МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования «Казанский (Приволжский) федеральный университет»

Набережночелнинский институт (филиал)

Кафедра Бизнес-информатики и математических методов в экономике

Объектно – ориентированное программирование

Учебно-методическое пособие

Набережные Челны
2019 г.

УДК 004.043
ББК 32.973–018.2
И
21

Печатается по решению учебно-методической комиссии экономического отделения Набережночелнинского института (филиала) федерального государственного автономного образовательного учреждения высшего образования «Казанский (Приволжский) федеральный университет», от «22» января 2019г. (протокол № 6)

Рецензенты:
Доктор физ.-мат. наук, профессор А.Г. Исавнин
Доктор экономических наук, профессор А.Н. Макаров

Лысанов Д.М., Хамидуллин М.Р. Объектно – ориентированное программироование: учебно-методическое пособие / Д.М. Лысанов, М.Р. Хамидуллин – Набережные Челны: Изд-во Набережночелнинского института КФУ, 2019. – 155 с.

Учебно-методическое пособие содержит последовательное изложение базовых понятий теории объектно – ориентированного програмирования. Подробно изложены: основы программирования на C#; классы; обработка исключений; интерфейсы; делегаты, события и лямбды.

Учебно-методическое пособие предназначено для использования в учебном процессе студентами технических направлений в экономике и экономического отделения дневной, заочной и дистанционной форм обучения.

Содержание

Основы программирования на C#
Переменные
Для хранения данных в программе применяются переменные. Переменная представляет именнованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.
Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:
```
тип имя_переменной;
```
Вначале идет тип переменной, потом ее имя. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:
- имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания
- в имени не должно быть знаков пунктуации и пробелов
- имя не может быть ключевым словом языка C#. Таких слов не так много, и при работе в Visual Studio среда разработки подсвечивает ключевые слова синим цветом.

Хотя имя переменой может быть любым, но следует давать переменным описательные имена, которые будут говорить об и предназначении.
Например, определим простейшую переменную:
```
string name;
```
В данном случае определена переменная name, которая имеет тип string. то есть переменная представляет строку. Поскольку определение переменной представляет собой инструкцию, то после него ставится точка с запятой.
При этом следует учитывать, что C# является регистрозависимым языком, поэтому следующие два определения переменных будут представлять две разные переменные:
```
string name;
string Name;
```
После определения переменной можно присвоить некоторое значение:
```
string name;
name = "Tom";
```
Так как переменная name представляет тип string, то есть строку, то мы можем присвоить ей строку в двойных кавычках. Причем переменной можно присвоить только то значение, которое соответствует ее типу.
В дальнейшем с помощью имени переменной мы сможем обращаться к той области памяти, в которой хранится ее значение.
Также мы можем сразу при определении присвоить переменной значение. Данный прием называется инициализацией:
```
string name = "Tom";
```
Отличительной чертой переменных является то, что в программе можно многократно менять их значение. Например, создадим небольшую программу, в которой определим переменную, поменяем ее значение и выведем его на консоль:
```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "Tom";  // определяем переменную и инициализируем ее

            Console.WriteLine(name);    // Tom

            name = "Bob";       // меняем значение переменной
            Console.WriteLine(name);    // Bob

            Console.Read();
        }
    }
```

```
}
```
Консольный вывод программы:
```
Tom
Bob
```

Литералы

Литералы представляют неизменяемые значения (иногда их еще называют константами). Литералы можно передавать переменным в качестве значения. Литералы бывают логическими, целочисленными, вещественными, символьными и строчными. И отдельный литерал представляет ключевое слово null.

Логические литералы

Есть две логических константы - true (истина) и false (ложь):

```
Console.WriteLine(true);
Console.WriteLine(false);
```

Целочисленные литералы

Целочисленные литералы представляют положительные и отрицательные целые числа, например, 1, 2, 3, 4, -7, -109. Целочисленные литералы могут быть выражены в десятичной, шестнадцатеричной и двоичной форме.

С целыми числами в десятичной форме все должно быть понятно, так как они используются в повседневной жизни:

```
Console.WriteLine(-11);
Console.WriteLine(5);
Console.WriteLine(505);
```

Числа в двоичной форме предваряются символами 0b, после которых идет набор из нулей и единиц:

```
Console.WriteLine(0b11);       // 3
Console.WriteLine(0b1011);     // 11
Console.WriteLine(0b100001);   // 33
```

Для записи числа в шестнадцатеричной форме применяются символы 0x, после которых идет набор символов от 0 до 9 и от A до F, которые собственно представляют число:

```
Console.WriteLine(0x0A);   // 10
Console.WriteLine(0xFF);   // 255
Console.WriteLine(0xA1);   // 161
```

Вещественные литералы

Вещественные литералы представляют вещественные числа. Этот тип литералов имеет две формы. Первая форма - вещественные числа с фиксированной запятой, при которой дробную часть отделяется от целой части точкой. Например:

```
3.14
100.001
-0.38
```

Также вещественные литералы могут определяться в экспоненциальной форме MEp, где M — мантисса, E - экспонента, которая фактически означает "*10^" (умножить на десять в степени), а p — порядок. Например:

```
Console.WriteLine(3.2e3);   // по сути равно 3.2 * 10<sup>3</sup> = 3200
Console.WriteLine(1.2E-1);  // равно 1.2 * 10<sup>-1</sup> = 0.12
```

Символьные литералы

Символьные литералы представляют одиночные символы. Символы заключаются в одинарные кавычки.

Символьные литералы бывают нескольких видов. Прежде всего это обычные символы:

```
'2'
'A'
'T'
```

Специальную группу представляют управляющие последовательности Управляющая последовательность представляет символ, перед которым ставится обратный слеш. И данная последовательность интерпретируется определенным образом. Наиболее часто используемые последовательности:

- '\n' - перевод строки
- '\t' - табуляция
- '\' - обратный слеш

И если компилятор встретит в тексте последовательность \t, то он будет воспринимать эту последовательность не как слеш и букву t, а как табуляцию - то есть длинный отступ.
Также символы могут определяться в виде шестнадцатеричных кодов, также заключенный в одинарные кавычки.
Еще один способ определения символов представляет использования шестнадцатеричных кодов ASCII. Для этого в одинарных кавычках указываются символы '\x', после которых идет шестнадцатеричный код символа из таблицы ASCII. Коды символов из таблицы ASCII можно посмотреть здесь.
Например, литерал '\x78' представляет символ "x":
```
Console.WriteLine('\x78');    // x
Console.WriteLine('\x5A');    // Z
```
И последний способ определения символьных литералов представляет применение кодов из таблицы символов Unicode. Для этого в одинарных кавычках указываются символы '\u', после которых идет шестнадцатеричный код Unicode. Например, код '\u0411' представляет кириллический символ 'Б':
```
Console.WriteLine('\u0420');    // P
Console.WriteLine('\u0421');    // C
```
Строковые литералы
Строковые литералы представляют строки. Строки заключаются в двойные кавычки:
```
Console.WriteLine("hello");
Console.WriteLine("фыва");
Console.WriteLine("hello word");
```
Если внутри строки необходимо вывести двойную кавычку, то такая внутренняя кавычка предваряется обратным слешем:
```
Console.WriteLine("Компания \"Рога и копыта\"");
```
Также в строках можно использовать управляющие последовательности. Например, последовательность '\n' осуществляет перевод на новую строку:
```
Console.WriteLine("Привет \nмир");
```
При выводе на консоль слово "мир" будет перенесено на новую строку:
```
Привет
мир
```
null
null представляет ссылку, которая не указывает ни на какой объект. То есть по сути отсутствие значения.

Типы данных
Как и во многих языках программирования, в C# есть своя система типов данных, которая используется для создания переменных. Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.
В языке C# есть следующие примитивные типы данных:

- bool: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

```
bool alive = true;
bool isDead = false;
```

- byte: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

```
byte bit1 = 1;
byte bit2 = 102;
```

- sbyte: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

```
sbyte bit1 = -101;
sbyte bit2 = 102;
```

- short: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

```
short n1 = 1;
short n2 = 102;
```

- ushort: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

```
ushort n1 = 1;
ushort n2 = 102;
```

- int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

```
int a = 10;
int b = 0b101;  // бинарная форма b =5
int c = 0xFF;   // шестнадцатеричная форма c = 255
```

- uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

```
uint a = 10;
uint b = 0b101;
uint c = 0xFF;
```

- long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

```
long a = -10;
long b = 0b101;
long c = 0xFF;
```

- ulong: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

```
ulong a = 10;
ulong b = 0b101;
ulong c = 0xFF;
```

- float: хранит число с плавающей точкой от $-3.4*10^{38}$ до $3.4*10^{38}$ и занимает 4 байта. Представлен системным типом System.Single
- double: хранит число с плавающей точкой от $\pm5.0*10^{-324}$ до $\pm1.7*10^{308}$ и занимает 8 байта. Представлен системным типом System.Double
- decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от 0 до +/–79 228 162 514 264 337 593 543 950 335; если с запятой, то от 0 до +/–7,9228162514264337593543950335 с 28 разрядами после запятой и занимает 16 байт. Представлен системным типом System.Decimal
- char: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

```
char a = 'A';
char b = '\x5A';
char c = '\u0420';
```

- string: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют символьные литералы.

```
string hello = "Hello";
string word = "world";
```

- object: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

```
object a = 22;
object b = 3.14;
object c = "hello code";
```

Например, определим несколько переменных разных типов и выведем их значения на консоль:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "Tom";
            int age = 33;
            bool isEmployed = false;
            double weight = 78.65;

            Console.WriteLine($"Имя: {name}");
            Console.WriteLine($"Возраст: {age}");
            Console.WriteLine($"Вес: {weight}");
```

```
            Console.WriteLine($"Работает: {isEmployed}");
        }
    }
}
```
Для вывода данных на консоль здесь применяется интерполяция: перед строкой ставится знак $ и после этого мы можем вводить в строку в фигурных скобках значения переменных. Консольный вывод программы:
```
Имя: Tom
Возраст: 33
Вес: 78,65
Работает: False
```
Использование суффиксов
При присвоении значений надо иметь в виду следующую тонкость: все вещественные литералы рассматриваются как значения типа double. И чтобы указать, что дробное число представляет тип float или тип decimal, необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.
```
float a = 3.14F;
float b = 30.6f;

decimal c = 1005.8M;
decimal d = 334.8m;
```
Подобным образом все целочисленные литералы рассматриваются как значения типа int. Чтобы явным образом указать, что целочисленный литерал представляет значение типа uint, надо использовать суффикс U/u, для типа long - суффикс L/l, а для типа ulong - суффикс UL/ul:
```
uint a = 10U;
long b = 20L;
ulong c = 30UL;
```
Использование системных типов
Выше при перечислении всех базовых типов данных для каждого упоминался системный тип. Потому что название встроенного типа по сути представляет собой сокращенное обозначение системного типа. Например, следующие переменные будут эквивалентны по типу:
```
int a = 4;
System.Int32 b = 4;
```
Неявная типизация
Ранее мы явным образом указывали тип переменных, например, int x;. И компилятор при запуске уже знал, что x хранит целочисленное значение.
Однако мы можем использовать и модель неявной типизации:
```
var hello = "Hell to World";
var c = 20;

Console.WriteLine(c.GetType().ToString());
Console.WriteLine(hello.GetType().ToString());
```
Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. В примере выше использовалось выражение Console.WriteLine(c.GetType().ToString());, которое позволяет нам узнать выведенный тип переменной c. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная c будет иметь тип int или System.Int32
Эти переменные подобны обычным, однако они имеют некоторые ограничения.
Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:
```
// этот код работает
int a;
a = 20;

// этот код не работает
var c;
c= 20;
```
Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:
```
// этот код не работает
var c=null;
```

Так как значение null, то компилятор не сможет вывести тип данных.

double или decimal

Из выше перечисленного списка типов данных очевидно, что если мы хотим использовать в программе числа до 256, то для их хранения мы можем использоват переменные типа byte. При использовании больших значений мы можем взять тип short, int, long. То же самое для дробных чисел - для обычных дробных чисел можно взять тип float, для очень больших дробных чисел - тип double. Тип decimal здесь стоит особняком в том плане, что несмотря на большую разрядность по сравнению с типом double, тип double может хранить большее значение. Однако значение decimal может содержать до 28-29 знаков после запятой, тогда как значение типа double - 15-16 знаков после запятой.

Decimal чаще находит применение в финансовых вычислениях, тогда как double - в математических операциях. Общие различия между этими двумя типами можно выразить следующей таблицей:

	Decimal	Double
Наибольшее значение	$\sim 10^{28}$	$\sim 10^{308}$
Наименьшее значение (без учета нуля)	10^{-28}	$\sim 10^{-323}$
Знаков после запятой	28-29	15-16
Разрядность	16 байт	8 байт
Операций в секунду	сотни миллионов	миллиарды

Консольный ввод-вывод

Консольный вывод

Для вывода информации на консоль мы уже использовали встроенный метод Console.WriteLine. То есть, если мы хотим вывести некоторую информацию на консоль, то нам надо передать ее в метод Console.WriteLine:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string hello = "Привет мир";
            Console.WriteLine(hello);
            Console.WriteLine("Добро пожаловать в C#!");
            Console.WriteLine("Пока мир...");
            Console.WriteLine(24.5);

            Console.ReadKey();
        }
    }
}
```

Консольный вывод:

```
Привет мир!
Добро пожаловать в C#!
Пока мир...
24,5
```

Нередко возникает необходимость вывести на консоль в одной строке значения сразу нескольких переменных. В этом случае мы можем использовать прием, который называется интерполяцией:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            string name = "Tom";
            int age = 34;
            double height = 1.7;
            Console.WriteLine($"Имя: {name}  Возраст: {age}  Рост: {height}м");

            Console.ReadKey();
        }
    }
}
```

Для встраивания отдельных значений в выводимую на консоль строку используются фигурные скобки, в которые заключается встраиваемое значение. Это можем значение переменной ({name}) или более сложное выражение (например, операция сложения {4 + 7}). А перед всей строкой ставится знак доллара $.

При выводе на консоль вместо помещенных в фигурные скобки выражений будут выводиться их значения:

```
Имя: Tom  Возраст: 34  Рост: 1,7м
```

Есть другой способ вывода на консоль сразу нескольких значений:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "Tom";
            int age = 34;
            double height = 1.7;
            Console.WriteLine("Имя: {0}  Возраст: {2}  Рост: {1}м", name, height, age);

            Console.ReadKey();
        }
    }
}
```

Этот способ подразумевает, что первый параметр в методе Console.WriteLine представляет выводимую строку ("Имя: {0} Возраст: {2} Рост: {1}м"). Все последующие параметры представляют значения, которые могут быть встроенны в эту строку (name, height, age). При этом важен порядок подобных параметров. Например, в данном случае вначале идет name, потом height и потом age. Поэтому у name будет представлять параметр с номером 0 (нумерация начинается с нуля), height имеет номер 1, а age - номер 2. Поэтому в строке "Имя: {0} Возраст: {2} Рост: {1}м" на место плейсхолдеров {0}, {2}, {1} будут вставляться значения соответствующих параметров.

Кроме Console.WriteLine() можно также использовать метод Console.Write(), он работает точно так же за тем исключением, что не осуществляет переход на следующую строку.

Консольный ввод

Кроме вывода информации на консоль мы можем получать информацию с консоли. Для этого предназначен метод Console.ReadLine(). Он позволяет получить введенную строку.

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите свое имя: ");
            string name = Console.ReadLine();
            Console.WriteLine($"Привет {name}");

            Console.ReadKey();
        }
    }
}
```

В данном случае все, что вводит пользователь, с помощью метода Console.ReadLine передается в переменную name.
Пример работы программы:
```
Введите свое имя: Том
Привет Том
```
Таким образом мы можем вводить информацию через консоль. Однако минусом этого метода является то, что Console.ReadLine считывает информацию именно в виде строки. Поэтому мы можем по умолчанию присвоить ее только переменной типа string. Как нам быть, если, допустим, мы хотим ввести возраст в переменную типа int или другую информацию в переменные типа double или decimal? По умолчанию платформа .NET предоставляет ряд методов, которые позволяют преобразовать различные значения к типам int, double и т.д. Некоторые из этих методов:

- Convert.ToInt32() (преобразует к типу int)
- Convert.ToDouble() (преобразует к типу double)
- Convert.ToDecimal() (преобразует к типу decimal)

Пример ввода значений:
```
using System;

namespace HelloApp
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.Write("Введите имя: ");
         string name = Console.ReadLine();

         Console.Write("Введите возраст: ");
         int age = Convert.ToInt32(Console.ReadLine());

         Console.Write("Введите рост: ");
         double height = Convert.ToDouble(Console.ReadLine());

         Console.Write("Введите размер зарплаты: ");
         decimal salary = Convert.ToDecimal(Console.ReadLine());

         Console.WriteLine($"Имя: {name}  Возраст: {age}  Рост: {height}м  Зарплата: {salary}$");

         Console.ReadKey();
      }
   }
}
```
При вводе важно учитывать текущую операционную систему. В одних культурах разделителем между целой и дробной частью является точка (США, Великобритания...), в других - запятая (Россия, Германия...). Например, если текущая ОС - русскоязычная, значит, надо вводить дробные числа с разделителем запятой. Если локализация англоязычная, значит, разделителем целой и дробной части при вводе будет точка.
Пример работы программы:
```
Введите имя: Том
Введите возраст: 25
Введите рост: 1,75
Введите размер зарплаты: 300,67
Имя: Том  Возраст: 25  Рост: 1,75м  Зарплата: 300,67$
```

Арифметические операции языка C#

В C# используется большинство операций, которые применяются и в других языках программирования. Операции представляют определенные действия над операндами - участниками операции. В качестве операнда может выступать переменной или какое-либо значение (например, число). Операции бывают унарными (выполняются над одним операндом), бинарными - над двумя операндами и тернарными - выполняются над тремя операндами. Рассмотрим все виды операций.

Бинарные арифметические операции:

- +

  Операция сложения двух чисел:

```
int x = 10;
int z = x + 12; // 22
```

- -

  Операция вычитания двух чисел:

```
int x = 10;
int z = x - 6; // 4
```

- *

  Операция умножения двух чисел:

```
int x = 10;
int z = x * 5; // 50
```

- /

  операция деления двух чисел:

```
int x = 10;
int z = x / 5; // 2

double a = 10;
double b = 3;
double c = a / b; // 3.33333333
```

При делении стоит учитывать, что если оба операнда представляют целые числа, то результат также будет округляться до целого числа:

```
double z = 10 /  4; //результат равен 2
```

Несмотря на то, что результат операции в итоге помещается в переменную типа double, которая позволяет сохранить дробную часть, но в самой опеации участвуют два литерала, которые по умолчанию рассматриваются как объекты int, то есть целые числа, и результат то же будет целочисленный.

Для выхода из этой ситуации необходимо определять литералы или переменные, участвующие в операции, именно как типы double или float:

```
double z = 10.0 /  4.0; //результат равен 2.5
```

- %

  Операция получение остатка от целочисленного деления двух чисел:

```
double x = 10.0;
double z = x % 4.0; //результат равен 2
```

Также есть ряд унарных операций, в которых принимает участие один операнд:

- ++

  Операция инкремента

  Инкремент бывает префиксным: ++x - сначала значение переменной x увеличивается на 1, а потом ее значение возвращается в качестве результата операции.

  И также существует постфиксный инкремент: x++ - сначала значение переменной x возвращается в качестве результата операции, а затем к нему прибавляется 1.

```
int x1 = 5;
int z1 = ++x1; // z1=6; x1=6
Console.WriteLine($"{x1} - {z1}");

int x2 = 5;
int z2 = x2++; // z2=5; x2=6
Console.WriteLine($"{x2} - {z2}");
```

- --

  Операция декремента или уменьшения значения на единицу. Также существует префиксная форма декремента (--x) и постфиксная (x--).

```
int x1 = 5;
int z1 = --x1; // z1=4; x1=4
Console.WriteLine($"{x1} - {z1}");

int x2 = 5;
int z2 = x2--; // z2=5; x2=4
Console.WriteLine($"{x2} - {z2}");
```

При выполнении сразу нескольких арифметических операций следует учитывать порядок их выполнения. Приоритет операций от наивысшего к низшему:

1. Инкремент, декремент
2. Умножение, деление, получение остатка
3. Сложение, вычитание

Для изменения порядка следования операций применяются скобки.
Рассмотрим набор операций:

```
int a = 3;
int b = 5;
int c = 40;
int d = c---b*a;    // a=3  b=5  c=39  d=25
Console.WriteLine($"a={a}  b={b}  c={c}  d={d}");
```

Здесь мы имеем дело с тремя операциями: декремент, вычитание и умножение. Сначала выполняется декремент переменной c, затем умножение b*a, и в конце вычитание. То есть фактически набор операций выглядел так:

```
int d = (c--)-(b*a);
```

Но с помощью скобок мы могли бы изменить порядок операций, например, следующим образом:

```
int a = 3;
int b = 5;
int c = 40;
int d = (c-(--b))*a;    // a=3  b=4  c=40  d=108
Console.WriteLine($"a={a}  b={b}  c={c}  d={d}");
```

Ассоциативность операторов

Как выше было отмечено, операции умножения и деления имеют один и тот же приоритет, но какой тогда результат будет в выражении:

```
int x = 10 / 5 * 2;
```

Стоит нам трактовать это выражение как `(10 / 5) * 2` или как `10 / (5 * 2)`? Ведь в зависимости от трактовки мы получим разные результаты.
Когда операции имеют один и тот же приоритет, порядок вычисления определяется ассоциативностью операторов. В зависимости от ассоциативности есть два типа операторов:

- Левоассоциативные операторы, которые выполняются слева направо
- Правоассоциативные операторы, которые выполняются справа налево

Все арифметические операторы (кроме префиксного инкремента и декремента) являются левоассоциативными, то есть выполняются слева направо. Поэтому выражение `10 / 5 * 2` необходимо трактовать как `(10 / 5) * 2`, то есть результатом будет 4.

Поразрядные операции

Особый класс операций представляют поразрядные операции. Они выполняются над отдельными разрядами числа. В этом плане числа рассматриваются в двоичном представлении, например, 2 в двоичном представлении 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

Логические операции

- &(логическое умножение)
  Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

```
int x1 = 2; //010
 int y1 = 5;//101
 Console.WriteLine(x1&y1); // выведет 0

int x2 = 4; //100
int y2 = 5; //101
Console.WriteLine(x2 & y2); // выведет 4
```

  В первом случае у нас два числа 2 и 5. 2 в двоичном виде представляет число 010, а 5 - 101. Поразрядно умножим числа (0*1, 1*0, 0*1) и в итоге получим 000.
  Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же как и у числа 5, поэтому в итоге получим (1*1, 0*0, 0 *1) = 100, то есть число 4 в десятичном формате.
- | (логическое сложение)

Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица. Например:

```
int x1 = 2; //010
int y1 = 5;//101
Console.WriteLine(x1|y1); // выведет 7 - 111
int x2 = 4; //100
int y2 = 5;//101
Console.WriteLine(x2 | y2); // выведет 5 - 101
```

- ^ (логическое исключающее ИЛИ)

  Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

```
int x = 45; // Значение, которое надо зашифровать - в двоичной форме 101101
int key = 102; //Пусть это будет ключ - в двоичной форме 1100110
int encrypt = x ^ key; //Результатом будет число 1001011 или 75
Console.WriteLine("Зашифрованное число: " +encrypt);

int decrypt = encrypt ^ key; // Результатом будет исходное число 45
Console.WriteLine("Расшифрованное число: " + decrypt);
```

  Здесь опять же производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Таким образом, мы получаем из 9^5 в качестве результата число 12. И чтобы расшифровать число, мы применяем ту же операцию к результату.

- ~ (логическое отрицание или инверсия)

  Еще одна поразрядная операция, которая инвертирует все разряды: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

```
int x = 12;              // 00001100
Console.WriteLine(~x);   // 11110011   или -13
```

Представление отрицательных чисел

Для записи чисел со знаком в C# применяется дополнительный код (two's complement), при котором старший разряд является знаковым. Если его значение равно 0, то число положительное, и его двоичное представление не отличается от представления беззнакового числа. Например, 0000 0001 в десятичной системе 1.

Если старший разряд равен 1, то мы имеем дело с отрицательным числом. Например, 1111 1111 в десятичной системе представляет -1. Соответственно, 1111 0011 представляет -13.

Чтобы получить из положительного числа отрицательное, его нужно инвертировать и прибавить единицу:

```
int x = 12;
int y = ~x;
y += 1;
Console.WriteLine(y);   // -12
```

Операции сдвига

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

- x<<y - сдвигает число x влево на y разрядов. Например, 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, то есть в итоге получается 1000 или число 8 в десятичном представлении.
- x>>y - сдвигает число x вправо на y разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичном представлении 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два.

Операции присваивания

Операции присвоения устанавливают значение. В операциях присвоения участвуют два операнда, причем левый операнд может представлять только модифицируемое именованное выражение, например, переменную

Как и во многих других языках программирования, в C# имеется базовая операция присваивания =, которая присвоивает значение правого операнда левому операнду:

```
int number = 23;
```

Здесь переменной number присваивается число 23. Переменная number представляет левый операнд, которому присваивается значение правого операнда, то есть числа 23.

Также можно выполнять множественно присвоение сразу нескольких переменным одновременно:

```
int a, b, c;
a = b = c = 34;
```

Стоит отметить, что операции присвоения имеют низкий приоритет. И вначале будет вычисляться значение правого операнда и только потом будет идти присвоение этого значения левому операнду. Например:

```
int a, b, c;
a = b = c = 34 * 2 / 4; // 17
```

Сначала будет вычисляться выражение 34 * 2 / 4, затем полученное значение будет присвоено переменным.

Кроме базовой операции присвоения в C# есть еще ряд операций:

- +=: присваивание после сложения. Присваивает левому операнду сумму левого и правого операндов: выражение A += B равнозначно выражению A = A + B
- -=: присваивание после вычитания. Присваивает левому операнду разность левого и правого операндов: A -= B эквивалентно A = A - B
- *=: присваивание после умножения. Присваивает левому операнду произведение левого и правого операндов: A *= B эквивалентно A = A * B
- /=: присваивание после деления. Присваивает левому операнду частное левого и правого операндов: A /= B эквивалентно A = A / B
- %=: присваивание после деления по модулю. Присваивает левому операнду остаток от целочисленного деления левого операнда на правый: A %= B эквивалентно A = A % B
- &=: присваивание после поразрядной конъюнкции. Присваивает левому операнду результат поразрядной конъюнкции его битового представления с битовым представлением правого операнда: A &= B эквивалентно A = A & B
- |=: присваивание после поразрядной дизъюнкции. Присваивает левому операнду результат поразрядной дизъюнкции его битового представления с битовым представлением правого операнда: A |= B эквивалентно A = A | B
- ^=: присваивание после операции исключающего ИЛИ. Присваивает левому операнду результат операции исключающего ИЛИ его битового представления с битовым представлением правого операнда: A ^= B эквивалентно A = A ^ B
- <<=: присваивание после сдвига разрядов влево. Присваивает левому операнду результат сдвига его битового представления влево на определенное количество разрядов, равное значению правого операнда: A <<= B эквивалентно A = A << B
- >>=: присваивание после сдвига разрядов вправо. Присваивает левому операнду результат сдвига его битового представления вправо на определенное количество разрядов, равное значению правого операнда: A >>= B эквивалентно A = A >> B

Применение операций присвоения:

```
int a = 10;
a += 10;       // 20
a -= 4;        // 16
a *= 2;        // 32
a /= 8;        // 4
a <<= 4;       // 64
a >>= 2;       // 16
```

Операции присвоения являются правоассоциативными, то есть выполняются справа налево. Например:

```
int a = 8;
int b = 6;
int c = a += b -= 5;    // 9
```

В данном случае выполнение выражения будет идти следующим образом:

1. b -= 5 (6-5=1)
2. a += (b-=5) (8+1 = 9)
3. c = (a += (b-=5)) (c = 9)

Преобразования базовых типов данных

При рассмотрении типов данных указывалось, какие значения может иметь тот или иной тип и сколько байт памяти он может занимать. В прошлой теме были расмотрены арифметические операции. Теперь применим операцию сложения к данным разных типов:

```
byte a = 4;
int b = a + 70;
```

Результатом операции вполне справедливо является число 74, как и ожидается.

Но теперь попробуем применить сложение к двум объектам типа byte:

```
byte a = 4;
byte b = a + 70;  // ошибка
```

Здесь поменялся только тип переменной, которая получает результат сложения - с int на byte. Однако при попытке скомпилировать программу мы получим ошибку на этапе компиляции. И если мы работаем в Visual Studio, среда подчеркнет вторую строку красной волнистой линией, указывая, что в ней ошибка.

При операциях мы должны учитывать диапазон значений, которые может хранить тот или иной тип. Но в данном случае число 74, которое мы ожидаем получить, вполне укладывается в диапазон значений типа byte, тем не менее мы получаем ошибку.

Дело в том, что операция сложения (да и вычитания) возвращает значение типа int, если в операции участвуют целочисленные типы данных с разрядностью меньше или равно int (то есть типы byte, short, int). Поэтому результатом операции `a + 70` будет объект, который имеет длину в памяти 4 байта. Затем этот объект мы пытаемся присвоить переменной b, которая имеет тип byte и в памяти занимает 1 байт.

И чтобы выйти из этой ситуации, необходимо применить операцию преобразования типов:

```
byte a = 4;
byte b = (byte)(a + 70);
```

Операция преобразования типов предполагает указание в скобках того типа, к которому надо преобразовать значение.

Сужающие и расширяющие преобразования

Преобразования могут сужающие (narrowing) и расширяющие (widening). Расширяющие преобразования расширяют размер объекта в памяти. Например:

```
byte a = 4;           // 0000100
ushort b = a;   // 000000000000100
```

В данном случае переменной типа ushort присваивается значение типа byte. Тип byte занимает 1 байт (8 бит), и значение переменной a в двоичном виде можно представить как:

```
00000100
```

Значение типа ushort занимает 2 байта (16 бит). И при присвоении переменной b значение переменной a расширяется до 2 байт

```
0000000000000100
```

То есть значение, которое занимает 8 бит, расширяется до 16 бит.

Сужающие преобразования, наоборот, сужают значение до типа меньшей разядности. Во втором листинге статьи мы как раз имели дело с сужающими преобразованиями:

```
ushort a = 4;
byte b = (byte) a;
```

Здесь переменной b, которая занимает 8 бит, присваивается значение ushort, которое занимает 16 бит. То есть из `0000000000000100` получаем `00000100`. Таким образом, значение сужается с 16 бит (2 байт) до 8 бит (1 байт).

Явные и неявные преобразования

Неявные преобразования

В случае с расширяющими преобразованиями компилятор за нас выполнял все преобразования данных, то есть преобразования были неявными (implicit conversion). Такие преобразования не вызывают каких-то затруднений. Тем не менее стоит сказать пару слов об общей механике подобных преобразований.

Если производится преобразование от безнакового типа меньшей разрядности к безнаковому типу большой разрядности, то добавляются дополнительные биты, которые имеют значени 0. Это называется дополнение нулями или zero extension.

```
byte a = 4;           // 0000100
ushort b = a;   // 000000000000100
```

Если производится преобразование к знаковому типу, то битовое представление дополняется нулями, если число положительное, и единицами, если число отрицательное. Последний разряд числа содержит знаковый бит - 0 для положительных и 1 для отрицательных чисел. При расширении в добавленные разряды компируется знаковый бит.
Рассмотрим преобразование положительного числа:

```
sbyte a = 4;          // 0000100
short b = a;    // 000000000000100
```

Преобразование отрицательного числа:

```
sbyte a = -4;          // 1111100
short b = a;    // 111111111111100
```

Явные преобразования

При явных преобразованиях (explicit conversion) мы сами должны применить операцию преобразования (операция ()). Суть операции преобразования типов состоит в том, что перед значением указывается в скобках тип, к которому надо привести данное значение:

```
int a = 4;
int b = 6;
byte c = (byte)(a+b);
```

Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью компилятор поводит неявно. Это могут быть следующие цепочки преобразований:
byte -> short -> int -> long -> decimal
int -> double
short -> float -> double
char -> int
Все безопасные преобразования автоматические преобразования можно описать следующей таблицей:

В какие типы безопасно преобразуется
short, ushort, int, uint, long, ulong, float, double, decimal
short, int, long, float, double, decimal
int, long, float, double, decimal
int, uint, long, ulong, float, double, decimal
long, float, double, decimal
long, ulong, float, double, decimal
float, double, decimal
float, double, decimal
double
ushort, int, uint, long, ulong, float, double, decimal

В остальных случаях следует использовать явные преобразования типов.
Также следует отметить, что несмотря на то, что и double, и decimal могут хранить дробные данные, а decimal имеет большую разрядность, чем double, но все равно значение double нужно явно приводить к типу decimal:

```
double a = 4.0;
decimal b = (decimal)a;
```

Потеря данных и ключевое слово checked

Рассмотрим другую ситуацию, что будет, например, в следующем случае:

```
int a = 33;
int b = 600;
byte c = (byte)(a+b);
```

Результатом будет число 121, так число 633 не попадает в допустимый диапазон для типа byte, и старшие биты будут усекаться. В итоге получится число 121. Поэтому при преобразованиях надо это учитывать. И мы в данном случае можем либо взять такие числа a и b, которые в сумме дадут число не больше 255, либо мы можем выбрать вместо byte другой тип данных, например, int.

Однако ситуации разные могут быть. Мы можем точно не знать, какие значения будут иметь числа a и b. И чтобы избежать подобных ситуаций, в c# имеется ключевое слово checked:

```
try
{
    int a = 33;
    int b = 600;
    byte c = checked((byte)(a + b));
    Console.WriteLine(c);
}
catch (OverflowException ex)
{
    Console.WriteLine(ex.Message);
}
```

При использовании ключевого слова checked приложение выбрасывает исключение о переполнении. Поэтому для его обработки в данном случае используется конструкция try...catch. Подробнее данную конструкцию и обработку исключений мы рассмотрим позже, а пока надо знать, что в блок try мы включаем действия, в которых может потенциально возникнуть ошибка, а в блоке catch обрабатываем ошибку.

Условные выражения

Отдельный набор операций представляет условные выражения. Такие операции возвращают логическое значение, то есть значение типа bool: true, если выражение истинно, и false, если выражение ложно. К подобным операциям относятся операции сравнения и логические операции.

Операции сравнения

В операциях сравнения сравниваются два операнда и возвращается значение типа bool - true, если выражение верно, и false, если выражение неверно.

- ==

  Сравнивает два операнда на равенство. Если они равны, то операция возвращает true, если не равны, то возвращается false:

```
int a = 10;
int b = 4;
bool c = a == b; // false
```

- !=

  Сравнивает два операнда и возвращает true, если операнды не равны, и false, если они равны.

```
int a = 10;
int b = 4;
bool c = a != b;    // true
bool d = a!=10;     // false
```

- <

  Операция "меньше чем". Возвращает true, если первый операнд меньше второго, и false, если первый операнд больше второго:

```
int a = 10;
int b = 4;
bool c = a < b; // false
```

- >

  Операция "больше чем". Сравнивает два операнда и возвращает true, если первый операнд больше второго, иначе возвращает false:

```
int a = 10;
int b = 4;
bool c = a > b;     // true
bool d = a > 25;    // false
```

- <=

  Операция "меньше или равно". Сравнивает два операнда и возвращает true, если первый операнд меньше или равен второму. Иначе возвращает false.

```
int a = 10;
int b = 4;
bool c = a <= b;     // false
bool d = a <= 25;    // true
```

- >=

Операция "больше или равно". Сравнивает два операнда и возвращает true, если первый операнд больше или равен второму, иначе возвращается false:

```
int a = 10;
int b = 4;
bool c = a >= b;     // true
bool d = a >= 25;    // false
```

Операции <, > <=, >= имеют больший приоритет, чем == и !=.

Логические операции

Также в C# определены логические операторы, которые также возвращают значение типа bool. В качестве операндов они принимают значения типа bool. Как правило, применяются к отношениям и объединяют несколько операций сравнения.

- |

  Операция логического сложения или логическое ИЛИ. Возвращает true, если хотя бы один из операндов возвращает true.

```
bool x1 = (5 > 6) | (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
bool x2 = (5 > 6) | (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false
```

- &

  Операция логического умножения или логическое И. Возвращает true, если оба операнда одновременно равны true.

```
bool x1 = (5 > 6) & (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false
bool x2 = (5 < 6) & (4 < 6); // 5 < 6 - true, 4 < 6 - true, поэтому возвращается true
```

- ||

  Операция логического сложения. Возвращает true, если хотя бы один из операндов возвращает true.

```
bool x1 = (5 > 6) || (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
bool x2 = (5 > 6) || (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false
```

- &&

  Операция логического умножения. Возвращает true, если оба операнда одновременно равны true.

```
bool x1 = (5 > 6) && (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false
bool x2 = (5 < 6) && (4 < 6); // 5 < 6 - true, 4 > 6 - true, поэтому возвращается true
```

- !

  Операция логического отрицания. Производится над одним операндом и возвращает true, если операнд равен false. Если операнд равен true, то операция возвращает false:

```
bool a = true;
bool b = !a;    // false
```

- ^

  Операция исключающего ИЛИ. Возвращает true, если либо первый, либо второй операнд (но не одновременно) равны true, иначе возвращает false

```
bool x5 = (5 > 6) ^ (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
bool x6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false
```

Здесь у нас две пары операций | и || (а также & и &&) выполняют похожие действия, однако же они не равнозначны.

В выражении z=x|y; будут вычисляться оба значения - x и y.

В выражении же z=x||y; сначала будет вычисляться значение x, и если оно равно true, то вычисление значения y уже смысла не имеет, так как у нас в любом случае уже z будет равно true. Значение y будет вычисляться только в том случае, если x равно false

То же самое касается пары операций &/&&. В выражении z=x&y; будут вычисляться оба значения - x и y.

В выражении же z=x&&y; сначала будет вычисляться значение x, и если оно равно false, то вычисление значения y уже смысла не имеет, так как у нас в любом случае уже z будет равно false. Значение y будет вычисляться только в том случае, если x равно true

Поэтому операции || и && более удобны в вычислениях, так как позволяют сократить время на вычисление значения выражения, и тем самым повышают производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами.

Операция && имеет больший приоритет, чем операция ||. Так, в выражении true || true && false сначала выполняется подвыражение true && false.

Условные конструкции

Условные конструкции - один из базовых компонентов многих языков программирования, которые направляют работу программы по одному из путей в зависимости от определенных условий.

В языке C# используются следующие условные конструкции: if..else и switch..case

Конструкция if/else

Конструкция if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
```

После ключевого слова if ставится условие. И если это условие выполняется, то срабатывает код, который помещен далее в блоке if после фигурных скобок. В качестве условий выступают ранее рассмотренные операции сравнения.

В данном случае у нас первое число больше второго, поэтому выражение num1 > num2 истинно и возвращает true, следовательно, управление переходит к строке Console.WriteLine("Число {num1} больше числа {num2}");

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
else
{
   Console.WriteLine($"Число {num1} меньше числа {num2}");
}
```

Но при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. Используя конструкцию else if, мы можем обрабатывать дополнительные условия:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
else if (num1 < num2)
{
   Console.WriteLine($"Число {num1} меньше числа {num2}");
}
else
{
   Console.WriteLine("Число num1 равно числу num2");
}
```

Также мы можем соединить сразу несколько условий, используя логические операторы:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2 && num1==8)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
```

В данном случае блок if будет выполняться, если num1 > num2 равно true и num1==8 равно true.

Конструкция switch

Конструкция switch/case аналогична конструкции if/else, так как позволяет обработать сразу несколько условий:

```
Console.WriteLine("Нажмите Y или N");
string selection = Console.ReadLine();
switch (selection)
{
```

```
    case "Y":
        Console.WriteLine("Вы нажали букву Y");
        break;
    case "N":
        Console.WriteLine("Вы нажали букву N");
        break;
    default:
        Console.WriteLine("Вы нажали неизвестную букву");
        break;
}
```

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

В конце каждого блока case должен ставиться один из операторов перехода: break, goto case, return или throw. Как правило, используется оператор break. При его применении другие блоки case выполняться не будут.

Однако если мы хотим, чтобы, наоборот, после выполнения текущего блока case выполнялся другой блок case, то мы можем использовать вместо break оператор goto case:

```
int number = 1;
switch (number)
{
    case 1:
        Console.WriteLine("case 1");
        goto case 5; // переход к case 5
    case 3:
        Console.WriteLine("case 3");
        break;
    case 5:
        Console.WriteLine("case 5");
        break;
    default:
        Console.WriteLine("default");
        break;
}
```

Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок default, как в примере выше.

Применение оператора return позволит выйти не только из блока case, но и из вызывающего метода. То есть, если в методе Main после конструкции switch..case, в которой используется оператор return, идут какие-либо операторы и выражения, то они выполняться не будут, а метод Main завершит работу.

Оператор throw применяется для выброса ошибок и будет рассмотрен в одной из следующим тем.

Тернарная операция

Тернарную операция имеет следующий синтаксис: [первый операнд - условие] ? [второй операнд] : [третий операнд]. Здесь сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:

```
int x=3;
int y=2;
Console.WriteLine("Нажмите + или -");
string selection = Console.ReadLine();

int z = selection=="+"? (x+y) : (x-y);
Console.WriteLine(z);
```

Здесь результатом тернарной операции является переменная z. Если мы выше вводим "+", то z будет равно второму операнду - (x+y). Иначе z будет равно третьему операнду.

Циклы

Циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:

- for
- foreach
- while
- do...while

Цикл for

Цикл for имеет следующее формальное определение:

```
for ([инициализация счетчика]; [условие]; [изменение счетчика])
{
   // действия
}
```

Рассмотрим стандартный цикл for:

```
for (int i = 0; i < 9; i++)
{
   Console.WriteLine($"Квадрат числа {i} равен {i*i}");
}
```

Первая часть объявления цикла - int i = 0 - создает и инициализирует счетчик i. Счетчик необязательно должен представлять тип int. Это может быть и другой числовой тип, например, float. И перед выполнением цикла его значение будет равно 0. В данном случае это то же самое, что и объявление переменной.

Вторая часть - условие, при котором будет выполняться цикл. Пока условное выражение возвращает true, будет выполняться цикл. В данном случае цикл будет выполняться, пока счетчик i не достигнет 9.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--.

В итоге блок цикла сработает 9 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

```
int i = 0;
for (; ;)
{
   Console.WriteLine($"Квадрат числа {++i} равен {i * i}");
}
```

Формально определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; i <;). У нас нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Мы также можем опустить ряд блоков:

```
int i = 0;
for (; i<9;)
{
   Console.WriteLine($"Квадрат числа {++i} равен {i * i}");
}
```

Этот пример по сути эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for.

Цикл do

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется. Например:

```
int i = 6;
do
{
   Console.WriteLine(i);
   i--;
}
while (i > 0);
```

Здесь код цикла сработает 6 раз, пока i не станет равным нулю. Но важно отметить, что цикл do гарантирует хотя бы единократное выполнение действий, даже если условие в инструкции while не будет истинно. То есть мы можем написать:

```
int i = -1;
do
{
```

```
    Console.WriteLine(i);
    i--;
}
while (i > 0);
```
Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.

Цикл while

В отличие от цикла do цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:
```
int i = 6;
while (i > 0)
{
    Console.WriteLine(i);
    i--;
}
```
Операторы continue и break

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором break.

Например:
```
for (int i = 0; i < 9; i++)
{
    if (i == 5)
        break;
    Console.WriteLine(i);
}
```
Хотя в условии цикла сказано, что цикл будет выполняться, пока счетчик i не достигнет значения 9, в реальности цикл сработает 5 раз. Так как при достижении счетчиком i значения 5, сработает оператор break, и цикл завершится.
```
0
1
2
3
4
```
Теперь поставим себе другую задачу. А что если мы хотим, чтобы при проверке цикл не завершался, а просто пропускал текущую итерацию. Для этого мы можем воспользоваться оператором continue:
```
for (int i = 0; i < 9; i++)
{
    if (i == 5)
        continue;
    Console.WriteLine(i);
}
```
В этом случае цикл, когда дойдет до числа 5, которое не удовлетворяет условию проверки, просто пропустит это число и перейдет к следующей итерации:
```
0
1
2
3
4
6
7
8
```
Массивы

Массив представляет набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки:
```
тип_переменной[] название_массива;
```
Например, определим массив целых чисел:
```
int[] numbers;
```
После определения переменной массива мы можем присвоить ей определенное значение:
```
int[] nums = new int[4];
```
Здесь вначале мы объявили массив nums, который будет хранить данные типа int. Далее используя операцию new, мы выделили память для 4 элементов массива: new int[4]. Число 4 еще

называется длиной массива. При таком определении все элементы получают значение по умолчанию, которое предусмотренно для их типа. Для типа int значение по умолчанию - 0.
Также мы сразу можем указать значения для этих элементов:

```
int[] nums2 = new int[4] { 1, 2, 3, 5 };

int[] nums3 = new int[] { 1, 2, 3, 5 };

int[] nums4 = new[] { 1, 2, 3, 5 };

int[] nums5 = { 1, 2, 3, 5 };
```

Все перечисленные выше способы будут равноценны.
Для обращения к элементам массива используются индексы. Индекс представляет номер элемента в массиве, при этом нумерация начинается с нуля, поэтому индекс первого элемента будет равен 0. А чтобы обратиться к четвертому элементу в массиве, нам надо использовать индекс 3, к примеру: nums[3]. Используем индексы для получения и установки значений элементов массива:

```
int[] nums = new int[4];
nums[0] = 1;
nums[1] = 2;
nums[2] = 3;
nums[3] = 5;
Console.WriteLine(nums[3]);  // 5
```

И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу: nums[5] = 5;. Если мы так попытаемся сделать, то мы получим исключение IndexOutOfRangeException.

Перебор массивов. Цикл foreach

Цикл foreach предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

```
foreach (тип_данных название_переменной in контейнер)
{
    // действия
}
```

Например:

```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
foreach (int i in numbers)
{
    Console.WriteLine(i);
}
```

Здесь в качестве контейнера выступает массив данных типа int. Поэтому мы объявляем переменную с типом int
Подобные действия мы можем сделать и с помощью цикл for:

```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.Length; i++)
{
    Console.WriteLine(numbers[i]);
}
```

В то же время цикл for более гибкий по сравнению с foreach. Если foreach последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы:

```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = numbers[i] * 2;
    Console.WriteLine(numbers[i]);
}
```

Многомерные массивы

Массивы характеризуются таким понятием как ранг или количество измерений. Выше мы рассматривали массивы, которые имеют одно измерение (то есть их ранг равен 1) - такие массивы можно представлять в виде горзонтального ряда элемента. Но массивы также бывают многомерными. У таких массивов количество измерений (то есть ранг) больше 1.

Массивы которые имеют два измерения (ранг равен 2) называют двухмерными. Например, создадим одномерный и двухмерный массивы, которые имеют одинаковые элементы:

```
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };

int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Визуально оба массива можно представить следующим образом:

Одномерный массив nums1

0	1	2	3	4	5

Двухмерный массив nums2

0	1	2
3	4	5

Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Все возможные способы определения двухмерных массивов:

```
int[,] nums1;
int[,] nums2 = new int[2, 3];
int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums5 = new [,]{ { 0, 1, 2 }, { 3, 4, 5 } };
int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Массивы могут иметь и большее количество измерений. Объявление трехмерного массива могло бы выглядеть так:

```
int[,,] nums3 = new int[2, 3, 4];
```

Соответственно могут быть и четырехмерные массивы и массивы с большим количеством измерений. Но на практике обычно используются одномерные и двухмерные массивы.

Определенную сложность может представлять перебор многомерного массива. Прежде всего надо учитывать, что длина такого массива - это совокупное количество элементов.

```
int[,] mas = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };
foreach (int i in mas)
    Console.Write($"{i} ");
Console.WriteLine();
```

В данном случае длина массива mas равна 12. И цикл foreach выводить все элементы массива в строку:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Но что если мы хотим отдельно пробежаться по каждой строке в таблице? В этом случае надо получить количество элементов в размерности. В частности, у каждого массива есть метод GetUpperBound(dimension), который возвращает индекс последнего элемента в определенной размерности. И если мы говорим непосредственно о двухмерном массиве, то первая размерность (с индексом 0) по сути это и есть таблица. И с помощью выражения mas.GetUpperBound(0) + 1 можно получить количество строк таблицы, представленной двухмерным массивом. А через mas.Length / rows можно получить количество элементов в каждой строке:

```
int[,] mas = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };

int rows = mas.GetUpperBound(0) + 1;
int columns = mas.Length / rows;
// или так
// int columns = mas.GetUpperBound(1) + 1;

for (int i = 0; i < rows; i++)
{
    for (int j = 0; j < columns; j++)
    {
        Console.Write($"{mas[i, j]} \t");
    }
    Console.WriteLine();
}
```

```
1       2       3
4       5       6
7       8       9
10      11      12
```

Массив массивов

От многомерных массивов надо отличать массив массивов или так называемый "зубчатый массив":

```
int[][] nums = new int[3][];
nums[0] = new int[2] { 1, 2 };          // выделяем память для первого подмассива
nums[1] = new int[3] { 1, 2, 3 };       // выделяем память для второго подмассива
nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива
```

Здесь две группы квадратных скобок указывают, что это массив массивов, то есть такой массив, который в свою очередь содержит в себе другие массивы. Причем длина массива указывается только в первых квадратных скобках, все последующие квадратные скобки должны быть пусты: new int[3][]. В данном случае у нас массив nums содержит три массива. Причем размерность каждого из этих массивов может не совпадать.

Зубчатый массив nums

Примеры массивов:

Причем мы можем использовать в качестве массивов и многомерные:

```
int[][,] nums = new int[3][,]
{
    new int[,] { {1,2}, {3,4} },
    new int[,] { {1,2}, {3,6} },
    new int[,] { {1,2}, {3,5}, {8, 13} }
};
```

Так здесь у нас массив из трех массивов, причем каждый из этих массивов представляет двухмерный массив.

Используя вложенные циклы, можно перебирать зубчатые массивы. Например:

```
int[][] numbers = new int[3][];
numbers[0] = new int[] { 1, 2 };
numbers[1] = new int[] { 1, 2, 3 };
numbers[2] = new int[] { 1, 2, 3, 4, 5 };
foreach(int[] row in numbers)
{
    foreach(int number in row)
    {
        Console.Write($"{number} \t");
    }
    Console.WriteLine();
}

// перебор с помощью цикла for
for (int i = 0; i<numbers.Length;i++)
{
    for (int j =0; j<numbers[i].Length; j++)
    {
        Console.Write($"{numbers[i][j]} \t");
    }
    Console.WriteLine();
}
```

Основные понятия массивов

Суммирую основные понятия массивов:

- Ранг (rank): количество измерений массива
- Длина измерения (dimension length): длина отдельного измерения массива
- Длина массива (array length): количество всех элементов массива

Например, возьмем массив

```
int[,] numbers = new int[3, 4];
```

Массив numbers двухмерный, то есть он имеет два измерения, поэому его ранг равен 2. Длина первого измерения - 3, длина второго измерения - 4. Длина массива (то есть общее количество элементов) - 12.

Задачи с массивами

Рассмотрим пару задач для работы с массивами.

Найдем количество положительных чисел в массиве:

```
int[] numbers = { -4, -3, -2, -1, 0, 1, 2, 3, 4 };
int result = 0;
foreach(int number in numbers)
{
    if(number > 0)
    {
        result++;
    }
}
Console.WriteLine($"Число элементов больше нуля: {result}");
```

Вторая задача - инверсия массива, то есть переворот его в обратном порядке:

```
int[] numbers = { -4, -3, -2, -1,0, 1, 2, 3, 4 };

int n = numbers.Length; // длина массива
int k = n / 2;          // середина массива
int temp;               // вспомогательный элемент для обмена значениями
for(int i=0; i < k; i++)
{
    temp = numbers[i];
    numbers[i] = numbers[n - i - 1];
    numbers[n - i - 1] = temp;
}
foreach(int i in numbers)
{
    Console.Write($"{i} \t");
}
```

Поскольку нам надо изменять элементы массива, то для этого используется цикл for. Алгоритм решения задачи подразумевает перебор элементов до середины массива, которая в программе представлена переменной k, и обмен значений элемента, который имеет индекс i, и элемента с индексом n-i-1.

Программа сортировки массива

Познакомившись с циклами, переменными, условными конструкциями и массивами, вполне уже можно писать примитивные программы. И сейчас мы одну из них напишем в целях тренировочного процесса. Это будет простейшая программа сортировки массива.

Создадим новое консольное приложение. И изменим код файла Program.cs на следующий:

```
using System;
namespace SortApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // ввод чисел
            int[] nums = new int[7];
            Console.WriteLine("Введите семь чисел");
            for (int i = 0; i < nums.Length; i++)
            {
                Console.Write("{0}-е число: ", i + 1);
                nums[i] = Int32.Parse(Console.ReadLine());
            }

            // сортировка
            int temp;
            for (int i = 0; i < nums.Length-1; i++)
            {
                for (int j = i + 1; j < nums.Length; j++)
                {
```

```
                if (nums[i] > nums[j])
                {
                    temp = nums[i];
                    nums[i] = nums[j];
                    nums[j] = temp;
                }
            }
        }

        // вывод
        Console.WriteLine("Вывод отсортированного массива");
        for (int i = 0; i < nums.Length; i++)
        {
            Console.WriteLine(nums[i]);
        }
        Console.ReadLine();
    }
  }
}
```

Вся программа условно поделена на три блока: ввод чисел, сортировку и вывод отсортированного массива. Здесь используются все те же конструкции, что были рассмотрены ранее. Сначала в цикле мы вводим все числа для массива. Так как метод Console.ReadLine() возвращает вводимую строку, а нам нужны числа, поэтому мы эту строку переводим в число с помощью метода Int32.Parse(Console.ReadLine()).

Затем сортируем: выполняем проходы по массиву и сравниваем элементы. Если элемент с меньшим индексом больше элемента с большим индексом, то меняем элементы местами.

В конце выводим все элементы.

Методы

Если переменные хранят некоторые значения, то методы содержат собой набор операторов, которые выполняют определенные действия. По сути метод - это именованный блок кода, который выполняет некоторые действия.

Общее определение методов выглядит следующим образом:

```
[модификаторы] тип_возвращаемого_значения название_метода ([параметры])
{
   // тело метода
}
```

Модификаторы и параметры необязательны.

Например, по умолчанию консольная программа на языке C# должна содержать как минимум один метод - метод Main, который является точкой входа в приложение:

```
static void Main(string[] args)
{

}
```

Ключевое слово static является модификатором. Далее идет тип возвращаемого значения. В данном случае ключевое слово void указывает на то, что метод ничего не возвращает.

Далее идет название метода - Main и в скобках параметры - string[] args. И в фигурные скобки заключено тело метода - все действия, которые он выполняет. В данном случае метод Main пуст, он не содержит никаких операторов и по сути ничего не выполняет.

Определим еще пару методов:

```
using System;

namespace HelloApp
{
   class Program
   {
      static void Main(string[] args)
      {

      }

      static void SayHello()
      {
```

```
            Console.WriteLine("Hello");
        }
        static void SayGoodbye()
        {
            Console.WriteLine("GoodBye");
        }
    }
}
```

В данном случае определены еще два метода: SayHello и SayGoodbye. Оба метода определены в рамках класса Program, они имеют модификатор static, а в качестве возвращаемого типа для них определен тип void. То есть данные методы ничего не возвращают, просто производят некоторые действия. И также оба метода не имеют никаких параметров, поэтому после названия метода указаны пустые скобки.

Оба метода выводят на консоль некоторую строку. Причем для вывода на консоль методы используют другой метод, который определен в .NET по умолчанию - Console.WriteLine().

Но если мы запустим даную программу, то мы не увидим никаких сообщений, которые должны выводить методы SayHello и SayGoodbye. Потому что стартовой точкой является метод Main. При запуске программы выполняется только метод Main и все операторы, которые составляют тело этого метода. Все остальные методы не выполняются.

Вызов методов

Чтобы использовать методы SayHello и SayGoodbye в программе, нам надо вызвать их в методе Main.

Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры).

```
название_метода (значения_для_параметров_метода);
```

Например, вызовем методы SayHello и SayGoodbye:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            SayHello();
            SayGoodbye();

            Console.ReadKey();
        }

        static void SayHello()
        {
            Console.WriteLine("Hello");
        }
        static void SayGoodbye()
        {
            Console.WriteLine("GoodBye");
        }
    }
}
```

Консольный вывод программы:

```
Hello
GoodBye
```

Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы. Например, в примере выше в двух методах для вывода строки на консоль используется метод Console.WriteLine.

Возвращение значения

Метод может возвращать значение, какой-либо результат. В примере выше были определены два метода, которые имели тип void. Методы с таким типом не возвращают никакого значения. Они просто выполняют некоторые действия.

Если метод имеет любой другой тип, отличный от void, то такой метод обязан вернуть значение этого типа. Для этого применяется оператор return, после которого идет возвращаемое значение:

```
return возвращаемое значение;
```

Например, определим еще пару методов:

```
static string GetHello()
{
    return "Hello";
}
static int GetSum()
{
    int x = 2;
    int y = 3;
    int z = x + y;
    return z;
}
```

Метод GetHello имеет тип string, следовательно, он должен возвратить строку. Поэтому в теле метода используется оператор return, после которого указана возвращаемая строка.

Метод GetSum имеет тип int, следовательно, он должен возвратить значение типа int - целое число. Поэтому в теле метода используется оператор return, после которого указано возвращаемое число (в данном случае результат суммы переменных x и y).

После оператора return также можно указывать сложные выражения, которые возвращают определенный результат. Например:

```
static int GetSum()
{
    int x = 2;
    int y = 3;
    return x + y;
}
```

При этом методы, которые в качестве возвращаемого типа имеют любой тип, отличный от void, обязательно должны использовать оператор return для возвращения значения. Например, следующее определение метода некорректно:

```
static string GetHello()
{
    Console.WriteLine("Hello");
}
```

Также между возвращаемым типом метода и возвращаемым значением после оператора return должно быть соответствие. Например, в следующем случае возвращаемый тип - int, но метод возвращает строку (тип string), поэтому такое определение метода некорректно:

```
static int GetSum()
{
    int x = 2;
    int y = 3;
    return "5"; // ошибка - надо возвращать число
}
```

Результат методов, который возвращают значение, мы можем присвоить переменным или использовать иным образом в программе:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string message = GetHello();
            int sum = GetSum();

            Console.WriteLine(message);  // Hello
            Console.WriteLine(sum);     // 5

            Console.ReadKey();
        }
```

```
    static string GetHello()
    {
        return "Hello";
    }
    static int GetSum()
    {
        int x = 2;
        int y = 3;
        return x + y;
    }
}
}
```

Метод GetHello возвращает значение типа string. Поэтому мы можем присвоить это значение какой-нибудь переменной типа string: `string message = GetHello();`

Второй метод - GetSum - возвращает значение типа int, поэтому его можно присвоить переменной, которая принимает значение этого типа: `int sum = GetSum();`.

Выход из метода

Оператор return не только возвращает значение, но и производит выход из метода. Поэтому он должен определяться после отальных инструкций. Например:

```
static string GetHello()
{
    return "Hello";
    Console.WriteLine("After return");
}
```

С точки зрения синтаксиса данный метод корректен, однако его инструкция `Console.WriteLine("After return")` не имеет смысла - она никогда не выполнится, так как до ее выполнения оператор return возвратит значение и произведет выход из метода.

Однако мы можем использовать оператор return и в методах с типам void. В этом случае после оператора return не ставится никакого возвращаемого значения (ведь метод ничего не возвращает). Типичная ситуация - в зависимости от опеределенных условий произвести выход из метода:

```
static void SayHello()
{
    int hour = 23;
    if(hour > 22)
    {
        return;
    }
    else
    {
        Console.WriteLine("Hello");
    }
}
```

Сокращенная запись методов

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода. Например, допустим у нас есть метод:

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

Мы можемего сократить следующим образом:

```
static void SayHello() => Console.WriteLine("Hello");
```

То есть после списка параметров ставится знак равно и больше чем, после которого идет выполняемая инструкция.

Подобным образом мы можем сокращать методы, которые возвращают значение:

```
static string GetHello()
{
    return "hello";
}
```

Анлогичен следующему методу:

```
static string GetHello() => "hello";
```

Параметры методов

Параметры позволяют передать в метод некоторые входные данные. Например, определеим метод, который складывает два числа:

```
static int Sum(int x, int y)
{
   return x + y;
}
```

Метод Sum имеет два параметра: x и y. Оба параметра представляют тип int. Поэтому при вызове данного метода нам обязательно надо передать на место этих параметров два числа.

```
class Program
{
   static void Main(string[] args)
   {
      int result = Sum(10, 15);
      Console.WriteLine(result);  // 25

      Console.ReadKey();
   }
   static int Sum(int x, int y)
   {
      return x + y;
   }
}
```

При вызове метода Sum значения передаются параметрам по позиции. Например, в вызове Sum(10, 15) число 10 передается параметру x, а число 15 - параметру y. Значения, которые передаются параметрам, еще называются аргументами. То есть передаваемые числа 10 и 15 в данном случае являются аргументами.

Иногда можно встретить такие определения как формальные параметры и фактические параметры. Формальные параметры - это собственно параметры метода (в данном случае x и y), а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают некоторое значение:

```
class Program
{
   static void Main(string[] args)
   {
      int a = 25;
      int b = 35;
      int result = Sum(a, b);
      Console.WriteLine(result);  // 60

      result = Sum(b, 45);
      Console.WriteLine(result);  // 80

      result = Sum(a + b + 12, 18); // "a + b + 12" представляет значение параметра x
      Console.WriteLine(result);  // 90

      Console.ReadKey();
   }
   static int Sum(int x, int y)
   {
      return x + y;
   }
}
```

Если параметрами метода передаются значения переменных, которые представляют базовые примитивные типы (за исключением типа object), то таким переменным должно быть присвоено значение. Например, следующая программа не скомпилируется:

```
class Program
{
   static void Main(string[] args)
   {
      int a;
```

```
        int b = 9;
        Sum(a, b);  // Ошибка - переменной a не присвоено значение

        Console.ReadKey();
    }
    static int Sum(int x, int y)
    {
        return x + y;
    }
}
```

При передаче значений параметрам важно учитывать тип параметров: между аргументами и параметрами должно быть соответствие по типу. Например:

```
class Program
{
    static void Main(string[] args)
    {
        Display("Tom", 24); // Name: Tom  Age: 24

        Console.ReadKey();
    }
    static void Display(string name, int age)
    {
        Console.WriteLine($"Name: {name}  Age: {age}");
    }
}
```

В данном случае первый параметр метода Display представляет тип string, поэтому мы должны передать этому параметру значение типа string, то есть строку. Второй параметр представляет тип int, поэтому должны передать ему целое число, которое соответствует типу int.

Другие данные параметрам мы передать не можем. Например, следующий вызов метода Display будет ошибочным:

```
Display(45, "Bob"); // Ошибка! несоответствие значений типам параметров
```

Необязательные параметры

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

```
static int OptionalParam(int x, int y, int z=5, int s=4)
{
    return x + y + z + s;
}
```

Так как последние два параметра объявлены как необязательные, то мы можем один из них или оба опустить:

```
static void Main(string[] args)
{
    OptionalParam(2, 3);

    OptionalParam(2,3,10);

    Console.ReadKey();
}
```

Именованные параметры

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. Но мы можем нарушить подобный порядок, используя именованные параметры:

```
static int OptionalParam(int x, int y, int z=5, int s=4)
{
    return x + y + z + s;
}
static void Main(string[] args)
{
    OptionalParam(x:2, y:3);
```

```
    //Необязательный параметр z использует значение по умолчанию
    OptionalParam(y:2, x:3, s:10);

    Console.ReadKey();
}
```

Передача параметров по ссылке и значению. Выходные параметры

Существует два способа передачи параметров в метод в языке C#: по значению и по ссылке.

Передача параметров по значению

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров:

```
class Program
{
    static void Main(string[] args)
    {
        Sum(10, 15);        // параметры передаются по значению
        Console.ReadKey();
    }
    static int Sum(int x, int y)
    {
        return x + y;
    }
}
```

Передача параметров по ссылке и модификатор ref

При передаче параметров по ссылке перед параметрами используется модификатор ref:

```
static void Main(string[] args)
{
    int x = 10;
    int y = 15;
    Addition(ref x, y); // вызов метода
    Console.WriteLine(x);   // 25

    Console.ReadLine();
}
// параметр x передается по ссылке
static void Addition(ref int x, int y)
{
    x += y;
}
```

Обратите внимание, что модификатор ref указывается, как при объявлении метода, так и при его вызове в методе Main.

Сравнение передачи по значению и по ссылке

В чем отличие двух способов передачи параметров? При передаче по значению метод получает не саму переменную, а ее копию. А при передаче параметра по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место.

Рассмотрим два аналогичных примера. Певый пример - передача параметра по значению:

```
class Program
{
    static void Main(string[] args)
    {
        int a = 5;
        Console.WriteLine($"Начальное значение переменной a = {a}");

        //Передача переменных по значению
        //После выполнения этого кода по-прежнему a = 5, так как мы передали лишь ее копию
        IncrementVal(a);
        Console.WriteLine($"Переменная a после передачи по значению равна = {a}");
        Console.ReadKey();
    }
```

```
    // передача по значению
    static void IncrementVal(int x)
    {
        x++;
        Console.WriteLine($"IncrementVal: {x}");
    }
}
```

Консольный вывод:

```
Начальное значение переменной a = 5
IncrementVal: 6
Переменная a после передачи по значению равна = 5
```

При вызове метод IncrementVal получает копию переменной a и увеличивает значение этой копии. Поэтому в самом методе IncrementVal мы видим, что значение параметра x увеличилось на 1, но после выполнения метода переменная a имеет прежнее значение - 5. То есть изменяется копия, а сама переменная не изменяется.

Второй пример - аналогичный метод с передачей параметра по ссылке:

```
class Program
{
    static void Main(string[] args)
    {
        int a = 5;
        Console.WriteLine($"Начальное значение переменной a  = {a}");
        //Передача переменных по ссылке
        //После выполнения этого кода a = 6, так как мы передали саму переменную
        IncrementRef(ref a);
        Console.WriteLine($"Переменная a после передачи ссылке равна = {a}");

        Console.ReadKey();
    }
    // передача по ссылке
    static void IncrementRef(ref int x)
    {
        x++;
        Console.WriteLine($"IncrementRef: {x}");
    }
}
```

Консольный вывод:

```
Начальное значение переменной a = 5
IncrementRef: 6
Переменная a после передачи по ссылке равна = 6
```

В метод IncrementRef передается ссылка на саму переменную a в памяти. И если значение параметра в IncrementRef изменяется, то это приводит и к изменению переменной a, так как и параметр и переменная указывают на один и тот же адрес в памяти.

Модификатор out

Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор out:

```
static void Sum(int x, int y, out int a)
{
    a = x + y;
}
```

Здесь результат возвращается не через оператор return, а через выходной параметр. Использование в программе:

```
static void Main(string[] args)
{
    int x = 10;

    int z;

    Sum(x, 15, out z);

    Console.WriteLine(z);

    Console.ReadKey();
}
```

Причем, как и в случае с ref ключевое слово out используется как при определении метода, так и при его вызове.
Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение. То есть следующий код будет недопустим, так как в нем для out-параметра не указано никакого значения:
```
static void Sum(int x, int y, out int a)
{
    Console.WriteLine(x+y);
}
```
Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не один вариант, а несколько. Например:
```
static void Main(string[] args)
{
    int x = 10;
    int area;
    int perimetr;
    GetData(x, 15, out area, out perimetr);
    Console.WriteLine("Площадь : " + area);
    Console.WriteLine("Периметр : " + perimetr);

    Console.ReadKey();
}
static void GetData(int x, int y, out int area, out int perim)
{
    area= x * y;
    perim= (x + y)*2;
}
```
Здесь у нас есть метод GetData, который, допустим, принимает стороны прямоугольника. А два выходных параметра мы используем для подсчета площади и периметра прямоугольника. По сути, как и в случае с ключевым словом ref, ключевое слово out применяется для для передачи аргументов по ссылке. Однако в отличие от ref для переменных, которые передаются с ключевым словам out, не требуется инициализация. И кроме того, вызываемый метод должен обязательно присвоить им значение.
Стоит отметить, что начиная с версии C# 7.0 можно определять переменные в непосредственно при вызове метода. То есть вместо:
```
int x = 10;
int area;
int perimetr;
GetData(x, 15, out area, out perimetr);
Console.WriteLine($"Площадь : {area}");
Console.WriteLine($"Периметр : {perimetr}");
```
Мы можем написать:
```
int x = 10;
GetData(x, 15, out int area, out int perimetr);
Console.WriteLine($"Площадь : {area}");
Console.WriteLine($"Периметр : {perimetr}");
```

Массив параметров и ключевое слово params

Во всех предыдущих примерах мы использовали постоянное число параметров. Но, используя ключевое слово params, мы можем передавать неопределенное количество параметров:
```
static void Addition(params int[] integers)
{
    int result = 0;
    for (int i = 0; i < integers.Length; i++)
    {
        result += integers[i];
    }
    Console.WriteLine(result);
}

static void Main(string[] args)
{
    Addition(1, 2, 3, 4, 5);
```

```
    int[] array = new int[] { 1, 2, 3, 4 };
    Addition(array);

    Addition();
    Console.ReadLine();
}
```

Причем, как видно из примера, на место параметра с модификатором params мы можем передать как отдельные значения, так и массив значений, либо вообще не передавать параметры.

Если же нам надо передать какие- то другие параметры, то они должны указываться до параметра с ключевм словом params:

```
//Так работает
static void Addition( int x, string mes, params int[] integers)
{}
```

Вызов подобного метода:

```
Addition(2, "hello", 1, 3, 4);
```

Однако после параметра с модификатором params мы НЕ можем указывать другие параметры. То есть следующее определение метода недопустимо:

```
//Так НЕ работает
static void Addition(params int[] integers, int x, string mes)
{}
```

Массив в качестве параметра

Также этот способ передачи параметров надо отличать от передачи массива в качестве параметра:

```
// передача параметра с params
static void Addition(params int[] integers)
{
    int result = 0;
    for (int i = 0; i < integers.Length; i++)
    {
        result += integers[i];
    }
    Console.WriteLine(result);
}
// передача массива
static void AdditionMas(int[] integers, int k)
{
    int result = 0;
    for (int i = 0; i < integers.Length; i++)
    {
        result += (integers[i]*k);
    }
    Console.WriteLine(result);
}

static void Main(string[] args)
{
    Addition(1, 2, 3, 4, 5);

    int[] array = new int[] { 1, 2, 3, 4 };
    AdditionMas(array, 2);

    Console.ReadLine();
}
```

Так как метод AdditionMas принимает в качестве параметра массив без ключевого слова params, то при его вызове нам обязательно надо передать в качестве параметра массив.

Область видимости (контекст) переменных

Каждая переменная доступна в рамках определенного контекста или области видимость. Вне этого контекста переменная уже не существует.

Существуют различные контексты:

- Контекст класса. Переменные, определенные на уровне класса, доступны в любом методе этого класса
- Контекст метода. Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны
- Контекст блока кода. Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны.

Например, пусть класс Program определен следующим образом:

```
class Program // начало контекста класса
{
   static int a = 9; // переменная уровня класса

   static void Main(string[] args) // начало контекста метода Main
   {
      int b = a - 1; // переменная уровня метода

      { // начало контекста блока кода

         int c = b - 1; // переменная уровня блока кода

      }  // конец контекста блока кода, переменная c уничтожается

      //так нельзя, переменная c определена в блоке кода
      //Console.WriteLine(c);

      //так нельзя, переменная d определена в другом методе
      //Console.WriteLine(d);

      Console.Read();

   } // конец контекста метода Main, переменная b уничтожается

   void Display() // начало контекста метода Display
   {
      // переменная a определена в контексте класса, поэтому доступна
      int d = a + 1;

   } // конец конекста метода Display, переменная d уничтожается

} // конец контекста класса, переменная a уничтожается
```

Здесь определенно четыре переменных: a, b, c, d. Каждая из них существует в своем контексте. Переменная a существует в контексте всего класса Program и доступна в любом месте и блоке кода в методах Main и Display.

Переменная b существует только в рамках метода Main. Также как и переменная d существует в рамках метода Display. В методе Main мы не можем обратиться к переменной d, так как она в другом контексте.

Переменная c существует только в блоке кода, границами которого являются открывающая и закрывающая фигурные скобки. Вне его границ переменная c не существует и к ней нельзя обратиться.

Нередко границы различных контекстов можно ассоциировать с открывающимися и закрывающимися фигурными скобками, как в данном случае, которые задают пределы блока кода, метода, класса.

При работе с переменными надо учитывать, что локальные переменные, определенные в методе или в блоке кода, скрывают переменные уровня класса, если их имена совпадают:

```
class Program
{
   static int a = 9; // переменная уровня класса

   static void Main(string[] args)
   {
      int a = 5; // скрывает переменную a, которая объявлена на уровне класса
      Console.WriteLine(a); // 5
```

```
    }
}
```
При объявлении переменных также надо учитывать, что в одном контексте нельзя определить несколько переменных с одним и тем же именем.

Рекурсивные функции

Отдельно остановимся на рекурсивных функциях. Рекурсивная функция представляет такую конструкцию, при которой функция вызывает саму себя.

Возьмем, к примеру, функцию, вычисляющую факториал числа:

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

Итак, здесь у нас задается условие, что если вводимое число не равно 0, то мы умножаем данное число на результат этой же функции, в которую в качестве параметра передается число x-1. То есть происходит рекурсивный спуск. И так, пока не дойдем того момента, когда значение параметра не будет равно единице.

При создании рекурсивной функции в ней обязательно должен быть некоторый базовый вариант, который использует оператор return и помещается в начале функции. В случае с факториалом это `if (x == 0) return 1;`.

И, кроме того, все рекурсивные вызовы должны обращаться к подфункциям, которые в конце концов сходятся к базовому варианту. Так, при передаче в функцию положительного числа при дальнейших рекурсивных вызовах подфункций в них будет передаваться каждый раз число, меньшее на единицу. И в конце концов мы дойдем до ситуации, когда число будет равно 0, и будет использован базовый вариант.

Другим распространенным показательным примером рекурсивной функции служит функция, вычисляющая числа Фиббоначчи. n-й член последовательности Фибоначчи определяется по формуле: f(n)=f(n-1) + f(n-2), причем f(0)=0, а f(1)=1.

```
static int Fibonachi(int n)
{
    if (n == 0)
    {
        return 0;
    }
    if (n == 1)
    {
        return 1;
    }
    else
    {
        return Fibonachi(n - 1) + Fibonachi(n - 2);
    }
}
```

Перечисления enum

Кроме примитивных типов данных в C# есть такой тип как enum или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum. Далее идет название перечисления, после которого указывается тип перечисления - он обязательно должен представлять целочисленный тип (byte, int, short, long). Если тип явным образом не указан, то по умолчанию используется тип int. Затем идет список элементов перечисления через запятую:

```
enum Days
{
    Monday,
```

```
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

enum Time : byte
{
    Morning,
    Afternoon,
    Evening,
    Night
}
```

В этих примерах каждому элементу перечисления присваивается целочисленное значение, причем первый элемент будет иметь значение 0, второй - 1 и так далее. Мы можем также явным образом указать значения элементов, либо указав значение первого элемента:

```
enum Operation
{
    Add = 1,   // каждый следующий элемент по умолчанию увеличивается на единицу
    Subtract, // этот элемент равен 2
    Multiply, // равен 3
    Divide    // равен 4
}
```

Но можно и для всех элементов явным образом указать значения:

```
enum Operation
{
    Add = 2,
    Subtract = 4,
    Multiply = 8,
    Divide = 16
}
```

При этом контанты перечисления могут иметь одинаковые значения, либо даже можно присваивать одной константе значение другой константы:

```
enum Color
{
    White = 1,
    Black = 2,
    Green = 2,
    Blue = White // Blue = 1
}
```

Каждое перечисление фактически определяет новый тип данных. Затем в программе мы можем определить переменную этого типа и использовать ее:

```
enum Operation
    {
        Add = 1,
        Subtract,
        Multiply,
        Divide
    }
class Program
{
    static void Main(string[] args)
    {
        Operation op;
        op = Operation.Add;
        Console.WriteLine(op); // Add

        Console.ReadLine();
    }
}
```

В программе мы можем присвоить значение этой переменной. При этом в качестве ее значения должна выступать одна из констант, определенных в данном перечислении. То есть несмотря на то, что каждая константа сопоставляется с определенным числом, мы не можем присвоить

ей числовое значение, например, Operation op = 1;. И также если мы будем выводить на консоль значение этой переменной, то мы получим им константы, а не числовое значение. Если же необходимо получить числовое значение, то следует выполнить приведение к числовому типу:

```
Operation op;
op = Operation.Multiply;
Console.WriteLine((int)op); // 3
```

Также стоит отметить, что перечисление необязательно определять внутри класса, можно и вне класса, но в пределах пространства имен:

```
enum Operation
{
    Add = 1,
    Subtract,
    Multiply,
    Divide
}
class Program
{
    static void Main(string[] args)
    {
        Operation op;
        op = Operation.Add;
        Console.WriteLine(op); // Add

        Console.ReadLine();
    }
}
```

Зачастую переменная перечисления выступает в качестве хранилища состояния, в зависимости от которого производятся некоторые действия. Так, рассмотрим применение перечисления на более реальном примере:

```
class Program
{
    enum Operation
    {
        Add = 1,
        Subtract,
        Multiply,
        Divide
    }

    static void MathOp(double x, double y, Operation op)
    {
        double result = 0.0;

        switch (op)
        {
            case Operation.Add:
                result = x + y;
                break;
            case Operation.Subtract:
                result = x - y;
                break;
            case Operation.Multiply:
                result = x * y;
                break;
            case Operation.Divide:
                result = x / y;
                break;
        }

        Console.WriteLine("Результат операции равен {0}", result);
    }

    static void Main(string[] args)
    {
        // Тип операции задаем с помощью константы Operation.Add, которая равна 1
```

```
        MathOp(10, 5, Operation.Add);
        // Тип операции задаем с помощью константы Operation.Multiply, которая равна 3
        MathOp(11, 5, Operation.Multiply);

        Console.ReadLine();
    }
}
```
Здесь у нас имеется перечисление Operation, которое представляет арифметические операции. Также у нас определен метод MathOp, который в качестве параметров принимает два числа и тип операции. В основном методе Main мы два раза вызываем процедуру MathOp, передав в нее два числа и тип операции.

Кортежи

Кортежи предоставляют удобный способ для работы с набором значений, который был добавлен в версии C# 7.0.

Кортеж представляет набор значений, заключенных в круглые скобки:

```
var tuple = (5, 10);
```
В данном случае определен кортеж tuple, который имеет два значения: 5 и 10. В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями Item[порядковый_номер_поля_в_кортеже]. Например:
```
static void Main(string[] args)
{
    var tuple = (5, 10);
    Console.WriteLine(tuple.Item1); // 5
    Console.WriteLine(tuple.Item2); // 10
    tuple.Item1 += 26;
    Console.WriteLine(tuple.Item1); // 31
    Console.Read();
}
```
В данном случае тип определяется неявно. Но мы ткже можем явным образом указать для переменной кортежа тип:
```
(int, int) tuple = (5, 10);
```
Так как кортеж содержит два числа, то в определении типа нам надо указать два числовых типа. Или другой пример определения кортежа:
```
(string, int, double) person = ("Tom", 25, 81.23);
```
Первый элемент кортежа в данном случае представляет строку, второй элемент - тип int, а третий - тип double.

Мы также можем дать названия полям кортежа:
```
var tuple = (count:5, sum:10);
Console.WriteLine(tuple.count); // 5
Console.WriteLine(tuple.sum); // 10
```
Теперь чтобы обратиться к полям кортежа используются их имена, а не названия Item1 и Item2.

Мы даже можем не использовать переменную для определения всего кортежа, а использовать отдельные переменные для его полей:
```
static void Main(string[] args)
{
    var (name, age) = ("Tom", 23);
    Console.WriteLine(name);    // Tom
    Console.WriteLine(age);     // 23
    Console.Read();
}
```
В этом случае с полями кортежа мы сможем работать как с переменными, которые определены в рамках метода.

Использование кортежей

Кортежи могут передаваться в качестве параметров в метод, могут быть возвращаемым результатом функции, либо использоваться иным образом.

Например, одной из распространенных ситуаций является возвращение из функции двух и более значений, в то время как функция можно возвращать только одно значение. И кортежи представляют оптимальный способ для решения этой задачи:
```
static void Main(string[] args)
{
```

```
    var tuple = GetValues();
    Console.WriteLine(tuple.Item1); // 1
    Console.WriteLine(tuple.Item2); // 3

    Console.Read();
}
private static (int, int) GetValues()
{
    var result = (1, 3);
    return result;
}
```

Здесь определен метод GetValues(), который возвращает кортеж. Кортеж определяется как набор значений, помещенных в круглые скобки. И в данном случае мы возвращаем кортеж из двух элементов типа int, то есть два числа.

Другой пример:

```
static void Main(string[] args)
{
    var tuple = GetNamedValues(new int[]{ 1,2,3,4,5,6,7});
    Console.WriteLine(tuple.count);
    Console.WriteLine(tuple.sum);

    Console.Read();
}
private static (int sum, int count) GetNamedValues(int[] numbers)
{
    var result = (sum:0, count: 0);
    for (int i=0; i < numbers.Length; i++)
    {
        result.sum += numbers[i];
        result.count++;
    }
    return result;
}
```

И также кортеж может передаваться в качестве параметра в метод:

```
static void Main(string[] args)
{
    var (name, age) = GetTuple(("Tom", 23), 12);
    Console.WriteLine(name);    // Tom
    Console.WriteLine(age);     // 35
    Console.Read();
}

private static (string name, int age) GetTuple((string n, int a) tuple, int x)
{
    var result = (name: tuple.n, age: tuple.a + x);
    return result;
}
```

Классы. Объектно-ориентированное программирование
Классы и объекты
C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.
Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.
По умолчанию проект консольного приложения уже по умолчанию содержит один класс Program, с которого и начинается выполнение программы.
По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова class:
```
class Person
{

}
```
Где определяется класс? Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Но в данном случае поместим новый класс в файле, где располагается класс Program. То есть файл Program.cs будет выглядеть следующим образом:
```
using System;

namespace HelloApp
{
    class Person
    {

    }
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```
Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Например, определим в классе Person поля и метод:
```
using System;

namespace HelloApp
{
    class Person
    {
        public string name; // имя
        public int age;     // возраст

        public void GetInfo()
        {
            Console.WriteLine($"Имя: {name}  Возраст: {age}");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person tom;

            Console.ReadKey();
        }
```

```
    }
}
```
В данном случае класс Person представляет человека. Поле name хранит имя, а поле age - возраст человека. А метод GetInfo выводит все данные на консоль. Чтобы все данные были доступны вне класса Person переменные и метод определены с модификатором public.

Поскольку класс представляет собой новый тип, то в программе мы можем определять переменные, которые представляют данный тип. Так, здесь в методе Main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение null. Поэтому вначале необходимо создать объект класса Person.

Конструкторы

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Конструктор по умолчанию

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет парамтров и не имеет тела. Выше класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию. И мы можем использовать этот конструктор. В частности, создадим один объект класса Person:

```
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Person();
        tom.GetInfo();      // Имя: Возраст: 0

        tom.name = "Tom";
        tom.age = 34;
        tom.GetInfo();  // Имя: Tom Возраст: 34

        Console.Read();
    }
}
```
Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, tom.name = "Tom";.

Консольный вывод данной программы:

```
Имя:    Возраст: 0
Имя: Tom        Возраст: 34
```

Создание конструкторов

Выше для инициализации объекта использовался конструктор по умолчанию. Однако мы сами можем определить свои конструкторы:

```
class Person
{
    public string name;
    public int age;

    public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор

    public Person(string n) { name = n; age = 18; }         // 2 конструктор

    public Person(string n, int a) { name = n; age = a; }   // 3 конструктор
```

```
    public void GetInfo()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```
Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:
```
static void Main(string[] args)
{
    Person tom = new Person();          // вызов 1-ого конструктора без параметров
    Person bob = new Person("Bob");     //вызов 2-ого конструктора с одним параметром
    Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами


    bob.GetInfo();          // Имя: Bob  Возраст: 18
    tom.GetInfo();          // Имя: Неизвестно  Возраст: 18
    sam.GetInfo();          // Имя: Sam  Возраст: 25

    Console.ReadKey();
}
```
Консольный вывод данной программы:
```
Имя: Неизвестно  Возраст: 18
Имя: Bob  Возраст: 18
Имя: Sam  Возраст: 25
```
При этом если в классе определены конструкторы, то при создании объекта необходимо использовать один из этих конструкторов.

Ключевое слово this

Ключевое слово this представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово this, передавая нужные значения для параметров:
```
class Person
{
    public string name;
    public int age;

    public Person() : this("Неизвестно")
    {
    }
    public Person(string name) : this(name, 18)
    {
    }
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void GetInfo()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```
В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:
```
public Person(string name) : this(name, 18)
{
}
```
идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
```

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.

Инициализаторы объектов

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

```
Person tom = new Person { name = "Tom", age=31 };
tom.GetInfo();          // Имя: Tom  Возраст: 31
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

При использовании инициализаторов следует учитывать следующие моменты:

- С помощью инициализатора мы можем установить значения только доступных из внешнего кода полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.
- Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

Структуры

Наряду с классами структуры представляют еще один способ создания обственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами.

Например, определим структуру, которая представляет человека:

```
struct User
{
    public string name;
    public int age;

    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}  Age: {age}");
    }
}
```

Как и классы, структуры могут хранить состояние в виде переменных и определять поведение в виде методов. Так, в данном случае определены две переменные - name и age для хранения соответственно имени и возраста человека и метод DisplayInfo для вывода информации о человеке.

Используем эту структуру в программе:

```
using System;

namespace HelloApp
{
    struct User
    {
        public string name;
        public int age;

        public void DisplayInfo()
        {
            Console.WriteLine($"Name: {name}  Age: {age}");
        }
```

```
    }

    class Program
    {
        static void Main(string[] args)
        {
            User tom;
            tom.name = "Tom";
            tom.age = 34;
            tom.DisplayInfo();

            Console.ReadKey();
        }
    }
}
```

В данном случае создается объект tom. У него устанавливаются значения глобальных переменных, и затем выводится информация о нем.

Конструкторы структуры

Как и класс, структура может определять констукторы. Но в отличие от класса нам не обязательно вызывать конструктор для создания объекта структуры:

```
User tom;
```

Однако если мы таким образом создаем объект структуры, то обязательно надо проинициализировать все поля (глобальные переменные) структуры перед получением их значений или перед вызовом методов структуры. То есть, например, в следующем случае мы получим ошибку, так как обращение к полям и методам происходит до присвоения им начальных значений:

```
User tom;
int x = tom.age;    // Ошибка
tom.DisplayInfo();  // Ошибка
```

Также мы можем использовать для создания структуры конструктор по умолчанию, при вызове которого полям структуры будет присвоено значение по умолчанию (например, для числовых типов это число 0):

```
User tom = new User();
tom.DisplayInfo();  // Name:   Age: 0
```

Также мы можем определить свои конструкторы. Например, изменим структуру User:

```
using System;
using System.Reflection;

namespace HelloApp
{
    struct User
    {
        public string name;
        public int age;
        public User(string name, int age)
        {
            this.name = name;
            this.age = age;
        }
        public void DisplayInfo()
        {
            Console.WriteLine($"Name: {name}  Age: {age}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            User tom = new User("Tom", 34);
            tom.DisplayInfo();

            User bob = new User();
            bob.DisplayInfo();
```

```
            Console.ReadKey();
        }
    }
}
```
Важно учитывать, что если мы определяем конструктор в структуре, то он должен инициализировать все поля структуры, как в данном случае устанавливаются значения для переменных name и age.
Также, как и для класса, можно использовать инициализатор для создания структуры:
```
User person = new User { name = "Sam", age = 31 };
```
Но в отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении, например, следующим образом:
```
struct User
{
    public string name = "Sam";     // ! Ошибка
    public int age = 23;            // ! Ошибка
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}  Age: {age}");
    }
}
```
Типы значений и ссылочные типы
Ранее мы рассматривали следующие элементарные типы данных: int, byte, double, string, object и др. Также есть сложные типы: структуры, перечисления, классы. Все эти типы данных можно разделить на типы значений, еще называемые значимыми типами, (value types) и ссылочные типы (reference types). Важно понимать между ними различия.
Типы значений:
- Целочисленные типы (byte, sbyte, char, short, ushort, int, uint, long, ulong)
- Типы с плавающей запятой (float, double)
- Тип decimal
- Тип bool
- Перечисления enum
- Структуры (struct)

Ссылочные типы:
- Тип object
- Тип string
- Классы (class)
- Интерфейсы (interface)
- Делегаты (delegate)

В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещаются поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.
Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.
Например:
```
class Program
{
    static void Main(string[] args)
    {
        Calculate(5);
        Console.ReadKey();
    }
```

```
    static void Calculate(int t)
    {
        int x = 6;
        int y = 7;
        int z = y + t;
    }
}
```
Пи запуске такой программы в стеке будут определяться два фрейма - для метода Main (так как он вызывается при запуске программы) и для метода Calculate:

При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, все эти переменные уничтожаются, и память в стеке очищается.
Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредсвенное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода Calculate представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.
Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.
При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, то ссылка из стека удаляется, и память очищается. После этого в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, и удаляет этот объект и очищает память.
Так, в частности, если мы изменим метод Calculate следующим образом:
```
static void Calculate(int t)
{
    object x = 6;
    int y = 7;
    int z = y + t;
}
```
То теперь значение переменной x будет храниться в куче, так как она представляет ссылочный тип object, а в стеке будет храниться ссылка на объект в куче.

Составные типы

Теперь рассмотим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:
```
class Program
{
    private static void Main(string[] args)
    {
        State state1 = new State(); // State - структура, ее данные размещены в стеке
        Country country1 = new Country(); // Country - класс, в стек помещается ссылка на адрес в хипе
                                          // а в хипе располагаются все данные объекта country1
    }
}
struct State
{
    public int x;
    public int y;
    public Country country;
}
class Country
{
    public int x;
    public int y;
}
```
Здесь в методе Main в стеке выделяется память для объекта state1. Далее в стеке создается ссылка для объекта country1 (Country country1), а с помощью вызова конструктора с ключевым

словом new выделяется место в хипе (new Country()). Ссылка в стеке для объекта country1 будет представлять адрес на место в хипе, по которому размещен данный объект..

Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект country1 в хипе. Однако в структуре State также определена переменная ссылочного типа Country. Где она будет хранить свое значение, если она определена в типе значений?

```
private static void Main(string[] args)
{
   State state1 = new State();
   state1.country = new Country();
   Country country1 = new Country();
}
```

Значение переменной state1.country также будет храниться в куче, так как эта переменная представляет ссылочный тип:

Копирование значений

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа он получает не копию объекта, а ссылку на этот объект в хипе. Например:

```
private static void Main(string[] args)
{
   State state1 = new State(); // Структура State
   State state2 = new State();
   state2.x = 1;
   state2.y = 2;
   state1 = state2;
   state2.x = 5; // state1.x=1 по-прежнему
   Console.WriteLine(state1.x); // 1
   Console.WriteLine(state2.x); // 5

   Country country1 = new Country(); // Класс Country
   Country country2 = new Country();
   country2.x = 1;
   country2.y = 4;
   country1 = country2;
   country2.x = 7; // теперь и country1.x = 7, так как обе ссылки и country1 и country2
                   // указывают на один объект в хипе
   Console.WriteLine(country1.x); // 7
   Console.WriteLine(country2.x); // 7

   Console.Read();
}
```

Так как state1 - структура, то при присвоении state1 = state2 она получает копию структуры state2. А объект класса country1 при присвоении country1 = country2; получает ссылку на тот же объект, на который указывает country2. Поэтому с изменением country2, так же будет меняться и country1.

Ссылочные типы внутри типов значений

Теперь рассмотрим более изощренный пример, когда внутри структуры у нас может быть переменная ссылочного типа, например, какого-нибудь класса:

```
class Program
{
   private static void Main(string[] args)
   {
      State state1 = new State();
      State state2 = new State();

      state2.country = new Country();
      state2.country.x = 5;
      state1 = state2;
      state2.country.x = 8; // теперь и state1.country.x=8, так как state1.country и state2.country
                            // указывают на один объект в хипе
```

```
        Console.WriteLine(state1.country.x); // 8
        Console.WriteLine(state2.country.x); // 8

        Console.Read();
    }
}
struct State
{
    public int x;
    public int y;
    public Country country;
}
class Country
{
    public int x;
    public int y;
}
```

Переменные ссылочных типов в структурах также сохраняют в стеке ссылку на объект в хипе. И при присвоении двух структур state1 = state2; структура state1 также получит ссылку на объект country в хипе. Поэтому изменение state2.country повлечет за собой также изменение state1.country.

Объекты классов как параметры методов

Организацию объектов в памяти следует учитывать при передаче параметров по значению и по ссылке. Если параметры методов представляют объекты классов, то использование параметров имеет некоторые особенности. Например, создадим метод, который в качестве параметра принимает объект Person:

```
class Program
{
    static void Main(string[] args)
    {
        Person p = new Person { name = "Tom", age=23 };
        ChangePerson(p);

        Console.WriteLine(p.name); // Alice
        Console.WriteLine(p.age); // 23

        Console.Read();
    }

    static void ChangePerson(Person person)
    {
        // сработает
        person.name = "Alice";
        // сработает только в рамках данного метода
        person = new Person { name = "Bill", age = 45 };
        Console.WriteLine(person.name); // Bill
    }
}
class Person
{
    public string name;
    public int age;
}
```

При передаче объекта класса по значению в метод передается копия ссылки на объект. Это копия указывает на тот же объект, что и исходная ссылка, потому мы можем изменить отдельные поля и свойства объекта, но не можем изменить сам объект. Поэтому в примере выше сработает только строка person.name = "Alice".

А другая строка person = new Person { name = "Bill", age = 45 } создаст новый объект в памяти, и person теперь будет указывать на новый объект в памяти. Даже если после этого мы его изменим, то это никак не повлияет на ссылку p в методе Main, поскольку ссылка p все еще указывает на старый объект в памяти.

Но при передаче параметра по ссылке (с помощью ключевого слова ref) в метод в качестве аргумента передается сама ссылка на объект в памяти. Поэтому можно изменить как поля и свойства объекта, так и сам объект:

```
class Program
{
    static void Main(string[] args)
    {
        Person p = new Person { name = "Tom", age=23 };
        ChangePerson(ref p);

        Console.WriteLine(p.name); // Bill
        Console.WriteLine(p.age); // 45

        Console.Read();
    }

    static void ChangePerson(ref Person person)
    {
        // сработает
        person.name = "Alice";
        // сработает
        person = new Person { name = "Bill", age = 45 };
    }
}
class Person
{
    public string name;
    public int age;
}
```

Операция new создаст новый объект в памяти, и теперь ссылка person (она же ссылка p из метода Main) будет указывать уже на новый объект в памяти.

Модификаторы доступа

Все члены класса - поля, методы, свойства - все они имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть контекст, в котором можно употреблять данную переменную или метод. В предыдущей теме мы уже с ними сталкивались, когда объявляли поля класса Book публичными (то есть с модификатором public).

В C# применяются следующие модификаторы доступа:

- public: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
- private: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
- protected: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
- internal: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
- protected internal: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
- private protected: такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором private. Классы, объявленные без модификатора, по умолчанию имеют доступ internal.

Все классы и структуры, определенные напрямую в пространствах имен и не являющиеся вложенными в другие классы, могут иметь только модификаторы public или internal.

Посмотрим на примере и создадим следующий класс State:

```
public class State
{
    int a; // все равно, что private int a;
    private int b; // поле доступно только из текущего класса
    protected int c; // доступно из текущего класса и производных классов
    internal int d; // доступно в любом месте программы
    protected internal int e; // доступно в любом месте программы и из классов-наследников
    public int f; // доступно в любом месте программы, а также для других программ и сборок
    protected private int g; // доступно из текущего класса и производных классов, которые определены в
том же проекте

    private void Display_f()
    {
        Console.WriteLine($"Переменная f = {f}");
    }

    public void Display_a()
    {
        Console.WriteLine($"Переменная a = {a}");
    }

    internal void Display_b()
    {
        Console.WriteLine($"Переменная b = {b}");
    }

    protected void Display_e()
    {
        Console.WriteLine($"Переменная e = {e}");
    }
}
```

Так как класс State объявлен с модификатором public, он будет доступен из любого места программы, а также из других программ и сборок. Класс State имеет пять полей для каждого уровня доступа. Плюс одна переменная без модификатора, которая является закрытой по умолчанию.

Также имеются четыре метода, которые будут выводить значения полей класса на экран. Обратите внимание, что так как все модификаторы позволяют использовать члены класса внутри данного класса, то и все переменные класса, в том числе закрытые, у нас доступны всем его методам, так как все находятся в контексте класса State.

Теперь посмотрим, как мы сможем использовать переменные нашего класса в программе (то есть в методе Main класса Program):

```
class Program
{
    static void Main(string[] args)
    {
        State state1 = new State();

        // присвоить значение переменной a у нас не получится,
        // так как она закрытая и класс Program ее не видит
        // И данную строку среда подчеркнет как неправильную

        state1.a = 4; //Ошибка, получить доступ нельзя

        // то же самое относится и к переменной b
        state1.b = 3; // Ошибка, получить доступ нельзя

        // присвоить значение переменной c то же не получится,
        // так как класс Program не является классом-наследником класса State
        state1.c = 1; // Ошибка, получить доступ нельзя

        // переменная d с модификатором internal доступна из любого места программы
        // поэтому спокойно присваиваем ей значение
        state1.d = 5;

        // переменная e так же доступна из любого места программы
```

```
        state1.e = 8;

        // переменная f общедоступна
        state1.f = 8;

        // Попробуем вывести значения переменных

        // Так как этот метод объявлен как private, мы можем использовать его только внутри класса State
        state1.Display_f() ;  // Ошибка, получить доступ нельзя

        // Так как этот метод объявлен как protected, а класс Program не является наследником класса State
        state1.Display_e();  // Ошибка, получить доступ нельзя

        // Общедоступный метод
        state1.Display_a();

        // Метод доступен из любого места программы
        state1.Display_b();

        Console.ReadLine();
    }
}
```

Таким образом, мы смогли установить только переменные d, e и f, так как их модификаторы позволяют использовать в данном контексте. И нам оказались доступны только два метода: state1.Display_a() и state1.Display_b(). Однако, так как значения переменных a и b не были установлены, то эти методы выведут нули, так как значение переменных типа int по умолчанию инициализируются нулями.

Несмотря на то, что модификаторы public и internal похожи по своему действию, но они имеют большое отличие. Классы и члены класса с модификатором public также будут доступны и другим программам, если данных класс поместить в динамическую библиотеку dll и потом ее использовать в этих программах.

Благодаря такой системе модификаторов доступа можно скрывать некоторые моменты реализации класса от других частей программы. Такое сокрытие называется инкапсуляцией.

Свойства и инкапсуляция

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют свойства. Они обеспечивают простой доступ к полям класса, узнать их значение или выполнить их установку.

Стандартное описание свойства имеет следующий синтаксис:

```
[модификатор_доступа] возвращаемый_тип произвольное_название
{
    // код свойства
}
```

Например:

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }
}
```

Здесь у нас есть закрытое поле name и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.
Через это свойство мы можем управлять доступом к переменной name. Стандартное определение свойства содержит блоки get и set. В блоке get мы возвращаем значение поля, а в блоке set устанавливаем. Параметр value представляет передаваемое значение.
Мы можем использовать данное свойство следующим образом:

```
Person p = new Person();

// Устанавливаем свойство - срабатывает блок Set
// значение "Tom" и есть передаваемое в свойство value
p.Name = "Tom";

// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get
string personName = p.Name;
```

Возможно, может возникнуть вопрос, зачем нужны свойства, если мы можем в данной ситуации обходиться обычными полями класса? Но свойства позволяют вложить дополнительную логику, которая может быть необходима, например, при присвоении переменной класса какого-либо значения. Например, нам надо установить проверку по возрасту:

```
class Person
{
    private int age;

    public int Age
    {
        set
        {
            if (value < 18)
            {
                Console.WriteLine("Возраст должен быть больше 17");
            }
            else
            {
                age = value;
            }
        }
        get { return age; }
    }
}
```

Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяют только блок get, то такое свойство доступно только для чтеня - мы можем получить его значение, но не установить. И, наоборот, если свойство имеет только блок set, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить:

```
class Person
{
    private string name;
    // свойство только для чтения
    public string Name
    {
        get
        {
            return name;
        }
    }

    private int age;
    // свойство только для записи
    public int Age
    {
        set
        {
            age = value;
```

```
        }
    }
}
```

Модификаторы доступа

Мы можем применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам - либо get, либо set:

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        private set
        {
            name = value;
        }
    }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Теперь закрытый блок set мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе, но никак не в другом классе:

```
Person p = new Person("Tom", 24);

// Ошибка - set объявлен с модификатором private
//p.Name = "John";

Console.WriteLine(p.Name);
```

При использовании модификаторов в свойствах следует учитывать ряд ограничений:

- Модификатор для блока set или get можно установить, если свойство имеет оба блока (и set, и get)
- Только один блок set или get может иметь модификатор доступа, но не оба сразу
- Модификатор доступа блока set или get должен быть более ограничивающим, чем модификатор доступа свойства. Например, если свойство имеет модификатор public, то блок set/get может иметь только модификаторы protected internal, internal, protected, private

Инкапсуляция

Выше мы посмотрели, что через свойства устанавливается доступ к приватным переменным класса. Подобное сокрытие состояния класса от вмешательства извне представляет механизм инкапсуляции, который представляет одну из ключевых концепций объектно-ориентированного программирования. (Стоит отметить, что само понятие инкапсуляции имеет довольно много различных трактовок, которые не всегда пересекаются друг с другом) Применение модификаторов доступа типа private защищает переменную от внешнего доступа. Для управления доступом во многих языках программирования используются специальные методы, геттеры и сеттеры. В C# их роль, как правило, выполняют свойства.

Например, есть некоторый класс Account, в котором определено поле sum, представляющее сумму:

```
class Account
{
    public int sum;
}
```

Поскольку переменная sum является публичной, то в любом месте программы мы можем получить к ней доступ и изменить ее, в том числе установить какое-либо недопустимое

значение, например, отрицательное. Вряд ли подобное поведение является желательным. Поэтому применяется инкапсуляция для ограничения доступа к переменной sum и сокрытию ее внутри класса:

```
class Account
{
    private int sum;
    public int Sum
    {
        get {return sum;}
        set
        {
            if (value > 0)
            {
                sum=value;
            }
        }
    }
}
```

Автоматические свойства

Свойства управляют доступом к полям класса. Однако что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в фреймворк .NET были добавлены автоматические свойства. Они имеют сокращенное объявление:

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

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

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.

В чем преимущество автосвойств, если по сути они просто обращаются к автоматически создаваемой переменной, почему бы напрямую не обратиться к переменной без автосвойств? Дело в том, что в любой момент времени при необходимости мы можем развернуть автосвойство в обычное свойство, добавить в него какую-то определенную логику.

Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

```
class Person
{
    public string Name { get; set; } = "Tom";
    public int Age { get; set; } = 23;
}

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person();
        Console.WriteLine(person.Name); // Tom
        Console.WriteLine(person.Age);  // 23

        Console.Read();
    }
}
```

И если мы не укажем для объекта Person значения свойств Name и Age, то будут действовать значения по умолчанию.

Автосвойства также могут иметь модификаторы доступа:

```
class Person
```

```
{
    public string Name { private set; get;}
    public Person(string n)
    {
        Name = n;
    }
}
```
Мы можем убрать блок set и сделать автосвойство доступным только для чтения. В этом случае для хранения значения этого свойства для него неявно будет создаваться поле с модификатором readonly, поэтому следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, как в примере выше, либо при инициализации свойства:
```
class Person
{
    public string Name { get;} = "Tom"
}
```
Сокращенная запись свойств
Как и методы, мы можем сокращать свойства. Например:
```
class Person
{
    private string name;

    // эквивалентно public string Name { get { return name; } }
    public string Name => name;
}
```
Перегрузка методов
Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется перегрузкой методов (method overloading).
И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Что такое сигнатура? Сигнатура складывается из следующих аспектов:
- Имя метода
- Количество параметров
- Типы параметров
- Порядок параметров
- Модификаторы параметров

Но названия параметров в сигнатуру НЕ входят. Например, возьмем следующий метод:
```
public int Sum(int x, int y)
{
    return x + y;
}
```
У данного метода сигнатура будет выглядеть так: Sum(int, int)
И перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:
- Количеству параметров
- Типу параметров
- Порядку параметров
- Модификаторам параметров

Например, пусть у нас есть следующий класс:
```
class Calculator
{
    public void Add(int a, int b)
    {
        int result = a + b;
        Console.WriteLine($"Result is {result}");
    }
    public void Add(int a, int b, int c)
    {
        int result = a + b + c;
        Console.WriteLine($"Result is {result}");
    }
```

```
    public int Add(int a, int b, int c, int d)
    {
        int result = a + b + c + d;
        Console.WriteLine($"Result is {result}");
        return result;
    }
    public void Add(double a, double b)
    {
        double result = a + b;
        Console.WriteLine($"Result is {result}");
    }
}
```

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.

То есть мы можем представить сигнатуры данных методов следующим образом:

```
Add(int, int)
Add(int, int, int)
Add(int, int, int, int)
Add(double, double)
```

После определения перегруженных версий мы можем использовать их в программе:

```
class Program
{
    static void Main(string[] args)
    {
        Calculator calc = new Calculator();
        calc.Add(1, 2); // 3
        calc.Add(1, 2, 3); // 6
        calc.Add(1, 2, 3, 4); // 10
        calc.Add(1.4, 2.5); // 3.9

        Console.ReadKey();
    }
}
```

Консольный вывод:

```
Result is 3
Result is 6
Result is 10
Result is 3.9
```

Также перегружаемые методы могут отличаться по используемым модификаторам. Например:

```
void Increment(ref int val)
{
    val++;
    Console.WriteLine(val);
}

void Increment(int val)
{
    val++;
    Console.WriteLine(val);
}
```

В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.

А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки. Например, возьмем следующий набор методов:

```
int Sum(int x, int y)
{
    return x + y;
}
int Sum(int number1, int number2)
{
```

```
    return x + y;
}
void Sum(int x, int y)
{
    Console.WriteLine(x + y);
}
```
Сигнатура у всех этих методов будет совпадать:
```
Sum(int, int)
```
Поэтому данный набор методов не представляет корректные перегрузки метода Sum и работать не будет.

Статические члены и модификатор static

Кроме обычных полей, методов, свойств класс может иметь статические поля, методы, свойства. Статические поля, методы, свойства относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр класса. Например:
```
class Account
{
    public static decimal bonus = 100;
    public decimal totalSum;
    public Account(decimal sum)
    {
        totalSum = sum + bonus;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Account.bonus);      // 100
        Account.bonus += 200;

        Account account1 = new Account(150);
        Console.WriteLine(account1.totalSum);   // 450


        Account account2 = new Account(1000);
        Console.WriteLine(account2.totalSum);   // 1300

        Console.ReadKey();
    }
}
```
В данном случае класс Account имеет два поля: bonus и totalSum. Поле bonus является статическим, поэтому оно хранит состояние класса в целом, а не отдельного объекта. И поэтому мы можем обращаться к этому полю по имени класса:
```
Console.WriteLine(Account.bonus);
Account.bonus += 200;
```
На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.

При этом память для статических переменных выделяется даже в том случае, если не создано ни одного объекта этого класса.

Статические свойства и методы

Подобным образом мы можем создавать и использовать статические методы и свойства:
```
class Account
{
    public Account(decimal sum, decimal rate)
    {
        if (sum < MinSum) throw new Exception("Недопустимая сумма!");
        Sum = sum; Rate = rate;
    }
    private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов
    public static decimal MinSum
```

```
    {
        get { return minSum; }
        set { if(value>0) minSum = value; }
    }

    public decimal Sum { get; private set; }    // сумма на счете
    public decimal Rate { get; private set; }   // процентная ставка

    // подсчет суммы на счете через определенный период по определенной ставке
    public static decimal GetSum(decimal sum, decimal rate, int period)
    {
        decimal result = sum;
        for (int i = 1; i <= period; i++)
            result = result + result * rate / 100;
        return result;
    }
}
```

Переменная minSum, свойство MinSum, а также метод GetSum здесь определены с ключевым словом static, то есть они являются статическими.

Переменная minSum и свойство MinSum представляют минимальную сумму, которая допустима для создания счета. Этот оказатель не относится к какому-то конкретному счету, а относится ко всем счетам в целом. Если мы изменим этот показатель для одного счета, то он также должен измениться и для другого счета. То есть в отличии от свойств Sum и Rate, которые хранят состояние объекта, переменная minSum хранит состояние для всех объектов данного класса.

То же самое с методом GetSum - он вычисляет сумму на счете через определенный период по определенной процентной ставке для определенной начальной суммы. Вызов и результат этого метода не зависит от конкретного объекта или его состояния.

Таким образом, переменные и свойства, которые хранят состояние, общее для всех объектов класса, следует определять как статические. И также методы, которые определяют общее для всех объектов поведение, также следует объявлять как статические.

Статические члены класса являются общими для всех объектов этого класса, поэтому к ним надо обращаться по имени класса:

```
Account.MinSum = 560;
decimal result = Account.GetSum(1000, 10, 5);
```

Следует учитывать, что статические методы могут обращаться только статическим членам класса. Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.

Нередко статические поля применяются для хранения счетчиков. Например, пусть у нас есть класс User, и мы хотим иметь счетчик, который позволял бы узнать, сколько объектов User создано:

```
class User
{
    private static int counter = 0;
    public User()
    {
        counter++;
    }

    public static void DisplayCounter()
    {
        Console.WriteLine($"Создано {counter} объектов User");
    }
}
class Program
{
    static void Main(string[] args)
    {
        User user1 = new User();
        User user2 = new User();
        User user3 = new User();
        User user4 = new User();
        User user5 = new User();
```

```
        User.DisplayCounter(); // 5

        Console.ReadKey();
    }
}
```

Статический конструктор

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

- Статические конструкторы не должны иметь модификатор доступа и не принимают параметров
- Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса
- Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз

Определим статический конструктор:

```
class User
{
    static User()
    {
        Console.WriteLine("Создан первый пользователь");
    }
}
class Program
{
    static void Main(string[] args)
    {
        User user1 = new User(); // здесь сработает статический конструктор
        User user2 = new User();

        Console.Read();
    }
}
```

Статические классы

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы. Например, если бы класс Account имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический:

```
static class Account
{
    private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов
    public static decimal MinSum
    {
        get { return minSum; }
        set { if(value>0) minSum = value; }
    }

    // подсчет суммы на счете через определенный период по определенной ставке
    public static decimal GetSum(decimal sum, decimal rate, int period)
    {
        decimal result = sum;
        for (int i = 1; i <= period; i++)
            result = result + result * rate / 100;
        return result;
    }
}
```

В C# показательным примером статического класса является класс Math, который применяется для различных математических операций.

Константы и поля для чтения

Полями класса называются обычные переменные уровня класса. Мы уже ранее рассматривали переменные - их объявление и инициализацию. Однако некоторые моменты мы еще не затрагивали, например, константы и поля для чтения.

Константы

Константы характеризуются следующими признаками:

- Константа должна быть проинициализирована при определении
- После определения значение константы не может быть изменено

Константы предназначены для описания таких значений, которые не должны изменяться в программе. Для определения констант используется ключевое слово const:

```
const double PI = 3.14;
const double E = 2.71;
```

При использовании констант надо помнить, что объявить мы их можем только один раз и что к моменту компиляции они должны быть определены.

```
class MathLib
{
    public const double PI=3.141;
    public const double E = 2.81;
    public const double K;      // Ошибка, константа не инициализирована
}

class Program
{
    static void Main(string[] args)
    {
        MathLib.E=3.8; // Ошибка, значение константы нельзя изменить
    }
}
```

Также обратите внимание на синтаксис обращения к константе. Так как неявно это статическое поле, для обращения к ней необходимо использовать имя класса.

```
class MathLib
{
    public const double PI=3.141;
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MathLib.PI);
    }
}
```

Но следует учитывать, что мы не можем объявить константу с модификатором static. Но в этом собственно и нет смысла.

Константу можно определить как на уровне класса, так и внутри метода:

```
class MathLib
{
    public double GetCircleArea(double radius)
    {
        const double PI = 3.141;
        return PI * radius * radius;
    }
}
```

Поля для чтения

Поля для чтения можно инициализировать при их объявлении либо на уровне класса, либо инициилизировать и изменять в конструкторе. Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

Поле для чтения объявляется с ключевым словом readonly:

```
class MathLib
{
    public readonly double K = 23;  // можно так инициализировать

    public MathLib(double _k)
```

```
    {
        K = _k; // поле для чтения может быть инициализировано или изменено в конструкторе после компиляции
    }
    public void ChangeField()
    {
        // так нельзя
        //K = 34;
    }
}

class Program
{
    static void Main(string[] args)
    {
        MathLib mathLib = new MathLib(3.8);
        Console.WriteLine(mathLib.K); // 3.8

        //mathLib.K = 7.6; // поле для чтения нельзя установить вне своего класса
        Console.ReadLine();

    }
}
```

Сравнение констант

- Константы должны быть определены во время компиляции, а поля для чтения могут быть определены во время выполнения программы.
  Соответственно инициализировать константу можно устанновить только при ее определении.
  Поле для чтения можно инициализировать либо при его определении, либо в конструкторе класса.
- Константы не могут быть статическими. Поля для чтения могут быть статическими.

Перегрузка операторов

Наряду с методами мы можем также перегружать операторы. Например, пусть у нас есть следующий класс Counter:

```
class Counter
{
    public int Value { get; set; }
}
```

Данный класс представляет некоторый счетчик, значение которого хранится в свойстве Value. И допустим, у нас есть два объекта класса Counter - два счетчика, которые мы хотим сравнивать или складывать на основании их свойства Value, используя стандартные операции сравнения и сложения:

```
Counter c1 = new Counter { Value = 23 };
Counter c2 = new Counter { Value = 45 };

bool result = c1 > c2;
Counter c3 = c1 + c2;
```

Но на данный момент ни операция сравнения, ни операция сложения для объектов Counter не доступны. Эти операции могут использоваться для ряда примитивных типов. Например, по умолчанию мы можем складывать числовые значения, но как складывать объекты комплексных типов - классов и структур компилятор не знает. И для этого нам надо выполнить перегрузку нужных нам операторов.

Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

```
public static возвращаемый_тип operator оператор(параметры)
{ }
```

Этот метод должен иметь модификаторы public static, так как перегружаемый оператор будет использоваться для всех объектов данного класса. Далее идет название возвращаемого типа. Возвращаемый тип представляет тот тип, объекты которого мы хотим получить. К примеру, в результате сложения двух объектов Counter мы ожидаем получить новый объект Counter. А в результате сравнения двух мы хотим получить объект типа bool, который указывает истинно

ли условное выражение или ложно. Но в зависимости от задачи возвращаемые типы могут быть любыми.

Затем вместо названия метода идет ключевое слово operator и собственно сам оператор. И далее в скобках перечисляются параметры. Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.

Например, перегрузим ряд операторов для класса Counter:

```
class Counter
{
    public int Value { get; set; }

    public static Counter operator +(Counter c1, Counter c2)
    {
        return new Counter { Value = c1.Value + c2.Value };
    }
    public static bool operator >(Counter c1, Counter c2)
    {
        return c1.Value > c2.Value;
    }
    public static bool operator <(Counter c1, Counter c2)
    {
        return c1.Value < c2.Value;
    }
}
```

Поскольку все перегруженные операторы - бинарные - то есть проводятся над двумя объектами, то для каждой перегрузки предусмотрено по два параметра.

Так как в случае с операцией сложения мы хотим сложить два объекта класса Counter, то оператор принимает два объекта этого класса. И так как мы хотим в результате сложения получить новый объект Counter, то данный класс также используется в качестве возвращаемого типа. Все действия этого оператора сводятся к созданию, нового объекта, свойство Value которого объединяет значения свойства Value обоих параметров:

```
public static Counter operator +(Counter c1, Counter c2)
{
    return new Counter { Value = c1.Value + c2.Value };
}
```

Также переопределены две операции сравнения. Если мы переопределяем одну из этих операций сравнения, то мы также должны переопределить вторую из этих операций. Сами операторы сравнения сравнивают значения свойств Value и в зависимости от результата сравнения возвращают либо true, либо false.

Теперь используем перегруженные операторы в программе:

```
static void Main(string[] args)
{
    Counter c1 = new Counter { Value = 23 };
    Counter c2 = new Counter { Value = 45 };
    bool result = c1 > c2;
    Console.WriteLine(result); // false

    Counter c3 = c1 + c2;
    Console.WriteLine(c3.Value);  // 23 + 45 = 68

    Console.ReadKey();
}
```

Стоит отметить, что так как по сути определение оператора представляет собой метод, то этот метод мы также можем перегрузить, то есть создать для него еще одну версию. Например, добавим в класс Counter еще один оператор:

```
public static int operator +(Counter c1, int val)
{
    return c1.Value + val;
}
```

Данный метод складывает значение свойства Value и некоторое число, возвращая их сумму. И также мы можем применить этот оператор:

```
Counter c1 = new Counter { Value = 23 };
int d = c1 + 27; // 50
```

```
Console.WriteLine(d);
```
Следует учитывать, что при перегрузке не должны изменяться те объекты, которые передаются в оператор через параметры. Например, мы можем определить для класса Counter оператор инкремента:
```
public static Counter operator ++(Counter c1)
{
    c1.Value += 10;
    return c1;
}
```
Поскольку оператор унарный, он принимает только один параметр - объект того класса, в котором данный оператор определен. Но это неправильное определение инкремента, так как оператор не должен менять значения своих параметров.
И более корректная перегрузка оператора инкремента будет выглядеть так:
```
public static Counter operator ++(Counter c1)
{
    return new Counter { Value = c1.Value + 10 };
}
```
То есть возвращается новый объект, который содержит в свойстве Value инкрементированное значение.
При этом нам не надо определять отдельно операторы для префиксного и для постфиксного инкремента (а также декремента), так как одна реализация будет работать в обоих случаях.
Например, используем операцию префиксного инкремента:
```
Counter counter = new Counter() { Value = 10 };
Console.WriteLine($"{counter.Value}");      // 10
Console.WriteLine($"{(++counter).Value}");  // 20
Console.WriteLine($"{counter.Value}");      // 20
```
Консольный вывод:
```
10
20
20
```
Теперь используем постфиксный инкремент:
```
Counter counter = new Counter() { Value = 10 };
Console.WriteLine($"{counter.Value}");      // 10
Console.WriteLine($"{(counter++).Value}");  // 10
Console.WriteLine($"{counter.Value}");      // 20
```
Консольный вывод:
```
10
10
20
```
Также стоит отметить, что мы можем переопределить операторы true и false. Например, определим их в классе Counter:
```
class Counter
{
    public int Value { get; set; }

    public static bool operator true(Counter c1)
    {
        return c1.Value != 0;
    }
    public static bool operator false(Counter c1)
    {
        return c1.Value == 0;
    }

    // остальное содержимое класса
}
```
Эти операторы перегружаются, когда мы хотим использовать объект типа в качестве условия. Например:
```
Counter counter = new Counter() { Value = 0 };
if (counter)
    Console.WriteLine(true);
else
    Console.WriteLine(false);
```

При перегрузке операторов надо учитывать, что не все операторы можно перегрузить. В частности, мы можем перегрузить следующие операторы:

- унарные операторы +, -, !, ~, ++, --
- бинарные операторы +, -, *, /, %
- операции сравнения ==, !=, <, >, <=, >=
- логические операторы &&, ||
- операторы присваивания +=, -=, *=, /=, %=

И есть ряд операторов, которые нельзя перегрузить, например, операцию равенства = или тернарный оператор ?:, а также ряд других.
Полный список перегружаемых операторов можно найти в документации msdn
При перегрузке операторов также следует помнить, что мы не можем изменить приоритет оператора или его ассоциативность, мы не можем создать новый оператор или изменить логику операторов в типах, который есть по умолчанию в .NET.

Значение null

Одно из отличий ссылочных типов от типов значений состоит в том, что переменные ссылочных типов могут принимать значение null. Например:

```
object o = null;
string s = null;
```

Если переменным ссылочного типа не присваивается значение, то им дается значение по умолчанию - значение null. Фактически оно говорит об отстутсвии значения как такового.
Но типы значений, например, int, decimal, double и т.д. не могут принимать значение null.

Оператор ??

Оператор ?? называется оператором null-объединения. Он применяется для установки значений по умолчанию для типов, которые допускают значение null. Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд. При этом левый операнд должен принимать null. Посмотрим на примере:

```
object x = null;
object y = x ?? 100;  // равно 100, так как x равен null

object z = 200;
object t = z ?? 44; // равно 200, так как z не равен null
```

Но мы не можем написать следующим образом:

```
int x = 44;
int y = x ?? 100;
```

Здесь переменная x представляет значимый тип int и не может принимать значение null, поэтому в качестве левого операнда в операции ?? она использоваться не может.

Оператор условного null

Иногда при работе с объектами, которые принимают значение null, мы можем столкнуться с ошибкой: мы пытаемся обратиться к объекту, а этот объект равен null. Например, пусть у нас есть следующая система классов:

```
class User
{
    public Phone Phone { get; set; }
}

class Phone
{
    public Company Company { get; set; }
}

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

Объект User содержит ссылку на объект Phone, а объект Phone содержит ссылку на объект Company, поэтому теоретически мы можем получить из объекта User название компании, например, так:

```
User user = new User();
Console.WriteLine(user.Phone.Company.Name);
```

В данном случае свойство Phone не определено, будет по умолчанию иметь значение null. Поэтому мы столкнемся с исключением NullReferenceException. Чтобы избежать этой ошибки мы могли бы использовать условную конструкцию для проверки на null:

```
User user = new User();

if(user!=null)
{
    if(user.Phone!=null)
    {
        if (user.Phone.Company != null)
        {
            string companyName = user.Phone.Company.Name;
            Console.WriteLine(companyName);
        }
    }
}
```

Получается многоэтажная конструкция, но на самом деле ее можно сократить:

```
if(user!=null && user.Phone!=null && user.Phone.Company!=null)
{
    string companyName = user.Phone.Company.Name;
    Console.WriteLine(companyName);
}
```

Если user не равно null, то проверяется следующее выражение user.Phone!=null и так далее. Конструкция намного проще, но все равно получается довольно большой. И чтобы ее упростить, в C# оператор условного null (Null-Conditional Operator):

```
string companyName = user?.Phone?.Company?.Name;
```

Выражение ?. и представляет оператор условного null. Здесь последовательно проверяется равен ли объект user и вложенные объекты значению null. Если же на каком-то этапе один из объектов окажется равным null, то companyName будет иметь значение по умолчанию, то есть null.

И в этом случае мы можем пойти дальше и применить операцию ?? для установки значения по умолчанию, если название компании не установлено:

```
User user = new User();
string companyName = user?.Phone?.Company?.Name ?? "не установлено";
Console.WriteLine(companyName);
```

Индексаторы

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Формальное определение индекатора:

```
возвращаемый_тип this [Тип параметр1, ...]
{
    get { ... }
    set { ... }
}
```

В отличие от свойств индексатор не имеет названия. Вместо его указывается ключевое слово this, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Посмотрим на примере. Допустим, у нас есть класс Person, который представляет человека, и класс People, который представляет группу людей. Используем индексаторы для определения класса People:

```
class Person
{
    public string Name { get; set; }
}
class People
{
    Person[] data;
    public People()
    {
```

```
        data = new Person[5];
    }
    // индексатор
    public Person this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }
}
```

Конструкция public Person this[int index] и представляет индексатор. Здесь определяем, во-первых, тип возвращаемого или присваиваемого объекта, то есть тип Person. Во-вторых, определяем через параметр int index способ доступа к элементам.

По сути все объекты Person хранятся в классе в массиве data. Для получения их по индексу в индекаторе определен блок get:

```
get
{
    return data[index];
}
```

Поскольку индексатор имеет тип Person, то в блоке get нам надо возвратить объект этого типа с помощью оператора return. Здесь мы можем определить разнообразную логику. В данном случае просто возвращаем объект из массива data.

В блоке set получаем через параметр value переданный объект Person и сохраняем его в массив по индексу.

```
set
{
    data[index] = value;
}
```

После этого мы можем работать с объектом People как с набором объектов Person:

```
class Program
    {
    static void Main(string[] args)
    {
        People people = new People();
        people[0] = new Person { Name = "Tom" };
        people[1] = new Person { Name = "Bob" };

        Person tom = people[0];
        Console.WriteLine(tom?.Name);

        Console.ReadKey();
    }
}
```

Индексатор, как полагается получает набор индексов в виде параметров. Однако индексы необязаткельно должны представлять тип int. Например, мы можем рассматривать объект как хранилище свойств и передавать имя атрибута объекта в виде строки:

```
class User
{
    string name;
    string email;
    string phone;

    public string this[string propname]
    {
        get
        {
            switch (propname)
            {
                case "name": return "Mr/Ms. " + name;
```

```
            case "email": return email;
            case "phone": return phone;
            default: return null;
        }
    }
    set
    {
        switch (propname)
        {
            case "name":
                name = value;
                break;
            case "email":
                email = value;
                break;
            case "phone":
                phone = value;
                break;
        }
    }
  }
}
class Program
{
  static void Main(string[] args)
  {
    User tom = new User();
    tom["name"] = "Tom";
    tom["email"] = "tomekvilmovskiy@gmail.ru";

    Console.WriteLine(tom["name"]); // Mr/Ms. Tom

    Console.ReadKey();
  }
}
```

Применение нескольких параметров

Также индексатор может принимать несколько параметров. Допустим, у нас есть класс, в котором хранилище определено в виде двухмерного массива или матрицы:

```
class Matrix
{
  private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };
  public int this[int i, int j]
  {
    get
    {
      return numbers[i,j];
    }
    set
    {
      numbers[i, j] = value;
    }
  }
}
```

Теперь для определения индексатора используются два индекса - i и j. И в программе мы уже должны обращаться к объекту, используя два индекса:

```
Matrix matrix = new Matrix();
Console.WriteLine(matrix[0, 0]);
matrix[0, 0] = 111;
Console.WriteLine(matrix[0, 0]);
```

Следует учитывать, что индексатор не может быть статическим и применяется только к экземпляру класса. Но при этом индексаторы могут быть виртуальными и абстрактными и могут переопределяться в производных классах.

Блоки get и set

Как и в свойствах, в индекаторах можно опускать блок get или set, если в них нет необходимости. Например, удалим блок set и сделаем индексатор доступным только для чтения:

```
class Matrix
{
    private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };
    public int this[int i, int j]
    {
        get
        {
            return numbers[i,j];
        }
    }
}
```

Также мы можем ограничивать доступ к блокам get и set, используя модификаторы доступа. Например, сделаем блок set приватным:

```
class Matrix
{
    private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };
    public int this[int i, int j]
    {
        get
        {
            return numbers[i,j];
        }
        private set
        {
            numbers[i, j] = value;
        }
    }
}
```

Перегрузка индексаторов

Подобно методам индексаторы можно перегружать. В этом случае также индексаторы должны отличаться по количеству, типу или порядку используемых параметров. Например:

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
class People
{
    Person[] data;
    public People()
    {
        data = new Person[5];
    }
    public Person this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }
    public Person this[string name]
    {
        get
        {
            Person person = null;
            foreach(var p in data)
            {
                if(p?.Name == name)
                {
```

```
                person = p;
                break;
            }
        }
        return person;
    }
  }
}
class Program
{
  static void Main(string[] args)
  {
    People people = new People();
    people[0] = new Person { Name = "Tom" };
    people[1] = new Person { Name = "Bob" };

    Console.WriteLine(people[0].Name);      // Tom
    Console.WriteLine(people["Bob"].Name);  // Bob

    Console.ReadKey();
  }
}
```

В данном случае класс People содержит две версии индексатора. Первая версия получает и устанавливает объект Person по индексу, а вторая - только получае объект Person по его имени.

Наследование

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс Person, который описывает отдельного человека:

```
class Person
{
  private string _name;

  public string Name
  {
    get { return _name; }
    set { _name = value; }
  }
  public void Display()
  {
    Console.WriteLine(Name);
  }
}
```

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

```
class Employee : Person
{

}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса.

Таким образом, наследование реализует отношение is-a (является), объект класса Employee также является объектом класса Person:

```
static void Main(string[] args)
{
  Person p = new Person { Name = "Tom"};
  p.Display();
  p = new Employee { Name = "Sam" };
  p.Display();
```

```
    Console.Read();
}
```
И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: `Person p = new Employee()`.

По умолчанию все классы наследуются от базового класса Object, даже если мы явным образом не устанавливаем наследование. Поэтому выше определенные классы Person и Employee кроме своих собственных методов, также будут иметь и методы класса Object: ToString(), Equals(), GetHashCode() и GetType().

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

- Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
- При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа internal, то производный класс может иметь тип доступа internal или private, но не public.
- Если класс объявлен с модификатором sealed, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{
}
```

- Нельзя унаследовать класс от статического класса.

Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(_name);
    }
}
```

Этот код не сработает и выдаст ошибку, так как переменная _name объявлена с модификатором private и поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство Name, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(Name);
    }
}
```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами private protected (если базовый и производный класс находятся в одной сборке), public, internal, protected и protected internal.

Ключевое слово base

Теперь добавим в наши классы конструкторы:

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

    public Person(string name)
    {
        Name = name;
    }

    public void Display()
    {
        Console.WriteLine(Name);
    }
```

```
}

class Employee : Person
{
   public string Company { get; set; }

   public Employee(string name, string company)
      : base(name)
   {
      Company = company;
   }
}
```
Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова base мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(name).

```
static void Main(string[] args)
{
   Person p = new Person("Bill");
   p.Display();
   Employee emp = new Employee ("Tom", "Microsoft");
   emp.Display();
   Console.Read();
}
```

Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base. Например, из класса Employee уберем определение конструктора:

```
class Employee : Person
{
   public string Company { get; set; }
}
```
В данном случае мы получим ошибку, так как класс Employee не соответствует классу Person, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

```
public Employee(string name, string company)
{
   Name = name;
   Company = company;
}
```
То есть в классе Employee через ключевое слово base надо явным образом вызвать конструктор класса Person:

```
public Employee(string name, string company)
      : base(name)
{
   Company = company;
}
```
Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

```
class Person
{
   // остальной код класса
   // конструктор по умолчанию
   public Person()
   {
```

```
        FirstName = "Tom";
        Console.WriteLine("Вызов конструктора без параметров");
    }
}
```
Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор
```
public Employee(string company)
{
    Company = company;
}
```
Фактически был бы эквивалентен следующему конструктору:
```
public Employee(string company)
    :base()
{
    Company = company;
}
```
Порядок вызова конструкторов
При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных. Например, возьмем следующие классы:
```
class Person
{
    string name;
    int age;

    public Person(string name)
    {
        this.name = name;
        Console.WriteLine("Person(string name)");
    }
    public Person(string name, int age) : this(name)
    {
        this.age = age;
        Console.WriteLine("Person(string name, int age)");
    }
}
class Employee : Person
{
    string company;

    public Employee(string name, int age, string company) : base(name, age)
    {
        this.company = company;
        Console.WriteLine("Employee(string name, int age, string company)");
    }
}
```
При создании объекта Employee:
```
Employee tom = new Employee("Tom", 22, "Microsoft");
```
Мы получим следующий консольный вывод:
```
Person(string name)
Person(string name, int age)
Employee(string name, int age, string company)
```
В итоге мы получаем следующую цепь выполнений.

1. Вначале вызывается конструктор Employee(string name, int age, string company). Он делегирует выполнение конструктору Person(string name, int age)
2. Вызывается конструктор Person(string name, int age), который сам пока не выполняется и передает выполнение конструктору Person(string name)
3. Вызывается конструктор Person(string name), который передает выполнение конструктору класса System.Object, так как это базовый по умолчанию класс для Person.
4. Выполняется конструктор System.Object.Object(), затем выполнение возвращается конструктору Person(string name)
5. Выполняется тело конструктора Person(string name), затем выполнение возвращается конструктору Person(string name, int age)

6. Выполняется тело конструктора Person(string name, int age), затем выполнение возвращается конструктору Employee(string name, int age, string company)
7. Выполняется тело конструктора Employee(string name, int age, string company). В итоге создается объект Employee

Преобразование типов

В предыдущей главе мы говорили о преобразованиях объектов простых типов. Сейчас затронем тему преобразования объектов классов. Допустим, у нас есть следующая иерархия классов:

```
class Person
{
    public string Name { get; set; }
    public Person(string name)
    {
        Name = name;
    }
    public void Display()
    {
        Console.WriteLine($"Person {Name}");
    }
}

class Employee : Person
{
    public string Company { get; set; }
    public Employee(string name, string company) : base(name)
    {
        Company = company;
    }
}

class Client : Person
{
    public string Bank { get; set; }
    public Client(string name, string bank) : base(name)
    {
        Bank = bank;
    }
}
```

В этой иерархии классов мы можем проследить следующую цепь наследования: Object (все классы неявно наследуются от типа Object) -> Person -> Employee|Client.

Причем в этой иерархии классов базовые типы находятся вверху, а производные типы - внизу.

Восходящие преобразования. Upcasting

Объекты производного типа (который находится внизу иерархии) в то же время представляют и базовый тип. Например, объект Employee в то же время является и объектом класса Person. Что в принципе естественно, так как каждый сотрудник (Employee) является человеком (Person). И мы можем написать, например, следующим образом:

```
static void Main(string[] args)
{
    Employee employee = new Employee("Tom", "Microsoft");
    Person person = employee;   // преобразование от Employee к Person

    Console.WriteLine(person.Name);
    Console.ReadKey();
}
```

В данном случае переменной person, которая представляет тип Person, присваивается ссылка на объект Employee. Но чтобы сохранить ссылку на объект одного класса в переменную другого класса, необходимо выполнить преобразование типов - в данном случае от типа Employee к типу Person. И так как Employee наследуется от класса Person, то автоматически

выполняется неявное восходящее преобразование - преобразование к типу, которые находятся вверху иерархии классов, то есть к базовому классу.
В итоге переменные employee и person будут указывать на один и тот же объект в памяти, но переменной person будет доступна только та часть, которая представляет функционал типа Person.

Подобным образом поизводятся и другие восходящие преобразования:

```
Person person2 = new Client("Bob", "ContosoBank");   // преобразование от Client к Person
```

Здесь переменная person2, которая представляет тип Person, хранит ссылку на объект Client, поэтому также выполняется восходящее неявное преобразование от производного класса Client к базовому типу Person.
Восходящее неявное преобразование будет происходить и в следующем случае:

```
object person1 = new Employee("Tom", "Microsoft");  // от Employee к object
object person2 = new Client("Bob", "ContosoBank");  // от Client к object
object person3 = new Person("Sam");                 // от Person к object
```

Так как тип object - базовый для всех остальных типов, то преобразование к нему будет производиться автоматически.

Нисходящие преобразования. Downcasting

Но кроме восходящих преобразований от производного к базовому типу есть нисходящие преобразования или downcasting - от базового типа к производному. Например, в следующем коде переменная person хранит ссылку на объект Employee:

```
Employee employee = new Employee("Tom", "Microsoft");
Person person = employee;   // преобразование от Employee к Person
```

И может возникнуть вопрос, можно ли обратиться к функционалу типа Employee через переменную типа Person. Но автоматически такие преобразования не проходят, ведь не каждый человек (объект Person) является сотрудником предприятия (объектом Employee). И для нисходящего преобразования необходимо применить явное преобразования, указав в скобках тип, к которому нужно выполнить преобразование:

```
Employee employee = new Employee("Tom", "Microsoft");
Person person = employee;   // преобразование от Employee к Person

//Employee employee2 = person;    // так нельзя, нужно явное преобразование
Employee employee2 = (Employee)person;  // преобразование от Person к Employee
```

Рассмотрим некоторые примеры преобразований:

```
// Объект Employee также представляет тип object
object obj = new Employee("Bill", "Microsoft");

// чтобы обратиться к возможностям типа Employee, приводим объект к типу Employee
Employee emp = (Employee) obj;

// объект Client также представляет тип Person
Person person = new Client("Sam", "ContosoBank");
// преобразование от типа Person к Client
Client client = (Client)person;
```

В первом случае переменной obj присвоена ссылка на объект Employee, поэтому мы можем преобразовать объект obj к любому типу который располагается в иерархии классов между типом object и Employee.
Если нам надо обратиться к каким-то отдельным свойствам или методам объекта, то нам необязательно присваивать преобразованный объект переменной :

```
// Объект Employee также представляет тип object
object obj = new Employee("Bill", "Microsoft");

// преобразование к типу Person для вызова метода Display
((Person)obj).Display();
// либо так
// ((Employee)obj).Display();

// преобразование к типу Employee, чтобы получить свойство Company
string comp = ((Employee)obj).Company;
```

В то же время необходимо соблюдать осторожность при подобных преобразованиях. Например, что будет в следующем случае:

```
// Объект Employee также представляет тип object
object obj = new Employee("Bill", "Microsoft");

// преобразование к типу Client, чтобы получить свойство Bank
string bank = ((Client)obj).Bank;
```

В данном случае мы получим ошибку, так как переменная obj хранит ссылку на объект Employee. Данный объект является также объектом типов object и Person, поэтому мы можем преобразовать его к этим типам. Но к типу Client мы преобразовать не можем.

Другой пример:

```
Employee emp = new Person("Tom");   // ! Ошибка

Person person = new Person("Bob");
Employee emp2 = (Employee) person;  // ! Ошибка
```

В данном случае мы пытаемся преобразовать объект типа Person к типу Employee, а объект Person не является объектом Employee.

Существует ряд способов, чтобы избежать подобных ошибок преобразования.

Способы преобразований

Во-первых, можно использовать ключевое слово as. С помощью него программа пытается преобразовать выражение к определенному типу, при этом не выбрасывает исключение. В случае неудачного преобразования выражение будет содержать значение null:

```
Person person = new Person("Tom");
Employee emp = person as Employee;
if (emp == null)
{
    Console.WriteLine("Преобразование прошло неудачно");
}
else
{
    Console.WriteLine(emp.Company);
}
```

Второй способ заключается в отлавливании исключения InvalidCastException, которое возникнет в результате преобразования:

```
Person person = new Person("Tom");
try
{
    Employee emp = (Employee)person;
    Console.WriteLine(emp.Company);
}
catch (InvalidCastException ex)
{
    Console.WriteLine(ex.Message);
}
```

Третий способ заключается в проверке допустимости преобразования с помощью ключевого слова is:

```
Person person = new Person("Tom");
if(person is Employee)
{
    Employee emp = (Employee)person;
    Console.WriteLine(emp.Company);
}
else
{
    Console.WriteLine("Преобразование не допустимо");
}
```

Выражение person is Employee проверяет, является ли переменная person объектом типа Employee. Но так как в данном случае явно не является, то такая проверка вернет значение false, и преобразование не сработает.

Перегрузка операций преобразования типов

В прошлой теме была рассмотрена тема перегрузки операторов. И с этой темой тесно связана тема перегрузки опрераторов преобразования типов.
Ранее мы рассматривали явные и неявные преобразования примитивных типов. Например:
```
int x = 50;
byte y = (byte)x; // явное преобразование от int к byte
int z = y;  // неявное преобразование от byte к int
```
И было бы не плохо иметь возможность определять логику преобразования одних типов в другие. И с помощью перегрузки операторов мы можем это делать. Для этого в классе определяется метод следующей формы:
```
public static implicit|explicit operator Тип_в_который_надо_преобразовать(исходный_тип param)
{
   // логика преобразования
}
```
После модификаторов public static идет ключевое слово explicit (если преобразование явное, то есть нужна операция приведения типов) или implicit (если преобразование неявное). Затем идет ключевое слово operator и далее возвращаемый тип, в который надо преобразовать объект. В скобках в качестве параметра передается объект, который надо преобразовать.
Например, пусть у нас есть следующий класс Counter, который представляет счетчик-секундомер и который хранит количество секунд в свойстве Seconds:
```
class Counter
{
   public int Seconds { get; set; }

   public static implicit operator Counter(int x)
   {
      return new Counter { Seconds = x };
   }
   public static explicit operator int(Counter counter)
   {
      return counter.Seconds;
   }
}
```
Первый оператор преобразует число - объект типа int к типу Counter. Его логика проста - создается новый объект Counter, у которого устанавливается свойство Seconds.
Второй оператор преобразует объект Counter к типу int, то есть получает из Counter число.
Примение операторов преобразования в программе:
```
static void Main(string[] args)
{
   Counter counter1 = new Counter { Seconds = 23 };

   int x = (int)counter1;
   Console.WriteLine(x);   // 23

   Counter counter2 = x;
   Console.WriteLine(counter2.Seconds);  // 23
}
```
Поскольку операция преобразования из Counter в int определена с ключевым словом explicit, то есть как явное преобразование, то в этом случае необходимо применить операцию приведения типов:
```
int x = (int)counter1;
```
В случае с операцией преобразования от int к Counter ничего подобного делать не надо, поскольку данная операция определена с ключевым словом implicit, то есть как неявная. Какие операции преобразования делать явными, а какие неявные, в данном случае не столь важно, это решает разработчик по своему усмотрению.
Следует учитывать, что оператор преобразования типов должен преобразовывать из типа или в тип, в котором этот оператор определен. То есть оператор преобразования, определенный в типе Counter, должен либо принимать в качестве параметра объект типа Counter, либо возвращать объект типа Counter.
Рассмотрим также более сложные преобразования, к примеру, из одного составного типа в другой составной тип. Допустим, у нас есть еще класс Timer:
```
class Timer
{
```

```
    public int Hours { get; set; }
    public int Minutes { get; set; }
    public int Seconds { get; set; }
}
class Counter
{
    public int Seconds { get; set; }

    public static implicit operator Counter(int x)
    {
        return new Counter { Seconds = x };
    }
    public static explicit operator int(Counter counter)
    {
        return counter.Seconds;
    }
    public static explicit operator Counter(Timer timer)
    {
        int h = timer.Hours * 3600;
        int m = timer.Minutes * 60;
        return new Counter { Seconds = h + m + timer.Seconds };
    }
    public static implicit operator Timer(Counter counter)
    {
        int h = counter.Seconds / 3600;
        int m = (counter.Seconds - h * 3600) / 60;
        int s = counter.Seconds - h * 3600 - m * 60;
        return new Timer { Hours = h, Minutes = m, Seconds = s };
    }
}
```

Класс Timer представляет условный таймер, который хранит часы, минуты и секунды. Класс Counter представляет условный счетчик-секундомер, который хранит количество секунд. Исходя из этого мы можем определить некоторую логику преобразования из одного типа к другому, то есть получение из секунд в объекте Counter часов, минут и секунд в объекте Timer. Например, 3675 секунд по сути это 1 час, 1 минута и 15 секунд

Применение операций преобразования:

```
static void Main(string[] args)
{
    Counter counter1 = new Counter { Seconds = 115 };

    Timer timer = counter1;
    Console.WriteLine($"{timer.Hours}:{timer.Minutes}:{timer.Seconds}"); // 0:1:55

    Counter counter2 = (Counter)timer;
    Console.WriteLine(counter2.Seconds);  //115

    Console.ReadKey();
}
```

Виртуальные методы и свойства

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором virtual. Такие методы и свойства называют виртуальными. А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором override. Переопределенный метод в класе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

Например, рассмотрим следующие классы:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
```

```
    {
        FirstName = firstName;
        LastName = lastName;
    }
    public virtual void Display()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
        : base(firstName, lastName)
    {
        Company = company;
    }
}
```

Здесь класс Person представляет человека. Класс Employee наследуется от Person и представляет сотрудника предприятия. Этот класс кроме унаследованных свойств имеет еще одно свойство - Company.

Чтобы сделать метод Display доступным для переопределения, этот метод определен с модификатором virtual. Поэтому мы можем переопределить этот метод, но можем и не переопределять. Допустим, нас устраивает реализация метода из базового класса. В этом случае объекты Employee будут использовать реализацию метода Display из класса Person:

```
static void Main(string[] args)
{
    Person p1 = new Person("Bill", "Gates");
    p1.Display(); // вызов метода Display из класса Person

    Employee p2 = new Employee("Tom", "Smith", "Microsoft");
    p2.Display(); // вызов метода Display из класса Person

    Console.ReadKey();
}
```

Консольный вывод:

```
Bill Gates
Tom Smith
```

Но также можем переопределить виртуальный метод. Для этого в классе-наследнике определяется метод с модификатором override, который имеет то же именем и набор параметров:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
        : base(firstName, lastName)
    {
        Company = company;
    }

    public override void Display()
    {
        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");
    }
}
```

Возьмем те же самые объекты:

```
static void Main(string[] args)
{
    Person p1 = new Person("Bill", "Gates");
    p1.Display(); // вызов метода Display из класса Person

    Employee p2 = new Employee("Tom", "Smith", "Microsoft");
    p2.Display(); // вызов метода Display из класса Employee
```

```
    Console.ReadKey();
}
```
Консольный вывод:
```
Bill Gates
Tom Smith работает в Microsoft
```
Виртуальные методы базового класса определяют интерфейс всей иерархии, то есть в любом производном классе, который не является прямым наследником от базового класса, можно переопределить виртуальные методы. Например, мы можем определить класс Manager, который будет производным от Employee, и в нем также переопределить метод Display.

При переопределении виртуальных методов следует учитывать ряд ограничений:

- Виртуальный и переопределенный методы должны иметь один и тот же модификатор доступа. То есть есть если виртуальный метод определен с помощью модификатора public, то и переопредленный метод также должен иметь модификатор public.
- Нельзя переопределить или объявить виртуальным статический метод.

Переопределение свойств

Также как и методы, можно переопределять свойства:
```
class Credit
{
    public virtual decimal Sum { get; set; }
}
class LongCredit : Credit
{
    private decimal sum;
    public override decimal Sum
    {
        get
        {
            return sum;
        }
        set
        {
            if(value > 1000)
            {
                sum = value;
            }
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        LongCredit credit = new LongCredit { Sum = 6000 };
        credit.Sum = 490;
        Console.WriteLine(credit.Sum);
        Console.ReadKey();
    }
}
```
Ключевое слово base

Кроме конструкторов, мы можем обратиться с помощью ключевого слова base к другим членам базового класса. В нашем случае вызов base.Display(); будет обращением к методу Display() в классе Person:
```
class Employee : Person
{
    public string Company { get; set; }

    public Employee(string lastName, string firstName, string company)
        :base(firstName, lastName)
    {
        Company = company;
    }

    public override void Display()
    {
```

```
        base.Display();
        Console.WriteLine($"работает в {Company}");
    }
}
```

Запрет переопределения методов

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором sealed:

```
class Employee : Person
{
    public string Company { get; set; }

    public Employee(string firstName, string lastName, string company)
            : base(firstName, lastName)
    {
        Company = company;
    }

    public override sealed void Display()
    {
        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");
    }
}
```

При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

И в этом случае мы не сможем переопределить метод Display в классе, унаследованном от Employee.

Сокрытие методов

В прошлой теме было рассмотрено определение и переопределение виртуальных методов. Другим способом изменить функциональность метода, унаследованного от базового класса, является сокрытие (shadowing / hiding).

Фактически сокрытие представляет определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для сокрытия членов класса применяется ключевое слово new. Например:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public void Display()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}

class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
            : base(firstName, lastName)
    {
        Company = company;
    }
    public new void Display()
    {
        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");
    }
}
```

Здесь определен класс Person, представляющий человека, и класс Employee, представляющий работника предприятия. Employee наследует от Person все свойства и методы. Но в классе Employee кроме унаследованных свойств есть также и собственное свойство Company, которое хранит название компании. И мы хотели бы в методе Display выводить информацию о компании вместе с именем и фамилией на консоль. Для этого определяется метод Display с ключевым словом new, который скрывает реализацию данного метода из базового класса.

В каких ситуациях можно использовать сокрытие? Например, в примере выше метод Display в базовом классе не является виртуальным, мы не можем его переопределить, но, допустим, нам не устраивает его реализация для производного класса, поэтому мы можем воспользоваться сокрытием, чтобы определить нужный нам функционал.

Используем эти классы в программе в методе Main:

```
class Program
{
    static void Main(string[] args)
    {
        Person bob = new Person("Bob", "Robertson");
        bob.Display();      // Bob Robertson

        Employee tom = new Employee("Tom", "Smith", "Microsoft");
        tom.Display();      // Tom Smith работает в Microsoft

        Console.ReadKey();
    }
}
```

Консольный вывод программы:

```
Bob Robertson
Tom Smith работает в Microsoft
```

Подобным обазом мы можем организовать сокрытие свойств:

```
class Person
{
    protected string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
class Employee : Person
{
    public new string Name
    {
        get { return "Employee " + base.Name; }
        set { name = value; }
    }
}
```

При этом если мы хотим обратиться именно к реализации свойства или метода в базовом классе, то опять же мы можем использовать ключевое слово base и через него обращаться к функциональности базового класса.

Более того мы даже можем применять сокрытие к переменным и константам, также используя ключевое слово new:

```
class ExampleBase
{
    public readonly int x = 10;
    public const int G = 5;
}
class ExampleDerived : ExampleBase
{
    public new readonly int x = 20;
    public new const int G = 15;
}
```

Различие переопределения и сокрытия методов

Ранее было рассмотрена два способа изменения функциональности методов, унаследованных от базового класса - сокрытие и переопределение. В чем разница между двумя этими способами?

Переопределение

Возьмем пример с переопределением методов:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
    public virtual void Display()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
        : base(firstName, lastName)
    {
        Company = company;
    }

    public override void Display()
    {
        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");
    }
}
```

Также создадим объект Employee и передадим его переменной типа Person:

```
Person tom = new Employee("Tom", "Smith", "Microsoft");
tom.Display();      // Tom Smith работает в Microsoft
```

Теперь мы получаем иной результат, нежели при сокрытии. А при вызове tom.Display() выполняется реализация метода Display из класса Employee.

Для работы с виртуальными методами компилятор формирует таблицу виртуальных методов (Virtual Method Table или VMT). В нее записывается адреса виртуальных методов. Для каждого класса создается своя таблица.

Когда создается объект класса, то компилятор передает в конструктор объекта специальный код, который связывает объект и таблицу VMT.

А при вызове виртуального метода из объекта берется адрес его таблицы VMT. Затем из VMT извлекается адрес метода и ему передается управление. То есть процесс выбора реализации метода производится во время выполнения программы. Собственно так и выполняется виртуальный метод. Следует учитывать, что так как среде выполнения вначале необходимо получить из таблицы VMT адрес нужного метода, то это немного замедляет выполнение программы.

Сокрытие

Теперь возьмем те же классы Person и Employee, но вместо переопределения используем сокрытие:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public void Display()
```

```
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}

class Employee : Person
{
    public string Company { get; set; }
    public Employee(string firstName, string lastName, string company)
            : base(firstName, lastName)
    {
        Company = company;
    }
    public new void Display()
    {
        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");
    }
}
```

И посмотрим, что будет в следующем случае:

```
Person tom = new Employee("Tom", "Smith", "Microsoft");
tom.Display();      // Tom Smith
```

Переменная tom представляет тип Person, но хранит ссылку на объект Employee. Однако при вызове метода Display будет выполняться та версия метода, которая определена именно в классе Person, а не в классе Employee. Почему? Класс Employee никак не переопределяет метод Display, унаследованный от базового класса, а фактически определяет новый метод. Поэтому при вызове tom.Display() вызывается метод Display из класса Person.

Абстрактные классы и члены классов

Кроме обычных классов в C# есть абстрактные классы. Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Единственное, что при определении абстрактных классов используется ключевое слово abstract:

```
abstract class Human
{
    public int Length { get; set; }
    public double Weight { get; set; }
}
```

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

```
Human h = new Human();
```

Зачем нужны абстрактные классы? Допустим, в нашей программе для банковского сектора мы можем определить две основных сущности: клиента банка и сотрудника банка. Каждая из этих сущностей будет отличаться, например, для сотрудника надо определить его должность, а для клиента - сумму на счете. Соответственно клиент и сотрудник будут составлять отдельные классы Client и Employee. В то же время обе этих сущности могут иметь что-то общее, например, имя и фамилию, какую-то другую общую функциональность. И эту общую функциональность лучше вынести в какой-то отдельный класс, например, Person, который описывает человека. То есть классы Employee (сотрудник) и Client (клиент банка) будут производными от класса Person. И так как все объекты в нашей системе будут представлять либо сотрудника банка, либо клиента, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным:

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

    public Person(string name)
    {
        Name = name;
    }

    public void Display()
    {
        Console.WriteLine(Name);
    }
```

```
}

class Client : Person
{
    public int Sum { get; set; }    // сумма на счету

    public Client(string name, int sum)
        : base(name)
    {
        Sum = sum;
    }
}

class Employee : Person
{
    public string Position { get; set; } // должность

    public Employee(string name, string position)
        : base(name)
    {
            Position = position;
    }
}
```

Затем мы сможем использовать эти классы:

```
Client client = new Client("Tom", 500);
Employee employee = new Employee ("Bob", "Apple");
client.Display();
employee.Display();
```

Или даже так:

```
Person client = new Client("Tom", 500);
Person employee = new Employee ("Bob", "Операционист");
```

Но мы НЕ можем создать объект Person, используя конструктор класса Person:

```
Person person = new Person ("Bill");
```

Однако несмотря на то, что напрямую мы не можем вызвать конструктор класса Person для создания объекта, тем не менее конструктор в абстрактных классах то же может играть важную роль, в частности, инициализировать некоторые общие для производных классов переменные и свойства, как в случае со свойством Name. И хотя в примере выше конструктор класса Person не вызывается, тем не менее производные классы Client и Employee могут обращаться к нему.

Абстрактные члены классов

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова abstract и не имеют никакого функционала. В частности, абстрактными могут быть:

- Методы
- Свойства
- Индексаторы
- События

Абстрактные члены классов не должны иметь модификатор private. При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод или свойство также объявляются с модификатором override (как и при обычном переопределении виртуальных методов и свойств). Также следует учесть, что если класс имеет хотя бы один абстрактный метод (или абстрактные свойство, индексатор, событие), то этот класс должен быть определен как абстрактный.

Абстрактные члены также, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами мы говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

Абстрактные методы

Например, сделаем в примере выше метод Display абстрактным:

```
abstract class Person
{
```

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

    public Person(string name)
    {
        Name = name;
    }

    public abstract void Display();
}

class Client : Person
{
    public int Sum { get; set; }    // сумма на счету

    public Client(string name, int sum)
        : base(name)
    {
        Sum = sum;
    }
    public override void Display()
    {
        Console.WriteLine($"{Name} имеет счет на сумму {Sum}");
    }
}

class Employee : Person
{
    public string Position { get; set; } // должность

    public Employee(string name, string position)
        : base(name)
    {
        Position = position;
    }

    public override void Display()
    {
        Console.WriteLine($"{Position} {Name}");
    }
}
```

Абстрактные свойства

Следует отметить использование абстрактных свойств. Их определение похоже на определение автосвойств. Например:

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

class Client : Person
{
    private string name;

    public override string Name
    {
        get { return "Mr/Ms. " + name; }
        set { name = value; }
    }
}

class Employee : Person
{
    public override string Name { get; set; }
}
```

В классе Person определено абстрактное свойство Name. Оно похоже на автосвойство, но это не автосвойство. Так как данное свойство не должно иметь реализацию, то оно имеет только пустые блоки get и set. В производных классах мы можем переопределить это свойство, сделав

его полноценным свойством (как в классе Client), либо же сделав его автоматическим (как в классе Employee).

Отказ от реализации абстрактных членов

Производный класс обязан реализовать все абстрактные члены базового класса. Однако мы можем отказаться от реализации, но в этом случае производный класс также должен быть определен как абстрактный:

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

abstract class Manager : Person
{
}
```

Пример абстрактного класса

Хрестоматийным примером является система геометрических фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

```
// абстрактный класс фигуры
abstract class Figure
{
    // абстрактный метод для получения периметра
    public abstract float Perimeter();
    // абстрактный метод для получения площади
    public abstract float Area();
}
// производный класс прямоугольника
class Rectangle : Figure
{
    public float Width { get; set; }
    public float Height { get; set; }

    public Rectangle(float width, float height)
    {
        this.Width = width;
        this.Height = height;
    }
    // переопределение получения периметра
    public override float Perimeter()
    {
        return Width * 2 + Height * 2;
    }
    // переопрелеление получения площади
    public override float Area()
    {
        return Width * Height;
    }
}
```

Класс System.Object и его методы

Все остальные классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как System.Int32, являются неявно производными от класса Object. Даже если мы не указываем класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object. Рассмотрим эти методы.

ToString

Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

```
int i = 5;
Console.WriteLine(i.ToString()); // выведет число 5

double d = 3.5;
```

```
Console.WriteLine(d.ToString()); // выведет число 3,5
```
Для классов же этот метод выводит полное название класса с указанием пространства имен, в котором определен этот класс. И мы можем переопределить данный метод. Посмотрим на примере:
```
using System;

namespace FirstApp
{
    class Program
    {
        private static void Main(string[] args)
        {
            Person person = new Person { Name = "Tom" };
            Console.WriteLine(person.ToString()); // выведет название класса Person

            Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };
            Console.WriteLine(clock.ToString()); // выведет 15:34:53

            Console.Read();
        }
    }
    class Clock
    {
        public int Hours { get; set; }
        public int Minutes { get; set; }
        public int Seconds { get; set; }
        public override string ToString()
        {
            return $"{Hours}:{Minutes}:{Seconds}";
        }
    }
    class Person
    {
        public string Name { get; set; }
    }
}
```
Для переопределения метода ToString в классе Clock, который представляет часы, используется ключевое слово override (как и при обычном переопределении вирутальных или абстрактных методов). В данном случае метод ToString() значение свойств Hours, Minutes, Seconds, объединенные в одну строку.

Класс Person не переопределяет метод ToString, поэтому для этого класса срабатывает стандартная реализация этого метода, которая выводит просто название класса.

Кстати в данном случае мы могли задействовать обе реализации:
```
class Person
{
    public string Name { get; set; }
    public override string ToString()
    {
        if (String.IsNullOrEmpty(Name))
            return base.ToString();
        return Name;
    }
}
```
То есть если имя - свойство Name не имеет значения, оно представляет пустую строку, то возвращается базовая реализация - название класса. Если же имя установлено, то возвращается значение свойства Name. Для проверки строки на пустоту применяется метод String.IsNullOrEmpty().

Стоит отметить, что различные технологии на платформе .NET активно используют метод ToString для разных целей. В частости, тот же метод Console.WriteLine() по умолчанию выводит именно строковое представление объекта. Поэтому, если нам надо вывести строковое представление объекта на консоль, то при передаче объекта в метод Console.WriteLine необязательно использовать метод ToString() - он вызывается неявно:
```
private static void Main(string[] args)
{
```

```
    Person person = new Person { Name = "Tom" };
    Console.WriteLine(person);

    Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };
    Console.WriteLine(clock); // выведет 15:34:53

    Console.Read();
}
```

Метод GetHashCode

Метод GetHashCode позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализаци базового типа:

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

    public override int GetHashCode()
    {
        return Name.GetHashCode();
    }
}
```

В данном случае метод GetHashCode возвращает то хеш-код для значения свойства Name. То есть два объекта Person, которые имеют одно и то же имя, будут возвращать один и тот же хеш-код. Однако в реальности алгоритм может быть самым различным.

Получение типа объекта и метод GetType

Метод GetType позволяет получить тип данного объекта:

```
Person person = new Person { Name = "Tom" };
Console.WriteLine(person.GetType());    // Person
```

Этот метод возвращает объект Type, то есть тип объекта.

С помощью ключевого слова typeof мы получаем тип класса и сравниваем его с типом объекта. И если этот объект представляет тип Client, то выполняем определенные действия.

```
object person = new Person { Name = "Tom" };
if (person.GetType() == typeof(Person))
    Console.WriteLine("Это реально класс Person");
```

Причем поскольку класс Object является базовым типом для всех классов, то мы можем переменной типа object присвоить объект любого типа. Однако для этого переменной метод GetType все равно вернет тот тип, на объект которого ссылается переменная. То есть в данном случае объект типа Person.

В отличие от методов ToString, Equals, GetHashCode метод GetType не переопределяется.

Метод Equals

Метод Equals позволяет сравнить два объекта на равенство:

```
class Person
{
    public string Name { get; set; }
    public override bool Equals(object obj)
    {
        if (obj.GetType() != this.GetType()) return false;

        Person person = (Person)obj;
        return (this.Name == person.Name);
    }
}
```

Метод Equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса. Затем сравниваем по именам. Если имена равны, возвращаем true, что будет говорить, что объекты равны. Однако при необходимости реализацию метода можно сделать более сложной, например, сравнивать по нескольким свойствам при их наличии.

Применение метода:

```
Person person1 = new Person { Name = "Tom" };
Person person2 = new Person { Name = "Bob" };
```

```
Person person3 = new Person { Name = "Tom" };
bool p1Ep2 = person1.Equals(person2);   // false
bool p1Ep3 = person1.Equals(person3);   // true
```
И если следует сравнивать два сложных объекта, как в данном случае, то лучше использовать метод Equals, а не стандартную операцию ==.

Обобщения

С выходом версии 2.0 фреймворк .NET стал поддерживать обобщенные типы (generics), а также создание обобщенных методов. Чтобы разобраться в особенности данного явления, сначала посмотрим на проблему, которая могла возникнуть до появления обобщенных типов. Посмотрим на примере. Допустим, мы определяем класс для представления банковского счета. К примеру, он мог бы выглядеть следующим образом:

```
class Account
{
    public int Id { get; set; }
    public int Sum { get; set; }
}
```
Класс Account определяет два свойства: Id - уникальный идентификатор и Sum - сумму на счете.

Здесь идентификатор задан как числовое значение, то есть банковские счета будут иметь значения 1, 2, 3, 4 и так далее. Однако также нередко для идентификатора используются и строковые значения. И у числовых, и у строковых значений есть свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме.

И на первый взгляд, чтобы выйти из подобной ситуации, мы можем определить свойство Id как свойство типа object. Так как тип object является универсальным типом, от которого наследуется все типы, соответственно в свойствах подобного типа мы можем сохранить и строки, и числа:

```
class Account
{
    public object Id { get; set; }
    public int Sum { get; set; }
}
```
Затем этот класс можно было использовать для создания банковских счетов в программе:

```
Account account1 = new Account { Sum = 5000 };
Account account2 = new Account { Sum = 4000 };
account1.Id = 2;
account2.Id = "4356";
int id1 = (int)account1.Id;
string id2 = (string)account2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```
Все вроде замечательно работает, но такое решение является не очень оптимальным. Дело в том, что в данном случае мы сталкиваемся с такими явлениями как упаковка (boxing) и распаковка (unboxing).

Так, при присвоении свойству Id значения типа int, происходит упаковка этого значения в тип Object:

```
account1.Id = 2;        // упаковка в значения int в тип Object
```
Чтобы обратно получить данные в переменную типов int, необходимо выполнить распаковку:

```
int id1 = (int)account1.Id;     // Распаковка в тип int
```
Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:

```
Account account2 = new Account { Sum = 4000 };
```

```
account2.Id = "4356";
int id2 = (int)account2.Id;     // Исключение InvalidCastException
```
Мы можем не знать, какой именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

Эти проблемы были призваны устранить обобщенные типы. Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс Account как обощенный:
```
class Account<T>
{
    public T Id { get; set; }
    public int Sum { get; set; }
}
```
Угловые скобки в описании class Account<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр T в угловых скобках в еще называется универсальным параметром, так как вместо него можно подставить любой тип.

Например, вместо параметра T можно использовать объект int, то есть число, представляющее номер счета. Это также может быть объект string, либо или любой другой класс или структура:
```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<string> account2 = new Account<string> { Sum = 4000 };
account1.Id = 2;        // упаковка не нужна
account2.Id = "4356";
int id1 = account1.Id;  // распаковка не нужна
string id2 = account2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```
Поскольку класс Account является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра T. В данном случае объекты Account типизируется типами int и string:
```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<string> account2 = new Account<string> { Sum = 4000 };
```
Поэтому у первого объекта account1 свойство Id будет иметь тип int, а у объекта account2 - тип string.

При попытке присвоить значение свойства Id переменной другого типа мы получим ошибку компиляции:
```
Account<string> account2 = new Account<string> { Sum = 4000 };
account2.Id = "4356";
int id1 = account2.Id;  // ошибка компиляции
```
Тем самым мы избежим проблем с типобезопасностью. Таким образом, используя обобщенный вариант класса, мы снижаем время на выполнение и количество потенциальных ошибок.

Значения по умолчанию

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и null. Но напрямую мы его присвоить не можем:
```
T id = null;
```
В этом случае нам надо использовать оператор default(T). Он присваивает ссылочным типам в качестве значения null, а типам значений - значение 0:
```
class Account<T>
{
    T id = default(T);
}
```
Статические поля обобщенных классов

При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов. Например, в классе Account определено следующее статическое поле:
```
class Account<T>
{
    public static T session;
```

```
    public T Id { get; set; }
    public int Sum { get; set; }
}
```
Теперь типизируем класс двумя типами int и string:
```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<int>.session = 5436;

Account<string> account2 = new Account<string> { Sum = 4000 };
Account<string>.session = "45245";

Console.WriteLine(Account<int>.session);      // 5436
Console.WriteLine(Account<string>.session);   // 45245
```
В итоге для Account<string> и для Account<int> будет создана своя переменная session.

Использование нескольких универсальных параметров

Обобщения могут использовать несколько универсальных параметров одновременно, которые могут представлять различные типы:
```
class Transaction<U, V>
{
    public U FromAccount { get; set; }  // с какого счета перевод
    public U ToAccount { get; set; }    // на какой счет перевод
    public V Code { get; set; }         // код операции
    public int Sum { get; set; }        // сумма перевода
}
```
Здесь класс Transaction использует два универсальных параметра. Применим данный класс:
```
Account<int> acc1 = new Account<int> { Id = 1857, Sum = 4500 };
Account<int> acc2 = new Account<int> { Id = 3453, Sum = 5000 };

Transaction<Account<int>, string> transaction1 = new Transaction<Account<int>, string>
{
    FromAccount = acc1,
    ToAccount = acc2,
    Code = "45478758",
    Sum = 900
};
```
Здесь объект Transaction типизируется типами Account<int> и string. То есть в качестве универсального параметра U используется класс Account<int>, а для параметра V - тип string. При этом, как можно заметить, класс, которым типизируется Transaction, сам является обобщенным.

Обобщенные методы

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:
```
class Program
{
    private static void Main(string[] args)
    {
        int x = 7;
        int y = 25;
        Swap<int>(ref x, ref y);
        Console.WriteLine($"x={x}    y={y}");   // x=25   y=7

        string s1 = "hello";
        string s2 = "bye";
        Swap<string>(ref s1, ref s2);
        Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye   s2=hello

        Console.Read();
    }
    public static void Swap<T> (ref T x, ref T y)
    {
        T temp = x;
        x = y;
        y = temp;
    }
}
```

Здесь определен обощенный метод Swap, который принимает параметры по ссылке и меняет их значения. При этом в данном случае не важно, какой тип представляют эти параметры.
В методе Main вызываем метод Swap, типизируем его определенным типом и передаем ему некоторые значения.

Ограничения обобщений

Ограничения универсальных типов

С помощью универсальных параметров мы можем типизировать обобщенные классы любым типом. Однако иногда возникает необходимость конкретизировать тип. Например, у нас есть следующий класс Account, который представляет банковский счет:

```
class Account
{
   public int Id { get; private set;} // номер счета
   public int Sum { get; set; }
   public Account(int _id)
   {
      Id = _id;
   }
}
```

Для перевода средств с одного счета на другой мы можем определить класс Transaction, который для выполнения всех операций будет использовать объекты класса Account.
Но у класса Account может быть много наследников: DepositAccount (депозитный счет), DemandAccount (счет до востребования) и т.д. И мы не можем знать, какие именно типы счетов будут использоваться в классе Transaction. Возможно, транзакции будут проводиться только между счетами до востребования. И в этом случае в качестве универсального параметра можно установить тип Account:

```
class Transaction<T> where T: Account
{
   public T FromAccount { get; set; }  // с какого счета перевод
   public T ToAccount { get; set; }    // на какой счет перевод
   public int Sum { get; set; }        // сумма перевода

   public void Execute()
   {
      if (FromAccount.Sum > Sum)
      {
         FromAccount.Sum -= Sum;
         ToAccount.Sum += Sum;
         Console.WriteLine($"Счет {FromAccount.Id}: {FromAccount.Sum}$ \nСчет {ToAccount.Id}: {ToAccount.Sum}$");
      }
      else
      {
         Console.WriteLine($"Недостаточно денег на счете {FromAccount.Id}");
      }
   }
}
```

С помощью выражения where T : Account мы указываем, что используемый тип T обязательно должен быть классом Account или его наследником. Благодаря подобному ограничению мы можем использовать внутри класса Transaction все объекты типа T именно как объекты Account и соответственно обращаться к их свойствам и методам.
Теперь применим класс Transaction в программе:

```
class Program
{
   static void Main(string[] args)
   {
      Account acc1 = new Account(1857) { Sum = 4500 };
      Account acc2 = new Account(3453) { Sum = 5000 };
      Transaction<Account> transaction1 = new Transaction<Account>
      {
         FromAccount = acc1,
         ToAccount = acc2,
         Sum = 6900
```

```
        };
        transaction1.Execute();

        Console.ReadLine();
    }
}
```

Следует учитывать, что только один класс может использоваться в качестве ограничения. В качестве ограничения также может выступать и обобщенный класс:

```
class Program
{
    private static void Main(string[] args)
    {
        Account<int> acc1 = new Account<int>(1857) { Sum = 4500 };
        Account<int> acc2 = new Account<int>(3453) { Sum = 5000 };

        Transaction<Account<int>> transaction1 = new Transaction<Account<int>>
        {
            FromAccount = acc1,
            ToAccount = acc2,
            Sum = 6900
        };
        transaction1.Execute();

        Console.Read();
    }
}
class Account<T>
{
    public T Id { get; private set; } // номер счета
    public int Sum { get; set; }
    public Account(T _id)
    {
        Id = _id;
    }
}
class Transaction<T> where T: Account<int>
{
    public T FromAccount { get; set; }  // с какого счета перевод
    public T ToAccount { get; set; }    // на какой счет перевод
    public int Sum { get; set; }        // сумма перевода

    public void Execute()
    {
        if (FromAccount.Sum > Sum)
        {
            FromAccount.Sum -= Sum;
            ToAccount.Sum += Sum;
            Console.WriteLine($"Счет {FromAccount.Id}: {FromAccount.Sum}$ \nСчет {ToAccount.Id}:
{ToAccount.Sum}$");
        }
        else
        {
            Console.WriteLine($"Недостаточно денег на счете {FromAccount.Id}");
        }
    }
}
```

В данном случае класс Transaction типизирован классом Account<int>. Класс Account же может быть типизирован абсолютно любым типом. Однако класс Transaction может использовать только объекты класса Account<int> или его наследников. То есть следующий код ошибочен и работать не будет:

```
Account<string> acc1 = new Account<string>("34") { Sum = 4500 };
Account<string> acc2 = new Account<string>("45") { Sum = 5000 };

// так нельзя написать, так как Bank должен быть типизирован классом Account<int> или его
наследником
Transaction<Account<string>> transaction1 = new Transaction<Account<string>>
```

```
{
   FromAccount = acc1,
   ToAccount = acc2,
   Sum = 900
};
```
В качестве ограничений мы можем использовать следующие типы:

- Классы
- Интерфейсы
- class - универсальный параметр должен представлять класс
- struct - универсальный параметр должен представлять структуру
- new() - универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров

Стандартные ограничения

Есть ряд стандартных ограничений, которые мы можем использовать. В частности, можно указать ограничение, чтобы использовались только структуры или другие типы значений:

```
class Account<T> where T : struct
{}
```
При этом использовать в качестве ограничения конкретные структуры в отличие от классов нельзя.

Также можно задать в качестве ограничения ссылочные типы:

```
class Transaction<T> where T : class
{}
```
А также можно задать с помощью слова new в качестве ограничения класс или структуру, которые имеют общедоступный конструктор без параметров:

```
class Transaction<T> where T : new()
{}
```
Если для универсального параметра задано несколько ограничений, то они должны идти в определенном порядке:

1. Название класса, class, struct. Причем мы можем одновременно определить только одно из этих ограничений
2. Название интерфейса
3. new()

Например:

```
interface IAccount
{
   int CurrentSum { get; set;}
}
class Person
{
   public string Name { get; set; }
}

class Transaction<T> where T: Person, IAccount, new()
{

}
```

Использование нескольких универсальных параметров

Если класс использует несколько универсальных параметров, то последовательно можно задать ограничения к каждому из них:

```
class Transaction<U, V>
      where U : Account<int>
      where V : struct
{

}
```

Ограничения методов

Подобным образом можно использовать и ограничения методов:

```
private static void Main(string[] args)
{
   Account<int> acc1 = new Account<int>(1857) { Sum = 4500 };
   Account<int> acc2 = new Account<int>(3453) { Sum = 5000 };
```

```
    Transact<Account<int>>(acc1, acc2, 900);

    Console.Read();
}

public static void Transact<T>(T acc1, T acc2, int sum) where T : Account<int>
{
    if (acc1.Sum > sum)
    {
        acc1.Sum -= sum;
        acc2.Sum += sum;
    }
    Console.WriteLine($"acc1: {acc1.Sum}   acc2: {acc2.Sum}");
}
```

Метод Transact в качестве ограничения принимает тип Account<int>.

Наследование обобщенных типов

Один обобщенный класс может быть унаследован от другого обобщенного. При этом можно использовать различные варианты наследования.

Допустим, у нас есть следующий базовый класс Account:

```
class Account<T>
{
    public T Id { get; private set; }
    public Account(T _id)
    {
        Id = _id;
    }
}
```

Первый вариант заключается в создание класса-наследника, который типизирован тем же типом, что и базовый:

```
class UniversalAccount<T> : Account<T>
{
    public UniversalAccount(T id) : base(id)
    {

    }
}
```

Применение класса:

```
Account<string> acc1 = new Account<string>("34");
Account<int> acc3 = new UniversalAccount<int>(45);
UniversalAccount<int> acc2 = new UniversalAccount<int>(33);
Console.WriteLine(acc1.Id);
Console.WriteLine(acc2.Id);
Console.WriteLine(acc3.Id);
```

Второй вариант представляет создание обычного необобщенного класса-наследника. В этом случае при наследовании у базового класса надо явным образом определить используемый тип:

```
class StringAccount : Account<string>
{
    public StringAccount(string id) : base(id)
    {
    }
}
```

Теперь в производном классе в качестве типа будет использоваться тип string. Применение класса:

```
StringAccount acc4 = new StringAccount("438767");
Account<string> acc5 = new StringAccount("43875");
// так нельзя написать
//Account<int> acc6 = new StringAccount("45545");
```

Третий вариант представляет типизацию производного класса параметром совсем другого типа, отличного от универсального параметра в базовом классе. В этом случае для базового класса также надо указать используемый тип:

```
class IntAccount<T> : Account<int>
```

```
{
    public T Code { get; set; }
    public IntAccount(int id) : base(id)
    {
    }
}
```
Здесь тип IntAccount типизирован еще одним типом, который может не совпадать с типом, который используется базовым классом. Применение класса:
```
IntAccount<string> acc7 = new IntAccount<string>(5) { Code = "r4556" };
Account<int> acc8 = new IntAccount<long>(7) { Code = 4587 };
Console.WriteLine(acc7.Id);
Console.WriteLine(acc8.Id);
```
И также в классах-наследниках можно сочетать использование универсального параметра из базового класса с применением своих параметров:
```
class MixedAccount<T, K> : Account<T>
    where K : struct
{
    public K Code { get; set; }
    public MixedAccount(T id) : base(id)
    {

    }
}
```
Здесь в дополнение к унаследованному от базового класса параметру T добавляется новый параметр K. Также если необходимо при этом задать ограничения, мы их можем указать после названия базового класса. Применение класса:
```
MixedAccount<string, int> acc9 = new MixedAccount<string, int>("456") { Code = 356 };
Account<string> acc10 = new MixedAccount<string, int>("9867") { Code = 35678 };
Console.WriteLine(acc9.Id);
Console.WriteLine(acc10.Id);
```
При этом стоит учитывать, что если на уровне базового класса для универсального параметра установлено ограничение, то подобное ограничение должно быть определено и в производных классах, которые также используют этот параметр:
```
class Account<T> where T : class
{
    public T Id { get; private set; }
    public Account(T _id)
    {
        Id = _id;
    }
}
class UniversalAccount<T> : Account<T>
    where T: class
{
    public UniversalAccount(T id) : base(id)
    {

    }
}
```
То есть если в базовом классе в качестве ограничение указано class, то есть любой класс, то в производном классе также надо указать в качестве ограничения class, либо же какой-то конкретный класс.

Обработка исключений
Конструкция try..catch..finally
Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются исключениями. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция try...catch...finally.
```
try
{

}
catch
{

}
finally
{

}
```
При использовании блока try...catch..finally вначале выполняются все инструкции в блоке try. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок finally. И затем конструкция try..catch..finally завершает свою работу.
Если же в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается, и среда CLR начинает искать блок catch, который может обработать данное исключение. Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.
Если нужный блок catch не найден, то при возникновении исключения программа аварийно завершает свое выполнение.
Рассмотрим следующий пример:
```
class Program
{
    static void Main(string[] args)
    {
        int x = 5;
        int y = x / 0;
        Console.WriteLine($"Результат: {y}");
        Console.WriteLine("Конец программы");
        Console.Read();
    }
}
```
В данном случае происходит деление числа на 0, что приведет к генерации исключения. И при запуске приложения в режиме отладки мы увидим в Visual Studio окошко, которое информирует об исключении:

В этом окошке мы видим, что возникло исключение, которое представляет тип System.DivideByZeroException, то есть попытка деления на ноль. С помощью пункта View Details можно посомтреть более детальную информацию об исключении.
И в этом случае единственное, что нам остается, это завершить выполнение программы.
Чтобы избежать подобного аварийного завершения программы, следует использовать для обработки исключений конструкцию try...catch...finally. Так, перепишем пример следующим образом:
```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 5;
            int y = x / 0;
            Console.WriteLine($"Результат: {y}");
        }
```

```
        catch
        {
            Console.WriteLine("Возникло исключение!");
        }
        finally
        {
            Console.WriteLine("Блок finally");
        }
        Console.WriteLine("Конец программы");
        Console.Read();
    }
}
```

В данном случае у нас опять же возникнет исключение в блоке try, так как мы пытаемся разделить на ноль. И дойдя до строки

```
int y = x / 0;
```

выполнение программы остановится. CLR найдет блок catch и передаст управление этому блоку.

После блока catch будет выполняться блок finally.

```
Возникло исключение!
Блок finally
Конец программы
```

Таким образом, программа по-прежнему не будет выполнять деление на ноль и соответственно не будет выводить результат этого деления, но теперь она не будет аварийно завершаться, а исключение будет обрабатываться в блоке catch.

Следует отметить, что в этой конструкции обязателен блок try. При наличии блока catch мы можем опустить блок finally:

```
try
{
    int x = 5;
    int y = x / 0;
    Console.WriteLine($"Результат: {y}");
}
catch
{
    Console.WriteLine("Возникло исключение!");
}
```

И, наоборот, при наличии блока finally мы можем опустить блок catch и не обрабатывать исключение:

```
try
{
    int x = 5;
    int y = x / 0;
    Console.WriteLine($"Результат: {y}");
}
finally
{
    Console.WriteLine("Блок finally");
}
```

Однако, хотя с точки зрения синтаксиса C# такая конструкция вполне корректна, тем не менее, поскольку CLR не сможет найти нужный блок catch, то исключение не будет обработано, и программа аварийно завершится.

Обработка исключений и условные конструкции

Ряд исключительных ситуаций может быть предвиден разработчиком. Например, пусть программа предусматривает ввод числа и вывод его квадрата:

```
static void Main(string[] args)
{
    Console.WriteLine("Введите число");
    int x = Int32.Parse(Console.ReadLine());

    x *= x;
    Console.WriteLine("Квадрат числа: " + x);
    Console.Read();
}
```

Если пользователь введет не число, а строку, какие-то другие символы, то программа выпадет в ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок try..catch, чтобы обработать возможную ошибку. Однако гораздо оптимальнее было бы проверить допустимость преобразования:

```
static void Main(string[] args)
{
    Console.WriteLine("Введите число");
    int x;
    string input = Console.ReadLine();
    if (Int32.TryParse(input, out x))
    {
        x *= x;
        Console.WriteLine("Квадрат числа: " + x);
    }
    else
    {
        Console.WriteLine("Некорректный ввод");
    }
    Console.Read();
}
```

Метод Int32.TryParse() возвращает true, если преобразование можно осуществить, и false - если нельзя. При допустимости преобразования переменная x будет содержать введенное число. Так, не используя try...catch можно обработать возможную исключительную ситуацию.

С точки зрения производительности использование блоков try..catch более накладно, чем применение условных конструкций. Поэтому по возможности вместо try..catch лучше использовать условные конструкции на проверку исключительных ситуаций.

Блок catch и фильтры исключений

Определение блока catch

За обработку исключения отвечает блок catch, который может иметь следующие формы:

-

```
catch
{
    // выполняемые инструкции
}
```

- Обрабатывает любое исключение, которое возникло в блоке try. Выше уже был продемонстрирован пример подобного блока.

-

```
catch (тип_исключения)
{
    // выполняемые инструкции
}
```

- Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch.
- Например, обработаем только исключения типа DivideByZeroException:

```
try
{
    int x = 5;
    int y = x / 0;
    Console.WriteLine($"Результат: {y}");
}
catch(DivideByZeroException)
{
    Console.WriteLine("Возникло исключение DivideByZeroException");
}
```

- Однако если в блоке try возникнут исключения каких-то других типов, отличных от DivideByZeroException, то они не будут обработаны.

-

```
catch (тип_исключения имя_переменной)
{
    // выполняемые инструкции
}
```

- Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch. А вся информация об исключении помещается в переменную данного типа. Например:

```
try
{
   int x = 5;
   int y = x / 0;
   Console.WriteLine($"Результат: {y}");
}
catch(DivideByZeroException ex)
{
   Console.WriteLine($"Возникло исключение {ex.Message}");
}
```

- Фактически этот случай аналогичен предыдущему за тем исключением, что здесь используется переменная. В данном случае в переменную ex, которая представляет тип DivideByZeroException, помещается информация о возникшем исключени. И с помощью свойства Message мы можем получить сообщение об ошибке.
- Если нам не нужна информация об исключении, то переменную можно не использовать как в предыдущем случае.

Фильтры исключений

Фильтры исключений позволяют обрабатывать исключения в зависимости от определенных условий. Для их применения после выражения catch идет выражение when, после которого в скобках указывается условие:

```
catch when(условие)
{

}
```

В этом случае обработка исключения в блоке catch производится только в том случае, если условие в выражении when истинно. Например:

```
int x = 1;
int y = 0;

try
{
   int result = x / y;
}
catch(DivideByZeroException) when (y==0 && x == 0)
{
   Console.WriteLine("y не должен быть равен 0");
}
catch(DivideByZeroException ex)
{
   Console.WriteLine(ex.Message);
}
```

В данном случае будет выброшено исключение, так как y=0. Здесь два блока catch, и оба они обрабатывают исключения типа DivideByZeroException, то есть по сути все исключения, генерируемые при делении на ноль. Но поскольку для первого блока указано условие y == 0 && x == 0, то оно не будет обрабатывать исключение - условие, указанное после оператора when возвращает false. Поэтому CLR будет дальше искать соответствующие блоки catch далее и для обработки исключения выберет второй блок catch. В итоге если мы уберем второй блок catch, то исключение вобще не будет обрабатываться.

Типы исключений. Класс Exception

Базовым для всех типов исключений является тип Exception. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

- InnerException: хранит информацию об исключении, которое послужило причиной текущего исключения
- Message: хранит сообщение об исключении, текст ошибки
- Source: хранит имя объекта или сборки, которое вызвало исключение
- StackTrace: возвращает строковое представление стека вызывов, которые привели к возникновению исключения

- TargetSite: возвращает метод, в котором и было вызвано исключение

Например, обработаем исключения типа Exception:

```
static void Main(string[] args)
{
    try
    {
        int x = 5;
        int y = x / 0;
        Console.WriteLine($"Результат: {y}");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Исключение: {ex.Message}");
        Console.WriteLine($"Метод: {ex.TargetSite}");
        Console.WriteLine($"Трассировка стека: {ex.StackTrace}");
    }

    Console.Read();
}
```

Однако так как тип Exception является базовым типом для всех исключений, то выражение catch (Exception ex) будет обрабатывать все исключения, которые могут возникнуть.

Но также есть более специализированные типы исключений, которые предназначены для обработки каких-то определенных видов исключений. Их довольно много, я приведу лишь некоторые:

- DivideByZeroException: представляет исключение, которое генерируется при делении на ноль
- ArgumentOutOfRangeException: генерируется, если значение аргумента находится вне диапазона допустимых значений
- ArgumentException: генерируется, если в метод для параметра передается некорректное значение
- IndexOutOfRangeException: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
- InvalidCastException: генерируется при попытке произвести недопустимые преобразования типов
- NullReferenceException: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)

И при необходимости мы можем разграничить обработку различных типов исключений, включив дополнительные блоки catch:

```
static void Main(string[] args)
{
    try
    {
        int[] numbers = new int[4];
        numbers[7] = 9;     // IndexOutOfRangeException

        int x = 5;
        int y = x / 0;  // DivideByZeroException
        Console.WriteLine($"Результат: {y}");
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("Возникло исключение DivideByZeroException");
    }
    catch (IndexOutOfRangeException ex)
    {
        Console.WriteLine(ex.Message);
    }

    Console.Read();
}
```

В данном случае блоки catch обрабатывают исключения типов IndexOutOfRangeException, DivideByZeroException и Exception. Когда в блоке try возникнет исключение, то CLR будет искать нужный блок catch для обработки исключения. Так, в данном случае на строке
```
numbers[7] = 9;
```
происходит обращение к 7-му элементу массива. Однако поскольку в массиве только 4 элемента, то мы получим исключение типа IndexOutOfRangeException. CLR найдет блок catch, который обрабатывает данное исключение, и передаст ему управление.
Следует отметить, что в данном случае в блоке try есть ситуация для генерации второго исключения - деление на ноль. Однако поскольку после генерации IndexOutOfRangeException управление переходит в соответствующий блок catch, то деление на ноль int y = x / 0 в принципе не будет выполняться, поэтому исключение типа DivideByZeroException никогда не будет сгенерировано.
Однако рассмотрим другую ситуацию:
```
static void Main(string[] args)
{
    try
    {
        object obj = "you";
        int num = (int)obj;     // InvalidCastException
        Console.WriteLine($"Результат: {num}");
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("Возникло исключение DivideByZeroException");
    }
    catch (IndexOutOfRangeException)
    {
        Console.WriteLine("Возникло исключение IndexOutOfRangeException");
    }

    Console.Read();
}
```
В данном случае в блоке try генерируется исключение типа InvalidCastException, однако соответствующего блока catch для обработки данного исключения нет. Поэтому программа аварийно завершит свое выполнение.
Мы также можем определить для InvalidCastException свой блок catch, однако суть в том, что теоретически в коде могут быть сгенерированы сами различные типы исключений. А определять для всех типов исключений блоки catch, если обработка исключений однотипна, не имеет смысла. И в этом случае мы определить блок catch для базового типа Exception:
```
static void Main(string[] args)
{
    try
    {
        object obj = "you";
        int num = (int)obj;     // InvalidCastException
        Console.WriteLine($"Результат: {num}");
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("Возникло исключение DivideByZeroException");
    }
    catch (IndexOutOfRangeException)
    {
        Console.WriteLine("Возникло исключение IndexOutOfRangeException");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Исключение: {ex.Message}");
    }
    Console.Read();
}
```
И в данном случае блок catch (Exception ex){} будет обрабатывать все исключения кроме DivideByZeroException и IndexOutOfRangeException. При этом блоки catch для более общих,

более базовых исключений следует помещать в конце - после блоков catch для более конкретный, специализированных типов. Так как CLR выбирает для обработки исключения первый блок catch, который соответствует типу сгенерированного исключения. Поэтому в данном случае сначала обрабатывается исключение DivideByZeroException и IndexOutOfRangeException, и только потом Exception (так как DivideByZeroException и IndexOutOfRangeException наследуется от класса Exception).

Создание классов исключений

Если нас не устраивают встроенные типы исключений, то мы можем создать свои типы. Базовым классом для всех исключений является класс Exception, соответственно для создания своих типов мы можем унаследовать данный класс.

Допустим, у нас в программе будет ограничение по возрасту:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Person p = new Person { Name = "Tom", Age = 17 };
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Ошибка: {ex.Message}");
        }
        Console.Read();
    }
}
class Person
{
    private int age;
    public string Name { get; set; }
    public int Age
    {
        get { return age; }
        set
        {
            if (value < 18)
            {
                throw new Exception("Лицам до 18 регистрация запрещена");
            }
            else
            {
                age = value;
            }
        }
    }
}
```

В классе Person при установке возраста происходит проверка, и если возраст меньше 18, то выбрасывается исключение. Класс Exception принимает в конструкторе в качестве параметра строку, которое затем передается в его свойство Message.

Но иногда удобнее использовать свои классы исключений. Например, в какой-то ситуации мы хотим обработать определенным образом только те исключения, которые относятся к классу Person. Для этих целей мы можем сделать специальный класс PersonException:

```
class PersonException : Exception
{
    public PersonException(string message)
        : base(message)
    { }
}
```

По сути класс кроме пустого конструктора ничего не имеет, и то в конструкторе мы просто обращаемся к конструктору базового класса Exception, передавая в него строку message. Но теперь мы можем изменить класс Person, чтобы он выбрасывал исключение именно этого типа и соответственно в основной программе обрабатывать это исключение:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Person p = new Person { Name = "Tom", Age = 17 };
        }
        catch (PersonException ex)
        {
            Console.WriteLine("Ошибка: " + ex.Message);
        }
        Console.Read();
    }
}
class Person
{
    private int age;
    public int Age
    {
        get { return age; }
        set
        {
            if (value < 18)
                throw new PersonException("Лицам до 18 регистрация запрещена");
            else
                age = value;
        }
    }
}
```

Однако необязательно наследовать свой класс исключений именно от типа Exception, можно взять какой-нибудь другой производный тип. Например, в данном случае мы можем взять тип ArgumentException, который представляет исключение, генерируемое в результате передачи аргументу метода некорректного значения:

```
class PersonException : ArgumentException
{
    public PersonException(string message)
        : base(message)
    { }
}
```

Каждый тип исключений может определять какие-то свои свойства. Например, в данном случае мы можем определить в классе свойство для хранения устанавливаемого значения:

```
class PersonException : ArgumentException
{
    public int Value { get;}
    public PersonException(string message, int val)
        : base(message)
    {
        Value = val;
    }
}
```

В конструкторе класса мы устанавливаем это свойство и при обработке исключения мы его можем получить:

```
class Person
{
    public string Name { get; set; }
    private int age;
    public int Age
    {
        get { return age; }
        set
        {
            if (value < 18)
                throw new PersonException("Лицам до 18 регистрация запрещена", value);
            else
                age = value;
```

```
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Person p = new Person { Name = "Tom", Age = 13 };
        }
        catch (PersonException ex)
        {
            Console.WriteLine($"Ошибка: {ex.Message}");
            Console.WriteLine($"Некорректное значение: {ex.Value}");
        }
        Console.Read();
    }
}
```

Поиск блока catch при обработке исключений

Если код, который вызывает исключение, не размещен в блоке try или помещен в конструкцию try..catch, которая не содержит соответствующего блока catch для обработки возникшего исключения, то система производит поиск соответствующего обработчика исключения в стеке вызовов.

Например, рассмотрим следующую программу:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                TestClass.Method1();
            }
            catch (DivideByZeroException ex)
            {
                Console.WriteLine($"Catch в Main : {ex.Message}");
            }
            finally
            {
                Console.WriteLine("Блок finally в Main");
            }
            Console.WriteLine("Конец метода Main");
            Console.Read();
        }
    }
    class TestClass
    {
        public static void Method1()
        {
            try
            {
                Method2();
            }
            catch (IndexOutOfRangeException ex)
            {
                Console.WriteLine($"Catch в Method1 : {ex.Message}");
            }
            finally
            {
                Console.WriteLine("Блок finally в Method1");
            }
```

```
            Console.WriteLine("Конец метода Method1");
        }
        static void Method2()
        {
            try
            {
                int x = 8;
                int y = x / 0;
            }
            finally
            {
                Console.WriteLine("Блок finally в Method2");
            }
            Console.WriteLine("Конец метода Method2");
        }
    }
}
```

В данном случае стек вызовов выглядит следующим образом: метод Main вызывает метод Method1, который, в свою очередь, вызывает метод Method2. И в методе Method2 генерируется исключение DivideByZeroException. Визуально стек вызовов можно представить следующим образом:

Внизу стека метод Main, с которого началось выполнение, и на самом верху метод Method2. Что будет происходить в данном случае при генерации исключения?

1. Метод Main вызывает метод Method1, а тот вызывает метод Method2, в котором генерируется исключение DivideByZeroException.
2. Система видит, что код, который вызывал исключение, помещен в конструкцию try..catch

```
try
{
    int x = 8;
    int y = x / 0;
}
finally
{
    Console.WriteLine("Блок finally в Method2");
}
```

3. Система ищет в этой конструкции блок catch, который обрабатывает исключение DivideByZeroException. Однако такого блока catch нет.
4. Система опускается в стеке вызовов в метод Method1, который вызывал Method2. Здесь вызов Method2 помещен в конструкцию try..catch

```
try
{
    Method2();
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine($"Catch в Method1 : {ex.Message}");
}
finally
{
    Console.WriteLine("Блок finally в Method1");
}
```

5. Система также ищет в этой конструкции блок catch, который обрабатывает исключение DivideByZeroException. Однако здесь также подобный блок catch отсутствует.
6. Система далее опускается в стеке вызовов в метод Main, который вызывал Method1. Здесь вызов Method1 помещен в конструкцию try..catch

```
try
{
    TestClass.Method1();
}
catch (DivideByZeroException ex)
{
```

```
    Console.WriteLine($"Catch в Main : {ex.Message}");
}
finally
{
    Console.WriteLine("Блок finally в Main");
}
```

7. Система снова ищет в этой конструкции блок catch, который обрабатывает исключение DivideByZeroException. И в данном случае ткой блок найден.
8. Система наконец нашла нужный блок catch в методе Main, для обработки исключения, которое возникло в методе Method2 - то есть к начальному методу, где непосредственно возникло исключение. Но пока данный блок catch НЕ выполняется. Система поднимается обратно по стеку вызовов в самый верх в метод Method2 и выполняет в нем блок finally:

```
finally
{
    Console.WriteLine("Блок finally в Method2");
}
```

9. Далее система возвращается по стеку вызовов вниз в метод Method1 и выполняет в нем блок finally:

```
finally
{
    Console.WriteLine("Блок finally в Method1");
}
```

10. Затем система переходит по стеку вызовов вниз в метод Main и выполняет в нем найденный блок catch и последующий блок finally:

```
catch (DivideByZeroException ex)
{
    Console.WriteLine($"Catch в Main : {ex.Message}");
}
finally
{
    Console.WriteLine("Блок finally в Main");
}
```

11. Далее выполняется код, который идет в методе Main после конструкции try..catch:

```
Console.WriteLine("Конец метода Main");
```

12. Стоит отметить, что код, который идет после конструкции try...catch в методах Method1 и Method2, не выполняется, потому что обработчик исключения найден именно в методе Main.

Консольный вывод программы:

```
Блок finally в Method2
Блок finally в Method1
Catch в Main: Попытка деления на нуль.
Блок finally в Main
Конец метода Main
```

Генерация исключения и оператор throw

Обычно система сама генерирует исключения при определенных ситуациях, например, при делении числа на ноль. Но язык C# также позволяет генерировать исключения вручную с помощью оператора throw. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения.

Например, в нашей программе происходит ввод строки, и мы хотим, чтобы, если длина строки будет больше 6 символов, возникало исключение:

```
static void Main(string[] args)
{
    try
    {
        Console.Write("Введите строку: ");
        string message = Console.ReadLine();
        if (message.Length > 6)
        {
            throw new Exception("Длина строки больше 6 символов");
        }
    }
```

```
    catch (Exception e)
    {
        Console.WriteLine($"Ошибка: {e.Message}");
    }
    Console.Read();
}
```

После оператора throw указывается объект исключения, через конструктор которого мы можем передать сообщение об ошибке. Естественно вместо типа Exception мы можем использовать объект любого другого типа исключений.

Затем в блоке catch сгенерированное нами исключение будет обработано.

Подобным образом мы можем генерировать исключения в любом месте программы. Но существует также и другая форма использования оператора throw, когда после данного оператора не указывается объект исключения. В подобном виде оператор throw может использоваться только в блоке catch:

```
try
{
    try
    {
        Console.Write("Введите строку: ");
        string message = Console.ReadLine();
        if (message.Length > 6)
        {
            throw new Exception("Длина строки больше 6 символов");
        }
    }
    catch
    {
        Console.WriteLine("Возникло исключение");
        throw;
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

В данном случае при вводе строки с длиной больше 6 символов возникнет исключение, которое будет обработано внутренним блоком catch. Однако поскольку в этом блоке используется оператор throw, то исключение будет передано дальше внешнему блоку catch.

Интерфейсы
Введение в интерфейсы
Интерфейс представляет ссылочный тип, который определяет набор методов и свойств, но не реализует их. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.
Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования. Например, интерфейс IMovable:
```
interface IMovable
{
   void Move();
}
```
У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не принимает никаких параметров и ничего не возвращает.
Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.
В целом интерфейсы могут определять следующие сущности:
- Методы
- Свойства
- Индексаторы
- События

Однако интерфейсы не могут определять статические члены, переменные, константы.
Затем какой-нибудь класс или структура могут применить данный интерфейс:
```
// применение интерфейса в классе
class Person : IMovable
{
   public void Move()
   {
      Console.WriteLine("Человек идет");
   }
}
// применение интерфейса в структуре
struct Car : IMovable
{
   public void Move()
   {
      Console.WriteLine("Машина едет");
   }
}
```
При применени интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода.
Если класс или структура не реализуют какие-либо свойства или методы интерфейса, то мы столкнемся с ошибкой на этапе компиляции.
Применение интерфейса в программе:
```
using System;

namespace HelloApp
{
   interface IMovable
   {
```

```
        void Move();
    }
    class Person : IMovable
    {
        public void Move()
        {
            Console.WriteLine("Человек идет");
        }
    }
    struct Car : IMovable
    {
        public void Move()
        {
            Console.WriteLine("Машина едет");
        }
    }
    class Program
    {
        static void Action(IMovable movable)
        {
            movable.Move();
        }
        static void Main(string[] args)
        {
            Person person = new Person();
            Car car = new Car();
            Action(person);
            Action(car);
            Console.Read();
        }
    }
}
```

В данной программе определен метод Action(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод.

Иными словами, интерфейс - это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Консольный вывод данной программы:

```
Человек идет
Машина едет
```

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы:

```
interface IMovable
{
    void Move();
}
abstract class Person : IMovable
{
    public abstract void Move();
}
class Driver : Person
{
    public override void Move()
    {
        Console.WriteLine("Шофер ведет машину");
    }
}
```

Стоит отметить, что в Visual Studio есть специальный компонент для добавления нового интерфейса в отдельном файле. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать Add-> New Item... и в диалоговом окне добавления нового компонента выбрать пункт Interface:

При реализации интерфейса учитываются также методы и свойства, унаследованные от базового класса. Например:

```
interface IAction
{
    void Move();
}
class BaseAction
{
    public void Move()
    {
        Console.WriteLine("Move in BaseAction");
    }
}
class HeroAction : BaseAction, IAction
{
}
```

Здесь класс HeroAction реализует интерфейс IAction, однако для реализации метода Move из интерфейса применяется метод Move, унаследованный от базового класса BaseAction. Таким образом, класс HeroAction может не реализовать метод Move, так как этот метод уже определен в базовом классе BaseAction.

Следует отметить, что если класс одновременно наследует другой класс и реализует интерфейс, как в примере выше класс HeroAction, то название базового класса должно быть указано до реализуемых интерфейсов: class HeroAction : `BaseAction`, IAction

Множественная реализация интерфейсов

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса, в отличие, скажем, от языка C++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

```
myClass: myInterface1, myInterface2, myInterface3, ...
{

}
```

Рассмотрим на примере:

```
using System;

namespace HelloApp
{
    interface IAccount
    {
        int CurrentSum { get; }  // Текущая сумма на счету
        void Put(int sum);      // Положить деньги на счет
        void Withdraw(int sum); // Взять со счета
    }
    interface IClient
    {
        string Name { get; set; }
    }
    class Client : IAccount, IClient
    {
        int _sum; // Переменная для хранения суммы
        public string Name { get; set; }
        public Client(string name, int sum)
        {
            Name = name;
            _sum = sum;
        }

        public int CurrentSum { get { return _sum; } }

        public void Put(int sum) {  _sum += sum; }

        public void Withdraw(int sum)
        {
```

```
            if (_sum >= sum)
            {
                _sum -= sum;
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Client client = new Client("Tom", 200);
            client.Put(30);
            Console.WriteLine(client.CurrentSum); //230
            client.Withdraw(100);
            Console.WriteLine(client.CurrentSum); //130
            Console.Read();
        }
    }
}
```

В данном случае определены два интерфейса. Интерфейс IAccount определяет свойство CurrentSum для текущей суммы денег на счете и два метода Put и Withdraw для добавления денег на счет и изъятия денег. Интерфейс IClient определяет свойство для хранения имени клиента.

Обатите внимание, что свойства CurrentSum и Name в интерфейсах похожи на автосвойства, но это не автосвойства. При реализации мы можем развернуть их в полноценные свойства, либо же сделать автосвойствами.

Класс Client реализует оба интерфейса и затем применяется в программе.

Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс IAccount, то переменная типа IAccount может хранить ссылку на объект типа Client:

```
// Все объекты Client являются объектами IAccount
IAccount account = new Client("Том", 200);
account.Put(200);
Console.WriteLine(account.CurrentSum); // 400
// Не все объекты IAccount являются объектами Client, необходимо явное приведение
Client client = (Client)account;
// Интерфейс IAccount не имеет свойства Name, необходимо явное приведение
string clientName = ((Client)account).Name;
```

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект Client реализует интерфейс IAccount.

Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному. Так как не каждый объект IAccount является объектом Client (ведь интерфейс IAccount могут реализовать и другие классы), то для подобного преобразования необходима операция приведения типов. И если мы хотим обратиться к методам класса Client, которые не определены в интерфейсе IAccount, но являются частью класса Client, то нам надо явным образом выполнить преобразование типов:

```
string clientName = ((Client)account).Name;
```

Дополнительно об интерфейсах

Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:

```
interface IAction
{
    void Move();
}
interface IRunAction : IAction
{
    void Run();
}
class BaseAction : IRunAction
```

```
{
    public void Move()
    {
        Console.WriteLine("Move");
    }
    public void Run()
    {
        Console.WriteLine("Run");
    }
}
```
При применении этого интерфейса класс BaseAction должен будет реализовать как методы и свойства интерфейса IRunAction, так и методы и свойства базового интерфейса IAction.
Однако в отличие от классов мы не можем применять к интерфейсам модификатор sealed (а также к методам интерфейсов), чтобы запретить наследование интерфейсов.
Также мы не можем применять к интерфейсам модификатор abstract, поскольку интерфейс фактически итак предоставляет абстрактный функционал, который должен быть реализован в классе.
Однако методы интерфейсов могут использовать ключевое слово new для сокрытия методов из базового интерфейса:
```
class RunAction : IRunAction
{
    public void Move()
    {
        Console.WriteLine("I am running");
    }
}

interface IAction
{
    void Move();
}
interface IRunAction : IAction
{
    new void Move();
}
```
Здесь метод Move из IRunAction скрывает метод Move из базового интерфейса IAction. Большого смысла в этом нет, так как в данном случае нечего скрывать, то тем не менее мы так можем делать. А класс RunAction реализует метод Move сразу для обоих интерфейсов.

Модификаторы доступа интерфейсов

Как и классы, интерфейсы по умолчанию имеют уровень доступа internal, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным:
```
public interface IAction
{
    void Move();
}
```
Но при наследовании интерфейсов, как и при наследовании классов, следует учитывать, что производный интерфейс должен иметь тот же уровень доступа или более строгий, чем базовый интерфейс. Например:
```
public interface IAction
{
    void Move();
}
internal interface IRunAction : IAction
{
    void Run();
}
```
Но не наоборот. Например, в следующем случае мы получим ошибку, и программа не скомпилируется, так как производный интерфейс имеет менее строгий уровень доступа, нежели базовый:
```
internal interface IAction
{
    void Move();
```

```
}
public interface IRunAction : IAction   // ошибка IRunAction может быть только internal
{
   void Run();
}
```

Изменение реализации интерфейсов в производных классах

Может сложиться ситуация, что базовый класс реализовал интерфейс, но в классе-наследнике необходимо изменить реализацию этого интерфейса. Что в этом случае делать? В этом случае мы можем использовать либо переопределение, либо сокрытие метода или свойства интерфейса.

Первый вариант - переопределение виртуальных/абстрактных методов:

```
interface IAction
{
   void Move();
}
class BaseAction : IAction
{
   public virtual void Move()
   {
      Console.WriteLine("Move in BaseAction");
   }
}
class HeroAction : BaseAction
{
   public override void Move()
   {
      Console.WriteLine("Move in HeroAction");
   }
}
```

В базовом классе BaseAction реализованный метод интерфейса определен как виртуальный (можно было бы также сделать его абстрактным), а в производном классе он переопределен.

При вызове метода через переменную интерфейса, если она ссылается на объект производного класса, будет использоваться реализация из производного класса:

```
BaseAction action1 = new HeroAction();
action1.Move();          // Move in HeroAction

IAction action2 = new HeroAction();
action2.Move();          // Move in HeroAction
```

Второй вариант - сокрытие метода в производном классе:

```
interface IAction
{
   void Move();
}
class BaseAction : IAction
{
   public void Move()
   {
      Console.WriteLine("Move in BaseAction");
   }
}
class HeroAction : BaseAction
{
   public new void Move()
   {
      Console.WriteLine("Move in HeroAction");
   }
}
```

Также используем эти классы:

```
BaseAction action1 = new HeroAction();
action1.Move();          // Move in BaseAction

IAction action2 = new HeroAction();
action2.Move();          // Move in BaseAction
```

Так как интерфейс реализован именно в классе BaseAction, то через переменную action2 можно обратиться только к реализации метода Move из базового класса BaseAction.
Третий вариант - повторная реализация интерфейса в классе-наследнике:
```
interface IAction
{
   void Move();
}
class BaseAction : IAction
{
   public void Move()
   {
      Console.WriteLine("Move in BaseAction");
   }
}
class HeroAction : BaseAction, IAction
{
   public new void Move()
   {
      Console.WriteLine("Move in HeroAction");
   }
}
```
В этом случае реализации этого метода из базового класса будет игнорироваться:
```
BaseAction action1 = new HeroAction();
action1.Move();          // Move in BaseAction

IAction action2 = new HeroAction();
action2.Move();           // Move in HeroAction
```
Также стоит отметить, что в случае с переменной action1 по-прежнему действует ранее связывание, в силу которого через эту переменную можно вызвать реализацию метода Move только из базового класса, который эта переменная представляет.

Явное применение интерфейсов

Кроме неявного применения интерфейсов, которое было рассмотрено выше, сушествует явная реализация интерфейса. При явной реализации указывается название метода или свойства вместе с названием интерфейса, при этом мы не можем использовать модификатор public, то есть методы являются закрытыми:
```
interface IAction
{
   void Move();
}
class BaseAction : IAction
{
   void IAction.Move()
   {
      Console.WriteLine("Move in Base Class");
   }
}
```
Следует учитывать, что при явной реализации интерфейса его методы и свойства не являются частью интерфейса класса. Поэтому напрямую через объект класса мы к ним обратиться не сможем:
```
static void Main(string[] args)
{
   BaseAction action = new BaseAction();
   ((IAction)action).Move();   // необходимо приведение к типу IAction

   // или так
   IAction action2 = new BaseAction();
   action2.Move();

   Console.ReadKey();
}
```
В какой ситуации может действительно понадобиться явная реализация интерфейса? Например, когда класс применяет несколько интерфейсов, но они имеют один и тот же метод с одним и тем же возвращаемым результатом и одним и тем же набором параметров:
```
class Person : ISchool, IUniversity
```

```
{
    public void Study()
    {
        Console.WriteLine("Учеба в школе или в университете");
    }
}

interface ISchool
{
    void Study();
}

interface IUniversity
{
    void Study();
}
```
Класс Person определяет один метод Study(), создавая одну общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект Person как объект типа ISchool или IUniversity, результат метода будет один и тот же.
Чтобы разграничить реализуемые интерфейсы, надо явным образом применить интерфейс:
```
class Person : ISchool, IUniversity
{
    void ISchool.Study()
    {
        Console.WriteLine("Учеба в школе");
    }
    void  IUniversity.Study()
    {
        Console.WriteLine("Учеба в университете");
    }
}
```
Использование:
```
static void Main(string[] args)
{
    Person p = new Person();

    ((ISchool)p).Study();
    ((IUniversity)p).Study();

    Console.Read();
}
```
Другая ситуация, когда в базовом классе уже реализован интерфейс, но необходимо в производном классе по-своему реализовать интерфейс:
```
interface IAction
{
    void Move();
}
class BaseAction : IAction
{
    public void Move()
    {
        Console.WriteLine("Move in BaseAction");
    }
}
class HeroAction : BaseAction, IAction
{
    void IAction.Move()
    {
        Console.WriteLine("Move in HeroAction");
    }
}
```
Несмотря на то, что базовый класс BaseAction уже реализовал интерфейс IAction, но производный класс по-своему реализует его. Применение классов:
```
HeroAction action1 = new HeroAction();
action1.Move();          // Move in BaseAction
```

```
((IAction)action1).Move(); // Move in HeroAction

IAction action2 = new HeroAction();
action2.Move();            // Move in HeroAction
```

Интерфейсы в обобщениях

Интерфейсы как ограничения обобщений

Интерфейсы могут выступать в качестве ограничений обобщений. При этом если в качестве ограничения можно указаь только один класс, то интерфейсов можно указать несколько. Допустим, у нас есть следующие интерфейсы и класс, который их реализует:

```
interface IAccount
{
    int CurrentSum { get; } // Текущая сумма на счету
    void Put(int sum);      // Положить деньги на счет
    void Withdraw(int sum); // Взять со счета
}
interface IClient
{
    string Name { get; set; }
}
class Client : IAccount, IClient
{
    int _sum; // Переменная для хранения суммы
    public Client(string name, int sum)
    {
        Name = name;
        _sum = sum;
    }

    public string Name { get; set; }
    public int CurrentSum
    {
        get { return _sum; }
    }
    public void Put(int sum)
    {
        _sum += sum;
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
        }
    }
}
```

Используем выше перечисленные интерфейсы в качестве ограничений обобщенного класса:

```
class Transaction<T> where T: IAccount, IClient
{
    public void Operate(T acc1, T acc2, int sum)
    {
        if(acc1.CurrentSum >= sum)
        {
            acc1.Withdraw(sum);
            acc2.Put(sum);
            Console.WriteLine($"{acc1.Name} : {acc1.CurrentSum}\n{acc2.Name} : {acc2.CurrentSum}");
        }
    }
}
```

В данном случае параметр T представляет тип, который который реализует сразу два интерфейса IAccount и IClient. Например, выше определен класс Client, который реализует оба интерфейса, поэтому мы можем данным типом типизировать объекты Transaction:

```
Client account1 = new Client("Tom", 200);
Client account2 = new Client("Bob", 300);
Transaction<Client> transaction = new Transaction<Client>();
```

```
transaction.Operate(account1, account2, 150);
```
Также параметр T может представлять интерфейс, который наследуется от обоих интерфейсов:
```
interface IClientAccount : IAccount, IClient
{

}
class ClientAccount : IClientAccount
{
    int _sum;
    public ClientAccount(string name, int sum)
    {
        _sum = sum; Name = name;
    }
    public int CurrentSum { get { return _sum; } }

    public string Name { get; set; }

    public void Put(int sum)
    {
        _sum += sum;
    }
    public void Withdraw(int sum)
    {
        if (_sum >= sum) _sum -= sum;
    }
}
```
В этом случае объекты Transaction мы можем типизировать типом IClientAccount:
```
IClientAccount account3 = new ClientAccount("Alice", 400);
IClientAccount account4 = new ClientAccount("Kate", 500);
Transaction<IClientAccount> operation = new Transaction<IClientAccount>();
operation.Operate(account3, account4, 200);
```
Обобщенные интерфейсы
Как и классы, интерфейсы могут быть обобщенными:
```
interface IUser<T>
{
    T Id { get; }
}
class User<T> : IUser<T>
{
    T _id;
    public User(T id)
    {
        _id = id;
    }
    public T Id { get { return _id; } }
}
```
Интерфейс IUser типизирован параметром T, который при реализации интерфейса используется в классе User. В частности, переменная _id определена как T, что позволяет нам использовать для id различные типы.
Определим две реализации: одна в качестве параметра будет использовать тип int, а другая - тип string:
```
IUser<int> user1 = new User<int>(6789);
Console.WriteLine(user1.Id);    // 6789

IUser<string> user2 = new User<string>("12345");
Console.WriteLine(user2.Id);    // 12345
```
Также при реализации интерфейса мы можем явным образом указать, какой тип будет использоваться для параметра T:
```
class IntUser : IUser<int>
{
    int _id;
    public IntUser(int id)
    {
```

```
        _id = id;
    }
    public int Id { get { return _id; } }
}
```

Копирование объектов. Интерфейс ICloneable

Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности:

```
class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person { Name="Tom", Age = 23 };
        Person p2 = p1;
        p2.Name = "Alice";
        Console.WriteLine(p1.Name); // Alice

        Console.Read();
    }
}

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

В данном случае объекты p1 и p2 будут указывать на один и тот же объект в памяти, поэтому изменения свойств в переменной p2 затронут также и переменную p1.

Чтобы переменная p2 указывала на новый объект, но со значениями из p1, мы можем применить клонирование с помощью реализации интерфейса ICloneable:

```
public interface ICloneable
{
    object Clone();
}
```

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:

```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public object Clone()
    {
        return new Person { Name = this.Name, Age = this.Age };
    }
}
```

Использование:

```
class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person { Name="Tom", Age = 23 };
        Person p2 = (Person)p1.Clone();
        p2.Name = "Alice";
        Console.WriteLine(p1.Name); // Tom

        Console.Read();
    }
}
```

Теперь все нормально копируется, изменения в свойствах p2 не сказываются на свойствах в p1.

Для сокращения кода копирования мы можем использовать специальный метод MemberwiseClone(), который возвращает копию объекта:

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

```
    public int Age { get; set; }
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}
```
Этот метод реализует поверхностное (неглубокое) копирование. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект Company:
```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Company Work { get; set; }

    public object Clone()
    {
        return this.MemberwiseClone();
    }
}

class Company
{
    public string Name { get; set; }
}
```
В этом случае при копировании новая копия будет указывать на тот же объект Company:
```
Person p1 = new Person { Name="Tom", Age = 23, Work= new Company { Name = "Microsoft" } };
Person p2 = (Person)p1.Clone();
p2.Work.Name = "Google";
p2.Name = "Alice";
Console.WriteLine(p1.Name); // Tom
Console.WriteLine(p1.Work.Name); // Google - а должно быть Microsoft
```
Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять глубокое копирование:
```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Company Work { get; set; }

    public object Clone()
    {
        Company company = new Company { Name = this.Work.Name };
        return new Person
        {
            Name = this.Name,
            Age = this.Age,
            Work = company
        };
    }
}

class Company
{
    public string Name { get; set; }
}
```
Сортировка объектов. Интерфейс IComparable
Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется Sort() можно сразу отсортировать по возрастанию весь набор данных. Например:
```
int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };
Array.Sort(numbers);
foreach (int n in numbers)
    Console.WriteLine(n);
```

Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс IComparable. Он имеет всего один метод:

```
public interface IComparable
{
   int CompareTo(object o);
}
```

Метод CompareTo предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

- Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
- Равен нулю. Значит, оба объекта равны
- Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Например, имеется класс Person:

```
class Person : IComparable
{
   public string Name { get; set; }
   public int Age { get; set; }
   public int CompareTo(object o)
   {
      Person p = o as Person;
      if (p != null)
         return this.Name.CompareTo(p.Name);
      else
         throw new Exception("Невозможно сравнить два объекта");
   }
}
```

Здесь в качестве критерия сравнения выбрано свойство Name объекта Person. Поэтому при сравнении здесь фактически идет сравнение значения свойства Name текущего объекта и свойства Name объекта, переданного через параметр. Если вдруг объект не удастся привести к типу Person, то выбрасывается исключение.

Применение:

```
Person p1 = new Person { Name = "Bill", Age = 34 };
Person p2 = new Person { Name = "Tom", Age = 23 };
Person p3 = new Person { Name = "Alice", Age = 21 };

Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people);

foreach(Person p in people)
{
   Console.WriteLine("{0} - {1}", p.Name, p.Age);
}
```

Интерфейс IComparable имеет обобщенную версию, поэтому мы могли бы сократить и упростить его применение в классе Person:

```
class Person : IComparable<Person>
{
   public string Name { get; set; }
   public int Age { get; set; }
   public int CompareTo(Person p)
   {
      return this.Name.CompareTo(p.Name);
   }
}
```

Применение компаратора

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

```
interface IComparer
{
   int Compare(object o1, object o2);
}
```

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше - то число меньше нуля; если оба объекта равны, возвращается ноль.
Создадим компаратор объектов Person. Пусть он сравнивает объекты в зависимости от длины строки - значения свойства Name:

```
class PeopleComparer : IComparer<Person>
{
    public int Compare(Person p1, Person p2)
    {
        if (p1.Name.Length > p2.Name.Length)
            return 1;
        else if (p1.Name.Length < p2.Name.Length)
            return -1;
        else
            return 0;
    }
}
```

В данном случае используется обобщенная версия интерфейса IComparer, чтобы не делать излишних преобразований типов. Применение компаратора:

```
Person p1 = new Person { Name = "Bill", Age = 34 };
Person p2 = new Person { Name = "Tom", Age = 23 };
Person p3 = new Person { Name = "Alice", Age = 21 };

Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people, new PeopleComparer());

foreach(Person p in people)
{
    Console.WriteLine("{0} - {1}", p.Name, p.Age);
}
```

Объект компаратора указывается в качестве второго параметра метода Array.Sort(). При этом не важно, реализует ли класс Person интерфейс IComparable или нет. Правила сортировки, установленные компаратором, будут иметь больший приоритет. В начале будут идти объекты Person, у которых имена меньше, а в конце - у которых имена длиннее:

```
Tom - 23
Bill - 34
Alice - 21
```

Ковариантность и контравариантность обобщенных интерфейсов

Понятия ковариантности и контравариантности связаны с возможностью использовать в приложении вместо некоторого типа другой тип, который находится ниже или выше в иерархии наследования.
Имеется три возможных варианта поведения:

- Ковариантность: позволяет использовать более конкретный тип, чем заданный изначально
- Контравариантность: позволяет использовать более универсальный тип, чем заданный изначально
- Инвариантность: позволяет использовать только заданный тип

Начиная с .NET 4.0 в C# была добавлена возможность создания ковариантных и контравариантных обобщенных интерфейсов. Это функциональность повышает гибкость при использовании обобщенных интерфейсов в программе. По умолчанию все обобщенные интерфейсы, например, IAccout<T> являются инвариантными.
Для рассмотрения ковариантных и контравариантных интерфейсов возьмем следующие классы:

```
class Account
{
    public virtual void DoTransfer(int sum)
    {
    Console.WriteLine($"Клиент положил на счет {sum} долларов");
    }
```

```
}
class DepositAccount : Account
{
    public override void DoTransfer(int sum)
    {
        Console.WriteLine($"Клиент положил на депозитный счет {sum} долларов");
    }
}
```
Здесь определен класс обычного счета - Account и унаследованный от него класс DepositAccount. В классе Account определен метод, который выполняет условную операцию с счетом. Класс DepositAccount немного переопределяет этот интерфейс.

Ковариантные интерфейсы

Обобщенные интерфейсы могут быть ковариантными, если к универсальному параметру применяется ключевое слово out. Например:

```
interface IBank<out T>
{
    T CreateAccount(int sum);
}

class Bank<T> : IBank<T> where T : Account, new()
{
    public T CreateAccount(int sum)
    {
        T acc = new T();  // создаем счет
        acc.DoTransfer(sum);
        return acc;
    }
}
```
Обобщенный интерфейс IBank определяет метод CreateAccount для создания счета. При этом на момент определения интерфейса мы не знаем, какой тип будет представлять счет. Ключевое слово out в определении интерфейса указывает, что данный интерфейс будет ковариантным.
Класс Bank, который представляет условный банк, реализует этот интерфейс и возвращает из метода CreateAccount объект, который представляет либо класс Account, либо один из его наследников.
Применим данные типы в программе:

```
static void Main(string[] args)
{
    IBank<DepositAccount> depositBank = new Bank<DepositAccount>();
    Account acc1 = depositBank.CreateAccount(34);

    IBank<Account> ordinaryBank = new Bank<DepositAccount>();
    // или так
    // IBank<Account> ordinaryBank = depositBank;
    Account acc2 = ordinaryBank.CreateAccount(45);

    Console.Read();
}
```
То есть мы можем присвоить более общему типу IBank<Account> объект более конкретного типа IBank<DepositAccount> или Bank<DepositAccount>.
В то же время если бы мы не использовали ключевое слово out:

```
interface IBank<out T>
```
то мы столкнулись бы с ошибкой в строке

```
IBank<Account> ordinaryBank = depositBank;
```
Поскольку в этом случае невозможно было бы привести объект Bank<DepositAccount> к типу IBank<Account>
При создании ковариантного интерфейса надо учитывать, что универсальный параметр может использоваться только в качестве типа значения, возвращаемого методами интерфейса. Но не может использоваться в качестве типа аргументов метода или ограничения методов интерфейса.

Контравариантные интерфейсы

Для создания контравариантного интерфейса надо использовать ключевое слово in. Например, возьмем те же классы Account и DepositAccount и определим следующие типы:

```
interface ITransaction<in T>
{
   void DoOperation(T account, int sum);
}

class Transaction<T> : ITransaction<T> where T : Account
{
   public void DoOperation(T account, int sum)
   {
      account.DoTransfer(sum);
   }
}
```

Здесь определен интерфейс ITransaction, который представляет условную банковскую операцию. Ключевое слово in в определении интерфейса указывает, что этот интерфейс - контравариантный. Интерфейс определяет метод DoOperation, который принимает некоторый счет и выполняет с ним операцию.

Класс Transaction реализует этот интерфейс и реализует его метод DoOperation.

Применим эти типы в программе:

```
static void Main(string[] args)
{
   ITransaction<Account> accTransaction = new Transaction<Account>();
   accTransaction.DoOperation(new Account(), 400);

   ITransaction<DepositAccount> depAccTransaction = new Transaction<Account>();
   depAccTransaction.DoOperation(new DepositAccount(), 450);

   Console.Read();
}
```

Так как интерфейс ITransaction использует универсальный параметр с ключевым словом in, то он является контравариантным, поэтому в коде мы можем объект Transaction<Account> привести к типу ITransaction<DepositAccount>:

```
ITransaction<DepositAccount> depAccTransaction = new Transaction<Account>();
```

Если бы ключевое слово in не использовалось бы, то мы не смогли бы выполнить подобное приведение. То есть объект интерфейса с более универсальным типом приводится к объекту интерфейса с более конкретным типом.

При создании контрвариантного интерфейса надо учитывать, что универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к аргументам, используемым в качестве возвращаемых типов.

Делегаты, события и лямбды

Делегаты

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Определение делегатов

Для объявления делегата используется ключевое слово delegate, после которого идет возвращаемый тип, название и параметры. Например:

```
delegate void Message();
```

Делегат Message в качестве возвращаемого типа имеет тип void (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Рассмотрим примение этого делегата:

```
class Program
{
    delegate void Message(); // 1. Объявляем делегат

    static void Main(string[] args)
    {
        Message mes; // 2. Создаем переменную делегата
        if (DateTime.Now.Hour < 12)
        {
            mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода
        }
        else
        {
            mes = GoodEvening;
        }
        mes(); // 4. Вызываем метод
        Console.ReadKey();
    }
    private static void GoodMorning()
    {
        Console.WriteLine("Good Morning");
    }
    private static void GoodEvening()
    {
        Console.WriteLine("Good Evening");
    }
}
```

Здесь сначала мы определяем делегат:

```
delegate void Message(); // 1. Объявляем делегат
```

В данном случае делегат определяется внутри класса, но также можно определить делегат вне класса внутри пространства имен.

Для использования делегата объявляется переменная этого делегата:

```
Message mes; // 2. Создаем переменную делегата
```

С помощью свойства DateTime.Now.Hour получаем текущий час. И в зависимости от времени в делегат передается адрес определенного метода. Обратите внимание, что методы эти имеют то же возвращаемое значение и тот же набор параметров (в данном случае отсутствие параметров), что и делегат.

```
mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода
```

Затем через делегат вызывам метод, на который ссылается данный делегат:

```
mes(); // 4. Вызываем метод
```

Вызов делегата производится подобно вызову метода.

Посмотрим на примере другого делегата:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        // присваивание адреса метода через контруктор
        Operation del = Add; // делегат указывает на метод Add
```

```
        int result = del(4,5); // фактически Add(4, 5)
        Console.WriteLine(result);

        del = Multiply; // теперь делегат указывает на метод Multiply
        result = del(4, 5); // фактически Multiply(4, 5)
        Console.WriteLine(result);

        Console.Read();
    }
    private static int Add(int x, int y)
    {
        return x+y;
    }
    private static int Multiply (int x, int y)
    {
        return x * y;
    }
}
```

В данном случае делегат Operation возвращает значение типа int и имеет два параметра типа int. Поэтому этому делегату соответствует любой метод, который возвращает значение типа int и принимает два параметра типа int. В данном случае это методы Add и Multiply. То есть мы можем присвоить переменной делегата любой из этих методов и вызывать.

Поскольку делегат принимает два параметра типа int, то при его вызове необходимо передать значения для этих параметров: del(4,5).

Делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур.

```
class Math
{
    public int Sum(int x, int y) { return x + y; }
}
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        Math math = new Math();
        Operation del = math.Sum;
        int result = del(4, 5);     // math.Sum(4, 5)
        Console.WriteLine(result);  // 9

        Console.Read();
    }
}
```

Присвоение ссылки на метод

Выше переменной делегата напрямую присваивался метод. Есть еще один способ - создание объекта делегата с помощью конструктора, в который передается нужный метод:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        Operation del = Add;
        Operation del2 = new Operation(Add);

        Console.Read();
    }
    private static int Add(int x, int y) { return x + y; }
}
```

Оба способа равноценны.

Соответствие методов делегату

Как было написано выше, методы соответствуют делегату, если они имеют один и тот же возвращаемый тип и один и тот же набор параметров. Но надо учитывать, что во внимание также принимаются модификаторы ref и out. Например, пусть у нас есть делегат:

```
delegate void SomeDel(int a, double b);
```

Этому делегату соответствует, например, следующий метод:

```
void SomeMethod1(int g, double n) { }
```

А следующие методы НЕ соответствуют:

```
int SomeMethod2(int g, double n) { }
void SomeMethod3(double n, int g) { }
void SomeMethod4(ref int g, double n) { }
void SomeMethod5(out int g, double n) { g = 6; }
```

Здесь метод SomeMethod2 имеет другой возвращаемый тип, отличный от типа делегата. SomeMethod3 имеет другой набор параметров. Параметры SomeMethod4 и SomeMethod5 также отличаются от параметров делегата, поскольку имеют модификаторы ref и out.

Добавление методов в делегат

В примерах выше переменная делегата указывала на один метод. В реальности же делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип. Все методы в делегате попадают в специальный список - список вызова или invocation list. И пр вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот сиок ни один, а несколько методов:

```
class Program
{
    delegate void Message();

    static void Main(string[] args)
    {
        Message mes1 = Hello;
        mes1 += HowAreYou;  // теперь mes1 указывает на два метода
        mes1(); // вызываются оба метода - Hello и HowAreYou
        Console.Read();
    }
    private static void Hello()
    {
        Console.WriteLine("Hello");
    }
    private static void HowAreYou()
    {
        Console.WriteLine("How are you?");
    }
}
```

В данном случае в список вызова делегата mes1 добавляются два метода - Hello и HowAreYou. И при вызове mes1 вызываются сразу оба этих метода.

Для добавления делегатов применяется операция +=. Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новый созданный объект делеагата будет присвоен переменной mes1.

При добавлении делегатов следует учитывать, что мы можем добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться сстолько раз, сколь он был добавлен:

```
Message mes1 = Hello;
mes1 += HowAreYou;
mes1 += Hello;
mes1 += Hello;

mes1();
```

Консольный вывод:

```
Hello
How are you?
Hello
Hello
```

Подобным образом мы можем удалять методы из делегата с помощью операции -=:

```
static void Main(string[] args)
{
    Message mes1 = Hello;
    mes1 += HowAreYou;
    Message mes2 = HowAreYou;
    Message mes3 = mes1 + mes2;
    mes1(); // вызываются все методы из mes1 и mes2
    mes1 -= HowAreYou;  // удаляем метод HowAreYou
    mes1(); // вызывается метод Hello

    Console.Read();
}
```

При удалении методов из делегата фактически будет создаватья новый делегат, который в списке вызова методов будет содержать на один метод меньше.

При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.

Объединение делегатов

Делегаты можно объединять в другие делегаты. Например:

```
class Program
{
    delegate void Message();

    static void Main(string[] args)
    {
        Message mes1 = Hello;
        Message mes2 = HowAreYou;
        Message mes3 = mes1 + mes2; // объединяем делегаты
        mes3(); // вызываются все методы из mes1 и mes2

        Console.Read();
    }
    private static void Hello()
    {
        Console.WriteLine("Hello");
    }
    private static void HowAreYou()
    {
        Console.WriteLine("How are you?");
    }
}
```

В данном случае объект mes3 представляет объединение делегатов mes1 и mes2. Объединение делегатов значит, что в список вызова делегата mes3 попадут все методы из делегатов mes1 и mes2. И при вызове делегата mes3 все эти методы одновременно будут вызваны.

Вызов делегата

В примерах выше делегат вызывался как обычный метод. Если делегат принимал параметры, то при ее вызове для параметров передавались необходимые значения:

```
class Program
{
    delegate int Operation(int x, int y);
    delegate void Message();

    static void Main(string[] args)
    {
        Message mes = Hello;
        mes();
        Operation op = Add;
        op(3, 4);
        Console.Read();
    }
    private static void Hello() { Console.WriteLine("Hello"); }
    private static int Add(int x, int y) { return x + y; }
}
```

Другой способ вызова делегата пердставляет метод Invoke():

```
class Program
{
   delegate int Operation(int x, int y);
   delegate void Message();

   static void Main(string[] args)
   {
      Message mes = Hello;
      mes.Invoke();
      Operation op = Add;
      op.Invoke(3, 4);
      Console.Read();
   }
   private static void Hello() { Console.WriteLine("Hello"); }
   private static int Add(int x, int y) { return x + y; }
}
```

Если делегат принимает параметры, то в метод Invoke передаются значения для этих параметров.

Следует учитывать, что если делегат пуст, то есть в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение, как, например, в следующем случае:

```
Message mes = null;
//mes();       // ! Ошибка: делегат равен null

Operation op = Add;
op -= Add;      // делегат op пуст
op(3, 4);       // !Ошибка: делегат равен null
```

Поэтому при вызове делегата всегда лучше проверять, не равен ли он null. Либо можно использовать метод Invoke и оператор условного null:

```
Message mes = null;
mes?.Invoke();       // ошибки нет, делегат просто не вызывается

Operation op = Add;
op -= Add;          // делегат op пуст
op?.Invoke(3, 4);   // ошибки нет, делегат просто не вызывается
```

Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова (если в списке вызова несколько методов). Например:

```
class Program
{
   delegate int Operation(int x, int y);

   static void Main(string[] args)
   {
      Operation op = Subtract;
      op += Multiply;
      op += Add;
      Console.WriteLine(op(7, 2));    // Add(7,2) = 9
      Console.Read();
   }
   private static int Add(int x, int y) { return x + y; }
   private static int Subtract(int x, int y) { return x - y; }
   private static int Multiply(int x, int y) { return x * y; }
}
```

Делегаты как параметры методов

Также делегаты могут быть параметрами методов:

```
class Program
{
   delegate void GetMessage();

   static void Main(string[] args)
   {
      if (DateTime.Now.Hour < 12)
      {
         Show_Message(GoodMorning);
```

```
        }
        else
        {
            Show_Message(GoodEvening);
        }
        Console.ReadLine();
    }
    private static void Show_Message(GetMessage _del)
    {
        _del?.Invoke();
    }
    private static void GoodMorning()
    {
        Console.WriteLine("Good Morning");
    }
    private static void GoodEvening()
    {
        Console.WriteLine("Good Evening");
    }
}
```

Обобщенные делегаты

Делегаты могут быть обобщенными, например:

```
delegate T Operation<T, K>(K val);

class Program
{
    static void Main(string[] args)
    {
        Operation<decimal, int> op = Square;

        Console.WriteLine(op(5));
        Console.Read();
    }

    static decimal Square(int n)
    {
        return n * n;
    }
}
```

Применение делегатов

В прошлой теме подробно были рассмотрены делегаты. Однако данные примеры, возможно, не показывают истинной силы делегатов, так как нужные нам методы в данном случае мы можем вызвать и напрямую без всяких делегатов. Однако наиболее сильная сторона делегатов состоит в том, что они позволяют делегировать выполнение некоторому коду извне. И на момент написания программы мы можем не знать, что за код будет выполняться. Мы просто вызываем делегат. А какой метод будет непосредственно выполняться при вызове делегата, будет решаться потом. Кроме того, на основе делегатов создать функционал методов обратного вызова, уведомляя другие объекты о произошедших событиях.

Рассмотрим подробный пример. Пусть у нас есть класс, описывающий счет в банке:

```
class Account
{
    int _sum; // Переменная для хранения суммы

    public Account(int sum)
    {
        _sum = sum;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }
```

```
    public void Put(int sum)
    {
        _sum += sum;
    }

    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
        }
    }
}
```

Допустим, в случае вывода денег с помощью метода Withdraw нам надо как-то уведомлять об этом самого клиента и, может быть, другие объекты. Для этого создадим делегат AccountStateHandler. Чтобы использовать делегат, нам надо создать переменную этого делегата, а затем присвоить ему метод, который будет вызываться делегатом.

Итак, добавим в класс Account следующие строки:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(string message);
    // Создаем переменную делегата
    AccountStateHandler _del;

    // Регистрируем делегат
    public void RegisterHandler(AccountStateHandler del)
    {
        _del = del;
    }

    // Далее остальные строки класса Account
```

Здесь фактически проделываются те же шаги, что были выше, и есть практически все кроме вызова делегата. В данном случае у нас делегат принимает параметр типа string. Теперь изменим метод Withdraw следующим образом:

```
public void Withdraw(int sum)
{
    if (sum <= _sum)
    {
        _sum -= sum;

        if (_del != null)
            _del($"Сумма {sum} снята со счета");
    }
    else
    {
        if (_del != null)
            _del("Недостаточно денег на счете");
    }
}
```

Теперь при снятии денег через метод Withdraw мы сначала проверяем, имеет ли делегат ссылку на какой-либо метод (иначе он имеет значение null). И если метод установлен, то вызываем его, передавая соответствующее сообщение в качестве параметра.

Теперь протестируем класс в основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        // создаем банковский счет
        Account account = new Account(200);
        // Добавляем в делегат ссылку на метод Show_Message
        // а сам делегат передается в качестве параметра метода RegisterHandler
        account.RegisterHandler(new Account.AccountStateHandler(Show_Message));
        // Два раза подряд пытаемся снять деньги
        account.Withdraw(100);
```

```
        account.Withdraw(150);
        Console.ReadLine();
    }
    private static void Show_Message(String message)
    {
        Console.WriteLine(message);
    }
}
```

Запустив программу, мы получим два разных сообщения:

```
Сумма 100 снята со счета
Недостаточно денег на счете
```

Таким образом, мы создали механизм обратного вызова для класса Account, который срабатывает в случае снятия денег. Поскольку делегат объявлен внутри класса Account, то чтобы к нему получить доступ, используется выражение Account.AccountStateHandler.

Опять же может возникнуть вопрос: почему бы к коде метода Withdraw() не выводить сообщение о снятии денег? Зачем нужно задействовать какой-то делегат?

Дело в том, что не всегда у нас есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как эти классы будут использоваться. А использовать эти классы будет другой разработчик.

Так, здесь мы выводим сообщение на консоль. Однако для класса Account не важно, как это сообщение выводится. Классу Account даже не известно, что вообще будет делаться в результате списания денег. Он просто посылает уведомление об этом через делегат.

В результате, если мы создаем консольное приложение, мы можем через делегат выводить сообщение на консоль. Если мы создаем графическое приложение Windows Forms или WPF, то можно выводить сообщение в виде графического окна. А можно не просто выводить сообщение. А, например, записать при списании информацию об этом действии в файл или отправить уведомление на электронную почту. В общем любыми способами обработать вызов делегата. И способ обработки не будет зависеть от класса Account.

Хотя в примере наш делегат принимал адрес на один метод, в действительности он может указывать сразу на несколько методов. Кроме того, при необходимости мы можем удалить ссылки на адреса определенных методов, чтобы они не вызывались при вызове делегата. Итак, изменим в классе Account метод RegisterHandler и добавим новый метод UnregisterHandler, который будет удалять методы из списка методов делегата:

```
// Регистрируем делегат
public void RegisterHandler(AccountStateHandler del)
{
    Delegate mainDel = System.Delegate.Combine(del, _del);
    _del = mainDel as AccountStateHandler;
}
// Отмена регистрации делегата
public void UnregisterHandler(AccountStateHandler del)
{
    Delegate mainDel = System.Delegate.Remove(_del, del);
    _del = mainDel as AccountStateHandler;
}
```

В первом методе метод Combine объединяет делегаты _del и del в один, который потом присваивается переменной _del. Во втором методе метод Remove возвращает делегат, из списка вызовов которого удален делегат del. Теперь перейдем к основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200);
        Account.AccountStateHandler colorDelegate = new Account.AccountStateHandler(Color_Message);

        // Добавляем в делегат ссылку на методы
        account.RegisterHandler(new Account.AccountStateHandler(Show_Message));
        account.RegisterHandler(colorDelegate);
        // Два раза подряд пытаемся снять деньги
        account.Withdraw(100);
        account.Withdraw(150);
```

```
        // Удаляем делегат
        account.UnregisterHandler(colorDelegate);
        account.Withdraw(50);

        Console.ReadLine();
    }
    private static void Show_Message(String message)
    {
        Console.WriteLine(message);
    }
    private static void Color_Message(string message)
    {
        // Устанавливаем красный цвет символов
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(message);
        // Сбрасываем настройки цвета
        Console.ResetColor();
    }
}
```

В целях тестирования мы создали еще один метод - Color_Message, который выводит то же самое сообщение только красным цветом. Для первого делегата создается отдельная переменная. Но большой разницы между передачей обоих в метод account.RegisterHandler нет: просто в одном случае мы сразу передаем объект, создаваемый конструктором account.RegisterHandler(new Account.AccountStateHandler(Show_Message));
Во втором случае создаем переменную и ее уже передаем в метод account.RegisterHandler(colorDelegate);.
В строке account.UnregisterHandler(colorDelegate); этот метод удаляется из списка вызовов делегата, поэтому этот метод больше не будет срабатывать. Консольный вывод будет иметь следующую форму:

```
Сумма 150 снята со счета
Сумма 150 снята со счета
Недостаточно денег на счете
Недостаточно денег на счете
Сумма 50 снята со счета
```

Также мы можем использовать сокращенную форму добавления и удаления делегатов. Для этого перепишем методы RegisterHandler и UnregisterHandler следующим образом:

```
// Регистрируем делегат
public void RegisterHandler(AccountStateHandler del)
{
    _del += del; // добавляем делегат
}
// Отмена регистрации делегата
public void UnregisterHandler(AccountStateHandler del)
{
    _del -= del; // удаляем делегат
}
```

События

События позволяют сигнализируют системе о том, что произошло определенное действие.
События объявляются в классе с помощью ключевого слова event, после которого идет название делегата:

```
// Объявляем делегат
public delegate void AccountStateHandler(string message);
// Событие, возникающее при выводе денег
public event AccountStateHandler Withdrawn;
```

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат.
Итак, посмотрим на примере. Для этого возьмем класс Account из прошлой темы и изменим его следующим образом:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(string message);
    // Событие, возникающее при выводе денег
```

```
    public event AccountStateHandler Withdrawn;
    // Событие, возникающее при добавление на счет
    public event AccountStateHandler Added;

    int _sum; // Переменная для хранения суммы

    public Account(int sum)
    {
        _sum = sum;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added($"На счет поступило {sum}");
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
            if (Withdrawn != null)
                Withdrawn($"Сумма {sum} снята со счета");
        }
        else
        {
            if (Withdrawn != null)
                Withdrawn("Недостаточно денег на счете");
        }
    }
}
```

Здесь мы определили два события: Withdrawn и Added. Оба события объявлены как экземпляры делегата AccountStateHandler, поэтому для обработки этих событий потребуется метод, принимающий строку в качестве параметра.

Затем в методах Put и Withdraw мы вызываем эти события. Перед вызовом мы проверяем, закреплены ли за этими событиями обработчики (if (Withdrawn != null)). Так как эти события представляют делегат AccountStateHandler, принимающий в качестве параметра строку, то и при вызове событий мы передаем в них строку.

Теперь используем события в основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.Withdrawn += Show_Message;

        account.Withdraw(100);
        // Удаляем обработчик события
        account.Withdrawn -= Show_Message;

        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
    }
    private static void Show_Message(string message)
    {
```

```
        Console.WriteLine(message);
    }
}
```
Для прикрепления обработчика события к определенному событию используется операция += и соответственно для открепления - операция -=: событие += метод_обработчика_события. Опять же обращаю внимание, что метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип. В итоге мы получим следующий консольный вывод:
```
Сумма 100 снята со счета
На счет поступило 150
```
Кроме использованного выше способа прикрепления обработчиков есть и другой с использованием делегата. Но оба способа будут равноценны:
```
account.Added += Show_Message;
account.Added += new Account.AccountStateHandler(Show_Message);
```
Класс данных события AccountEventArgs
Нередко при возникновении события обработчику события требуется передать некоторую информацию о событии. Например, добавим и в нашу программу новый класс AccountEventArgs со следующим кодом:
```
class AccountEventArgs
{
    // Сообщение
    public string Message{get;}
    // Сумма, на которую изменился счет
    public int Sum {get;}

    public AccountEventArgs(string mes, int sum)
    {
        Message = mes;
        Sum = sum;
    }
}
```
Данный класс имеет два свойства: Message - для хранения выводимого сообщения и Sum - для хранения суммы, на которую изменился счет.
Теперь применим класс AccoutEventArgs, изменив класс Account следующим образом:
```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(object sender, AccountEventArgs e);
    // Событие, возникающее при выводе денег
    public event AccountStateHandler Withdrawn;
    // Событие, возникающее при добавлении на счет
    public event AccountStateHandler Added;

    int _sum; // Переменная для хранения суммы

    public Account(int sum)
    {
        _sum = sum;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added(this, new AccountEventArgs($"На счет поступило {sum}", sum));
    }
    public void Withdraw(int sum)
    {
        if (_sum >= sum)
```

```
        {
            _sum -= sum;
            if (Withdrawn != null)
                Withdrawn(this, new AccountEventArgs($"Сумма {sum} снята со счета", sum));
        }
        else
        {
            if (Withdrawn != null)
                Withdrawn(this, new AccountEventArgs("Недостаточно денег на счете", sum));
        }
    }
}
```

По сравнению с предыдущей версией класса Account здесь изменилось только количество параметров у делегата и соответственно количество параметров при вызове события. Теперь они также принимают объект AccountEventArgs, который хранит информацию о событии, получаемую через конструктор.

Теперь изменим основную программу:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.Withdrawn += Show_Message;

        account.Withdraw(100);
        // Удаляем обработчик события
        account.Withdrawn -= Show_Message;

        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
    }
    private static void Show_Message(object sender, AccountEventArgs e)
    {
        Console.WriteLine($"Сумма транзакции: {e.Sum}");
        Console.WriteLine(e.Message);
    }
}
```

По сравнению с предыдущим вариантом здесь мы только изменяем количество параметров и сущность их использования в обработчике Show_Message.

Анонимные методы

С делегатами тесно связаны Анонимные методы. Анонимные методы используются для создания экземпляров делегатов.

Определение анонимных методов начинается с ключевого слова delegate, после которого идет в скобках список параметров и тело метода в фигурных скобках:

```
delegate(параметры)
{
    // инструкции
}
```

Например:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        MessageHandler handler = delegate(string mes)
        {
            Console.WriteLine(mes);
        };
        handler("hello world!");
```

```
        Console.Read();
    }
}
```
Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

И важно отметить, что в отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        ShowMessage("hello!", delegate(string mes)
        {
            Console.WriteLine(mes);
        });

        Console.Read();
    }
    static void ShowMessage(string mes, MessageHandler handler)
    {
        handler(mes);
    }
}
```
Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        MessageHandler handler = delegate
        {
            Console.WriteLine("анонимный метод");
        };
        handler("hello world!");    // анонимный метод

        Console.Read();
    }
}
```
То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.

При этом параметры анонимного метода не могут быть опущены, если один или несколько параметров определены с модификатором out.

Также, как и обычные методы, анонимные могут возвращать результат:

```
delegate int Operation(int x, int y);
static void Main(string[] args)
{
    Operation operation = delegate (int x, int y)
    {
        return x + y;
    };
    int d = operation(4, 5);
    Console.WriteLine(d);       // 9
```

```
    Console.Read();
}
```
При этом анонимный метод имеет доступ ко всем переменным, определенным во внешнем коде:
```
delegate int Operation(int x, int y);
static void Main(string[] args)
{
    int z = 8;
    Operation operation = delegate (int x, int y)
    {
        return x + y + z;
    };
    int d = operation(4, 5);
    Console.WriteLine(d);       // 17
    Console.Read();
}
```
В каких ситуациях используются анонимные методы? Когда нам надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. Иногда такие методы нужны для обработки одного события и больше ценности не представляют и нигде не применяются. Например, применим анонимные методы для обработки событий:
```
delegate void AccountStateHandler(object sender, AccountEventArgs e);
class AccountEventArgs
{
    public string Message { get;}
    public int Sum { get;}
    public AccountEventArgs(string message, int sum)
    {
        Message = message; Sum = sum;
    }
}
class Account
{
    int _sum;
    public event AccountStateHandler Added;
    public event AccountStateHandler Withdrawn;
    public Account(int sum)
    {
        _sum = sum;
    }
    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null) Added(this,
            new AccountEventArgs($"На счет пришло {sum}", sum));
    }
    public void Withdraw(int sum)
    {
        if(_sum >=sum)
        {
            _sum -= sum;
            if (Withdrawn != null)
                Withdrawn(this, new AccountEventArgs($"Со счета снято {sum}", sum));
        }
        else
        {
            if (Withdrawn != null)
                Withdrawn(this, new AccountEventArgs("На счете недостаточно средств", 0));
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200);
```

```
        // Добавляем обработчики события
        account.Added += delegate (object sender, AccountEventArgs e)
        {
            Console.WriteLine($"Сумма транзакции: {e.Sum}");
            Console.WriteLine(e.Message);
        };
        account.Put(230);

        Console.Read();
    }
}
```

В данном случае, допустим, мы не собираемся использовать код анонимного метода в каких-то других ситуациях, кроме обработки события. Поэтому нет смысла оформлять его в полноценный метод.

Лямбды

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры: (список_параметров) => выражение. Например:

```
class Program
{
    delegate int Operation(int x, int y);
    static void Main(string[] args)
    {
        Operation operation = (x, y) => x + y;
        Console.WriteLine(operation(10, 20));       // 30
        Console.WriteLine(operation(40, 20));       // 60
        Console.Read();
    }
}
```

Здесь код (x, y) => x + y; представляет лямбда-выражение, где x и y - это параметры, а x + y - выражение. При этом нам не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. То есть в данном случае использованное лямбда-выражение соответствует делегату Operation как по типу возвращаемого значения, так и по типу и количеству параметров.

Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить:

```
class Program
{
    delegate int Square(int x); // объявляем делегат, принимающий int и возвращающий int
    static void Main(string[] args)
    {
        Square square = i => i * i; // объекту делегата присваивается лямбда-выражение

        int z = square(6); // используем делегат
        Console.WriteLine(z); // выводит число 36
        Console.Read();
    }
}
```

Бывает, что параметров не требуется. В этом случае вместо параметра в лямбда-выражении используются пустые скобки. Также бывает, что лямбда-выражение не возвращает никакого значения:

```
class Program
{
    delegate void Hello(); // делегат без параметров
```

```
    static void Main(string[] args)
    {
        Hello hello1 = () => Console.WriteLine("Hello");
        Hello hello2 = () => Console.WriteLine("Welcome");
        hello1();       // Hello
        hello2();       // Welcome
        Console.Read();
    }
}
```

В данном случае лямда-выражение ничего не возвращает, так как после лямбда-оператора идет действие, которое ничего не возвращает.

Как видно, из примеров выше, нам необязательно указывать тип параметров у лямбда-выражения. Однако, нам обязательно нужно указывать тип, если делегат, которому должно соответствовать лямбда-выражение, имеет параметры с модификаторами ref и out:

```
class Program
{
    delegate void ChangeHandler(ref int x);
    static void Main(string[] args)
    {
        int x = 9;
        ChangeHandler ch = (ref int n) => n = n * 2;
        ch(ref x);
        Console.WriteLine(x);   // 18
        Console.Read();
    }
}
```

Лямбда-выражения также могут выполнять другие методы:

```
class Program
{
    delegate void Hello(); // делегат без параметров
    static void Main(string[] args)
    {
        Hello message = () => Show_Message();
        message();
    }
    private static void Show_Message()
    {
        Console.WriteLine("Привет мир!");
    }
}
```

Лямбда-выражения в обработке событий

Одним из частых примеров использования лямбда-выражений является обработка событий. Возьмем класс Account из прошлой темы:

```
delegate void AccountStateHandler(object sender, AccountEventArgs e);

    class AccountEventArgs
    {
        public string Message { get;}
        public int Sum { get;}
        public AccountEventArgs(string message, int sum)
        {
            Message = message; Sum = sum;
        }
    }
    class Account
    {
        int _sum;
        public event AccountStateHandler Added;
        public event AccountStateHandler Withdrawn;
        public Account(int sum)
        {
            _sum = sum;
        }
        public void Put(int sum)
        {
            _sum += sum;
```

```
            if (Added != null) Added(this,
                new AccountEventArgs($"На счет пришло {sum}", sum));
        }
        public void Withdraw(int sum)
        {
            if(_sum >=sum)
            {
                _sum -= sum;
                if (Withdrawn != null)
                    Withdrawn(this, new AccountEventArgs($"Со счета снято {sum}", sum));
            }
            else
            {
                if (Withdrawn != null)
                    Withdrawn(this,
                        new AccountEventArgs("На счете недостаточно средств", 0));
            }
        }
    }
```

И перепишем прикрепление обработчика события с помощью лямбды-выражения:

```
static void Main(string[] args)
{
    Account account = new Account(100);
    account.Added += (sender, e) =>
    {
        Console.WriteLine($"Сумма транзакции: {e.Sum}");
        Console.WriteLine(e.Message);
    };
    account.Put(200);
    account.Put(109);
}
```

Поскольку здесь используется несколько параметров, то они заключаются в скобки. И так как в теле лямбда-выражения применяется несколько выражений, то они заключаются в блок из фигурных скобок.

Лямбда-выражения как аргументы методов

Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно:

```
class Program
{
    delegate bool IsEqual(int x);

    static void Main(string[] args)
    {
        int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // найдем сумму чисел больше 5
        int result1 = Sum(integers, x => x > 5);
        Console.WriteLine(result1); // 30

        // найдем сумму четных чисел
        int result2 = Sum(integers, x => x % 2 == 0);
        Console.WriteLine(result2);  //20

        Console.Read();
    }

    private static int Sum (int[] numbers, IsEqual func)
    {
        int result = 0;
        foreach(int i in numbers)
        {
            if (func(i))
                result += i;
        }
        return result;
    }
```

```
}
```
Метод Sum принимает в качестве параметра массив чисел и делегат IsEqual и возвращает сумму чисел массива в виде объекта int. В цикле проходим по всем числам и складываем их. Причем складываем только те числа, для которых делегат IsEqual func возвращает true. То есть делегат IsEqual здесь фактически задает условие, которому должны соответствовать значения массива. Но на момент написания метода Sum нам неизвестно, что это за условие.

При вызове метода Sum ему передается массив и лямбда-выражение:

```
int result1 = Sum(integers, x => x > 5);
```

То есть параметр x здесь будет представлять число, которое передается в делегат:

```
if (func(i))
```

А выражение x > 5 представляет условие, которому должно соответствовать число. Если число соответствует этому условию, то лямбда-выражение возвращает true, а переданное число складывается с другими числами.

Подобным образом работает второй вызов метода Sum, только здесь уже идет проверка числа на четность, то есть если остаток от деления на 2 равен нулю:

```
int result2 = Sum(integers, x => x % 2 == 0);
```

Ковариантность и контравариантность делегатов

Делегаты могут быть ковариантными и контравариантными. Ковариантность делегата предполагает, что возвращаемым типом может быть более производный тип. Контрвариантность делегата предполагает, что типом параметра может быть более универсальный тип.

Ковариантность

Ковариантность позволяет возвращать из метода объект, тип которого является производным от типа, возвращаемого делегатом.

Допустим, имеется следующая структура классов:

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

В этом случае ковариантность делегата может выглядеть следующим образом:

```
delegate Person PersonFactory(string name);
static void Main(string[] args)
{
    PersonFactory personDel;
    personDel = BuildClient; // ковариантность
    Person p = personDel("Tom");
    Console.WriteLine(p.Name);
    Console.Read();
}
private static Client BuildClient(string name)
{
    return new Client {Name = name};
}
```

То есть здесь делегат возвращает объект Person. Однако благодаря ковариантности данный делегат может указывать на метод, который возвращает объект производного типа, например, Client.

Контрвариантность

Контрвариантность предполагает возможность передавать в метод объект, тип которого является более универсальным по отношению к типу параметра делегата. Например, возьмем выше определенные классы Person и Client и используем их в следующем примере:

```
delegate void ClientInfo(Client client);
static void Main(string[] args)
{
    ClientInfo clientInfo = GetPersonInfo; // контравариантность
    Client client = new Client{Name = "Alice"};
    clientInfo(client);
    Console.Read();
}
private static void GetPersonInfo(Person p)
```

```
{
    Console.WriteLine(p.Name);
}
```
Несмотря на то, что делегат в качестве параметра принимает объект Client, ему можно присвоить метод, принимающий в качестве параметра объект базового типа Person. Может показаться на первый взгляд, что здесь есть некоторое противоречие, то есть использование более универсального тип вместо более производного. Однако в реальности в делегат при его вызове мы все равно можем передать только объекты типа Client, а любой объект типа Client является объектом типа Person, который используется в методе.

Ковариантность и контравариантность в обобщенных делегатах

Обобщенные делегаты также могут быть ковариантными и контравариантными, что дает нам больше гибкости в их использовании.

Например, возьмем следующую иерархию классов:

```
class Person
{
    public string Name { get; set; }
    public virtual void Display() => Console.WriteLine($"Person {Name}");
}
class Client : Person
{
    public override void Display() => Console.WriteLine($"Client {Name}");
}
```
Теперь объявим и используем ковариантный обобщенный делегат:
```
class Program
{
    delegate T Builder<out T>(string name);
    static void Main(string[] args)
    {
        Builder<Client> clientBuilder = GetClient;
        Builder<Person> personBuilder1 = clientBuilder;     // ковариантность
        Builder<Person> personBuilder2 = GetClient;         // ковариантность

        Person p = personBuilder1("Tom"); // вызов делегата
        p.Display(); // Client: Tom

        Console.Read();
    }
    private static Person GetPerson(string name)
    {
        return new Person {Name = name};
    }
    private static Client GetClient(string name)
    {
        return new Client {Name = name};
    }
}
```
Благодаря использованию out мы можем присвоить делегату типа Builder<Person> делегат типа Builder<Client> или ссылку на метод, который возвращает значение Client.

Рассмотрим контравариантный обобщенный делегат:
```
class Program
{
    delegate void GetInfo<in T>(T item);
    static void Main(string[] args)
    {
        GetInfo<Person> personInfo = PersonInfo;
        GetInfo<Client> clientInfo = personInfo;      // контравариантность

        Client client = new Client { Name = "Tom" };
        clientInfo(client); // Client: Tom

        Console.Read();
    }
    private static void PersonInfo(Person p) => p.Display();
    private static void ClientInfo(Client cl) => cl.Display();
}
```

Использование ключевого слова in позволяет присвоить делегат с более универсальным типом (GetInfo<Person>) делегату с производным типом (GetInfo<Client>).
Как и в случае с обобщенными интерфейсами параметр ковариантного типа применяется только к типу значения, которые возвращается делегатом. А параметр контравариантного типа применяется только к входным аргументам делегата.

Делегаты Action, Predicate и Func

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми, с которыми часто приходится сталкиваться, являются Action, Predicate и Func.

Action

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

```
public delegate void Action<T>(T obj)
```

Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,....in T16>. Таким образом можно передать до 16 значений в метод.
Как правило, этот делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия. Например:

```
static void Main(string[] args)
{
    Action<int, int> op;
    op = Add;
    Operation(10, 6, op);
    op = Substract;
    Operation(10, 6, op);

    Console.Read();
}

static void Operation(int x1, int x2, Action<int, int> op)
{
    if (x1 > x2)
        op(x1, x2);
}

static void Add(int x1, int x2)
{
    Console.WriteLine("Сумма чисел: " + (x1 + x2));
}

static void Substract(int x1, int x2)
{
    Console.WriteLine("Разность чисел: " + (x1 - x2));
}
```

Predicate

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

```
Predicate<int> isPositive = delegate (int x) { return x > 0; };

Console.WriteLine(isPositive(20));
Console.WriteLine(isPositive(-20));
```

В данном случае возвращается true или false в зависимости от того, больше нуля число или нет.

Func

Еще одним распространенным делегатом является Func. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров.

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

```
TResult Func<in T, out TResult>(T arg)
TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)
TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)
TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
//.....................................
```

Данный делегат также часто используется в качестве параметра в методах:

```
static void Main(string[] args)
{
    Func<int, int> retFunc = Factorial;
    int n1 = GetInt(6, retFunc);
    Console.WriteLine(n1);  // 720

    int n2 = GetInt(6, x=> x *x);
    Console.WriteLine(n2); // 36

    Console.Read();
}

static int GetInt(int x1, Func<int, int> retF)
{
    int result = 0;
    if (x1 > 0)
        result = retF(x1);
    return result;
}
static int Factorial(int x)
{
    int result = 1;
    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    return result;
}
```

Метод GetInt() в качестве параметра принимает делегат Func<int, int>, то есть ссылку на метод, который принимает число int и возвращает также значение int.

При первом вызове метода GetInt() ему передается ссылка на метод вычисления факториала. Во втором случае передается лямбда-выражение x=> x*x, то есть опять же выражение, которое принимает параметр int x и возвращает результат x * x.

Список использованных источников

1. Агуров, Павел C#. Сборник рецептов / Павел Агуров. - М.: "БХВ-Петербург", 2012. - 432 с.
2. Албахари, Джозеф C# 3.0. Справочник / Джозеф Албахари , Бен Албахари. - М.: БХВ-Петербург, 2012. - 944 с.
3. Албахари, Джозеф C# 3.0. Справочник / Джозеф Албахари , Бен Албахари. - М.: БХВ-Петербург, 2013. - 944 с.
4. Альфред, В. Ахо Компиляторы. Принципы, технологии и инструментарий / Альфред В. Ахо и др. - М.: Вильямс, 2015. - 266 с.
5. Бишоп, Дж. C# в кратком изложении / Дж. Бишоп, Н. Хорспул. - М.: Бином. Лаборатория знаний, 2013. - 472 с.
6. Вагнер, Билл C# Эффективное программирование / Билл Вагнер. - М.: ЛОРИ, 2013. - 320 с.
7. Зиборов, В.В. Visual C# 2012 на примерах / В.В. Зиборов. - М.: БХВ-Петербург, 2013. - 480 с.
8. Зиборов, Виктор Visual C# 2010 на примерах / Виктор Зиборов. - М.: "БХВ-Петербург", 2011. - 432 с.
9. Ишкова, Э. А. Самоучитель C#. Начала программирования / Э.А. Ишкова. - М.: Наука и техника, 2013. - 496 с.
10. Касаткин, А. И. Профессиональное программирование на языке си. Управление ресурсами / А.И. Касаткин. - М.: Высшая школа, 2012. - 432 с.
11. Лотка, Рокфорд C# и CSLA .NET Framework. Разработка бизнес-объектов / Рокфорд Лотка. - М.: Вильямс, 2010. - 816 с.
12. Мак-Дональд, Мэтью Silverlight 5 с примерами на C# для профессионалов / Мэтью Мак-Дональд. - М.: Вильямс, 2013. - 848 с.
13. Марченко, А. Л. Основы программирования на C# 2.0 / А.Л. Марченко. - М.: Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2011. - 552 с.
14. Подбельский, В. В. Язык C#. Базовый курс / В.В. Подбельский. - М.: Финансы и статистика, Инфра-М, 2011. - 384 с.
15. Прайс, Джейсон Visual C# 2.0. Полное руководство / Джейсон Прайс , Майк Гандэрлой. - М.: Век +, Корона-Век, Энтроп, 2010. - 736 с.
16. Рихтер, Джеффри CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C# / Джеффри Рихтер. - М.: Питер, 2013. - 928 с.
17. Смоленцев, Н. К. MATLAB. Программирование на Visual C#, Borland JBuilder, VBA (+ CD-ROM) / Н.К. Смоленцев. - М.: ДМК Пресс, 2011. - 456 с.
18. Троелсен, Эндрю Язык программирования C# 5.0 и платформа .NET 4.5 / Эндрю Троелсен. - М.: Вильямс, 2015. - 486 с.
19. Троелсен, Эндрю Язык программирования C# 2008 и платформа .NET 3.5 / Эндрю Троелсен. - М.: Вильямс, 2010. - 370 с.
20. Фримен, Адам ASP.NET MVC 3 Framework с примерами на C# для профессионалов / Адам Фримен , Стивен Сандерсон. - М.: Вильямс, 2011. - 672 с.

Лысанов Д.М., Хамидуллин М.Р.

Объектно – ориентированное программироование

Учебно-методическое пособие

Подписано в печать 22.04. 2019.
Формат 60x84/16. Печать ризографическая.
Бумага офсетная. Гарнитура «Times New Roman».
Усл.п.л. 9,69 Уч.-изд. л. 9,3
Тираж 100 экз. Заказ № 1231

Отпечатано в Издательско-полиграфическом центре
Набережночелнинского института
Казанского (Приволжского) федерального университета

423810, г. Набережные Челны, Новый город, пр.Мира, 68/19
тел./факс (8552) 39-65-99 e-mail: ic-nchi-kpfu@mail.ru