
var anotherAnonymous = new { a = 1, b = false, c = "data" };
anonymous = anotherAnonymous;       // допустимое присваивание
```

Хотя анонимный тип задумывался как хранилище данных (концепция анонимных типов близка к концепции кортежей), действия в анонимный тип можно поместить, используя делегаты:

```
Action<int> m = x => Console.WriteLine(x);
var anonymous = new { data = 1, method = m };
anonymous.method(3);
```

1.25. ПРОСТРАНСТВА ИМЁН

Пространства имён служат для логической группировки пользовательских типов. Применение пространств имён обосновано в крупных программных проектах для снижения риска конфликта имён и улучшения структуры библиотек кода.

Синтаксис описания пространства имён следующий:

```
namespace <имя пространства имён>
{
    [<компоненты пространства имён>]
}
```

Компонентами пространства имён могут быть классы, интерфейсы, делегаты, перечисления, структуры и другие пространства имён. Само пространство имён может быть вложено только в другое пространство имён.

Если в разных местах программы определено несколько пространств имён с одинаковыми именами, компилятор собирает компоненты из этих пространств в общее пространство имён. Для этого необходимо, чтобы одноименные пространства имён находились на одном уровне вложенности в иерархии пространств имён.

Для доступа к компонентам пространства имён используется синтаксис *<имя пространства имён>.<имя компонента>*.

Для использования в программе некоего пространства имён служит *директива* `using`. Её синтаксис следующий:

```
using <имя пространства имён>;
using [<имя псевдонима> =] <имя пространства>[.<имя типа>];
```

Импортирование пространства имён позволяет сократить полные имена классов. *Псевдоним*, используемый при импортировании, это обычно короткий

идентификатор для ссылки на пространство имён (или элемент из пространства имён) в тексте программы. Импортировать можно пространства имён как из текущего проекта, так и из подключенных к проекту сборок.

Рассмотрим некоторые тонкости при работе с пространствами имён. Предположим, что создаётся проект, использующий внешние сборки `A.dll` и `B.dll`. Пусть сборка `A` содержит пространство имён `NS` с классом `C`, и сборка `B` содержит такое же пространство и класс. Как поступить для доступа к различным классам `C` в коде? Эту проблему решает операция `::` и директива `extern alias`. Во-первых, сборкам `A.dll` и `B.dll` нужно назначит текстовые псевдонимы. В Visual Studio псевдоним для подключённой сборки можно установить в свойствах сборки. При использовании компилятора командной строки псевдоним указывается с опцией ссылки на сборку.

```
csc.exe program.cs /r:A=A.dll /r:B=B.dll
```

Затем с элементами сборок можно работать следующим образом:

```
extern alias A;
extern alias B;

public class Program
{
    private static void Main()
    {
        var a = new A::NS.C();
        var b = new B::NS.C();
    }
}
```

Существует предопределённый псевдоним с именем `global` для всех стандартных сборок платформы .NET.

1.26. ГЕНЕРАЦИЯ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Опишем возможности по генерации и обработке исключительных ситуаций в языке C#. Для генерации исключительной ситуации используется оператор `throw`:

```
throw <объект класса исключительной ситуации>;
```

Объект, указанный после `throw`, должен быть объектом класса исключительной ситуации. В C# классами исключительных ситуаций являются класс `System.Exception` и все его наследники. В некоторых языках для .NET можно (хотя и не рекомендуется) генерировать исключения, не являющиеся производными от `Exception`. В таком случае CLR автоматически поместит объект исключения в оболочку класса `RuntimeWrappedException`, который наследуется от `Exception`.

Класс `Exception` – это базовый класс для представления исключительных ситуаций. Основными элементами этого класса являются:

- свойство только для чтения `Message`, содержащее строку с описанием ошибки;
- перегруженный конструктор с одним параметром-строкой, записываемым в свойство `Message`;
- строковое свойство `StackTrace`, описывающее содержимое стека вызова, в которой первым отображается самый последний вызов метода.
- свойство `InnerException` – объект `Exception`, описывающий ошибку, вызывающую текущее исключение.
- коллекция-словарь `Data` с дополнительной информацией об ошибке.

В пространстве имён `System` содержится несколько классов для описания наиболее распространённых исключений:

- `ArgumentException` - генерируется, когда методу передаётся недопустимый аргумент.
- `ArgumentNullException` (наследник `ArgumentException`) - генерируется, когда методу передаётся аргумент, равный `null`.
- `ArgumentOutOfRangeException` (наследник `ArgumentException`) - генерируется, когда методу передаётся аргумент, выходящий за допустимый диапазон.
- `InvalidOperationException` - сигнализирует о том, что состояние объекта препятствует выполнению метода (пример: запись в файл, который открыт только для чтения).
- `NotSupportedException` - сигнализирует о том, что функциональная возможность не поддерживается.
- `NotImplementedException` - сигнализирует о том, что функциональная возможность не реализована.
- `ObjectDisposedException` - генерируется, когда методу вызывается у удалённого из памяти объекта.

Разработчик может создать собственный класс для представления информации об исключительной ситуации. Единственным условием для этого класса в C# является прямое или косвенное наследование от класса `Exception`.

Рассмотрим пример программы с генерацией исключительной ситуации:

```
using System;

public class ExampleClass
{
    private int _field;

    public int Field
    {
        get { return _field; }
        set
```

```
        {
            if (value < 0)
            {
                // объект исключит. ситуации создается "на месте"
                throw new ArgumentOutOfRangeException();
            }
            _field = value;
        }
    }
}

public class MainClass
{
    public static void Main()
    {
        var a = new ExampleClass();
        a.Field = -3; // ИС генерируется, но не обрабатывается!
    }
}
```

Так как в данном примере исключительная ситуация генерируется, но никак не обрабатывается, при работе приложения появится стандартное окно с сообщением об ошибке.

Опишем возможности по обработке исключительных ситуаций. Для перехвата исключительных ситуаций служит блок `try` – `catch` – `finally`. Синтаксис блока следующий:

```
try
{
    [<операторы, способные вызвать исключительную ситуацию>]
}
[<один или несколько блоков catch>]
[finally
{
    <операторы из секции завершения>
}]
```

Операторы из части `finally` (если она присутствует) выполняются всегда, вне зависимости от того, произошла исключительная ситуация или нет. Если один из операторов, расположенных в блоке `try`, вызвал исключительную ситуацию, управление немедленно передается на блоки `catch`. Синтаксис отдельного блока `catch` следующий:

```
catch [(<тип ИС> [<идентификатор объекта ИС>])]
{
    <операторы обработки исключительной ситуации>
}
```

Здесь *<идентификатор объекта ИС>* – это некая временная переменная, которая может использоваться для извлечения информации из объекта исключительной ситуации. Отдельно описывать эту переменную нет необходимости. Если указать в блоке `catch` оператор `throw`, это приведёт к тому, что обрабатываемая исключительная ситуация будет сгенерирована повторно.

Модифицируем программу, описанную выше, добавив в нее блок перехвата ошибки:

```
public static void Main()
{
    var a = new ExampleClass();
    try
    {
        Console.WriteLine("Эта строка печатается всегда");
        a.Field = -3;
        Console.WriteLine("Эта строка не печатается, если ошибка");
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.WriteLine(ex.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        Console.WriteLine("Строка печатается всегда (finally)");
    }
}
```

Если используется несколько блоков `catch`, то обработка исключительных ситуаций должна вестись по принципу «от частного – к общему», так как после выполнения одного блока `catch` управление передается на часть `finally` (при отсутствии `finally` – на оператор после `try` – `catch`). Компилятор C# не позволяет разместить блоки `catch` так, чтобы предыдущий блок перехватывал исключительные ситуации, предназначенные последующим блокам.

1.27. ПРЕПРОЦЕССОРНЫЕ ДИРЕКТИВЫ

Препроцессорные директивы – это инструкции для компилятора, начинающиеся с символа `#` и расположенные в отдельной строке кода. В табл. 4 дано описание директив и их действия.

Таблица 4

Препроцессорные директивы

Директива	Действие
`#define символ`	Определяет указанный *символ*
`#undef символ`	Удаляет определение *символа*
`#if символ [операция символ_2]...`	Тестирует, определён ли *символ* или набор символов, связанных *операциями* ==, !=, &&, ||
`#else`	Альтернативная ветвь для `#if`
`#elif символ [операция символ_2]...`	Комбинация `#else` и последующего `#if`
`#endif`	Окончание блока `#if`
`#warning текст`	Задаёт текст предупреждения, генерируемого компилятором
`#error текст`	Текст ошибки, генерируемой компилятором
`#line [номер ["файл"] | hidden]`	Позволят принудительно задать номер строки в исходном коде
`#region [имя]`	Начало региона кода, который можно «свернуть» в редакторе
`#endregion`	Окончание региона кода

Наиболее часто применяются условные директивы, которые дают возможность включить или проигнорировать участок кода при компиляции. В следующем примере вызов `Console.WriteLine()` будет компилироваться, пока установлена директива `#define DEBUG`:

```
#define DEBUG
internal class MyClass
{
    private int x;

    private void Foo()
    {
# if DEBUG
        Console.WriteLine("Тест: x = {0}", x);
# endif
    }
}
```

Символы для условной компиляции можно описать не только с помощью `#define`, но и указав ключ компилятора командной строки `/define` или используя окно свойств проекта в Visual Studio (в этих случаях описание символа распространяется не на отдельный файл, а на всю сборку).

1.28. ДОКУМЕНТИРОВАНИЕ ИСХОДНОГО КОДА

C# позволяет при написании программы снабжать исходный код особыми *документирующими комментариями*. Содержимое документирующих комментариев может затем быть выделено и обработано. Так реализуется концепция, при которой сам исходный код содержит необходимую документацию, описывающую его.

Рассмотрим общие принципы документирования кода. Документирующие комментарии – это комментарии, начинающиеся с последовательности `///`. Они могут располагаться в любом месте кода, но обычно их помещают перед описанием пользовательского типа или перед методом. Кроме собственно текста, комментарии могут содержать *документирующие теги*. Теги позволяют выделить некие особые составляющие комментария – например, имя метода, параметры, пример использования. Если в тексте комментария нужно использовать символы `<` и `>`, то они заменяются последовательностями `<` и `>`. В случае ссылок на универсальные шаблоны имя параметра-типа может быть записано в фигурных скобках.

Приведём табл. 5, в которой описаны документирующие теги.

Таблица 5

Документирующие теги

Тег и синтаксис	Описание
`<c>text</c>` `<code>content</code>`	Позволяют вставить в комментарий текст, являющийся кодом. Второй тег применяется при необходимости вставить несколько строк кода

`<example>`
`  description`
`</example>`
	Помечает части комментария, являющиеся примером. Обычно данный тег включает тег `<code>`

`<exception cref="member">`
`  description`
`</exception>`
	Применяется при определении методов и показывает, какие исключения методы могут сгенерировать. Атрибут `cref` должен указывать на существующий тип ИС

`<para>content</para>`	Используется для визуального оформления - выделяет параграф

`<param name="name">`
`  description`
`</param>`
	Описывает параметр метода

`<paramref name="name"/>`	Указывает, что элемент комментария является не просто словом, а параметром метода

`<remarks>`
`  description`
`</remarks>`
	Содержит дополнительное описание

`<returns>`
`  description`
`</returns>`
	Описание возвращаемого значения метода

`<see cref="member"/>` `<seealso cref="mem"/>`	Устанавливают ссылки на существующий тип или элемент типа

`<summary>`
`  description`
`</summary>`
	Содержит основное описание типа или элемента типа

`<typeparam name="name">`
`  description`
`</typeparam>`
	Позволяет указать описание generic-параметра

`<value>`
`  property-description`
`</value>`
	Используется для описания свойства

Ниже приведен фрагмент кода с документирующими комментариями.

```
/// <summary>
/// Starts the network device.
/// </summary>
/// <see cref="List{T}"/>
/// <param name="startTimeout">The start timeout.</param>
/// <param name="sessionId">The session id.</param>
public bool Start(int startTimeout, int sessionId) { . . . }
```

Чтобы выделить документирующие комментарии из исходного кода, можно откомпилировать программу с ключом `/doc:file`, где `file` – это имя XML-файла с комментариями. При работе с Visual Studio в свойствах проекта достаточно установить флаг `Build | Output | Xml Documentation File`.

Заметим, что существуют самостоятельные проекты, которые расширяют возможности встроенной системы документирования кода. Упомянем такие проекты как NDoc и Sandcastle. Также достаточно популярным является GhostDoc - дополнение к Visual Studio, облегчающее генерирование документирующих комментариев.