Автор курса: доцент каф. ИСТ Кислицын Дмитрий Игоревич

ННГАСУ - 2016

Содержание 1

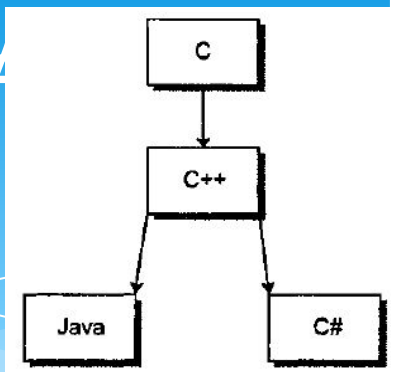
Содержание 2

Содержание 3

Что почитать

* Герберт Шилдт - C# 4.0. Полное руководство, 2011
* MSDN. Visual C#
* Казанский А. А. Объектно-ориентированное программирование на языке Microsoft Visual C# в среде разработки Microsoft Visual Studio 2008 и .NET Framework 4.3 : Учебное пособие и практикум, Москва : Московский государственный строительный университет, ЭБС АСВ, 2013
* Культин Н. Б. Microsoft Visual C# в задачах и примерах. – СПб.: БХВ-Петербург, 2009. – 320 с.

История создания C#



C# - прямой потомок двух самых успешных языков программирования C (был наиболее популярен в 1980-х) и C++ (был наиболее популярен в 1990-х) и тесно связан с Java.

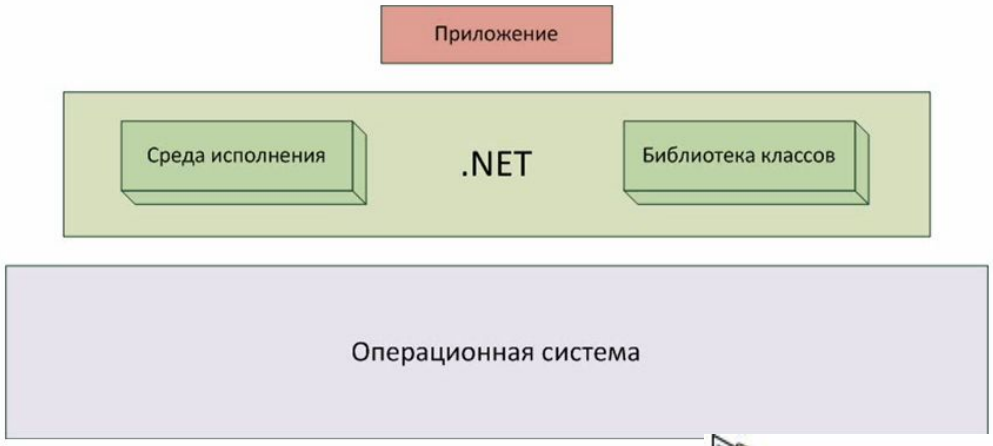
C и C++ являются платформо-зависимыми языками, что мешало переносимости программ. Java, который начал разрабатываться в 1991г., стал платформо-независимым языком

Однако Java:

- не имеет *межъязыковой возможности взаимодействия* (cross-language interoperability) программных и аппаратных изделий разных поставщиков, или *многоязыкового программирования* (mixed-language programming)
- не достигнута полная интеграция с платформой Windows

Microsoft стала разрабатывать C# в конце 1990-х и стал частью общей .NET-стратегии Microsoft. Впервые он увидел свет в качестве альфа-версии в середине 2000 года

Связь C# с оболочкой .NET Framework

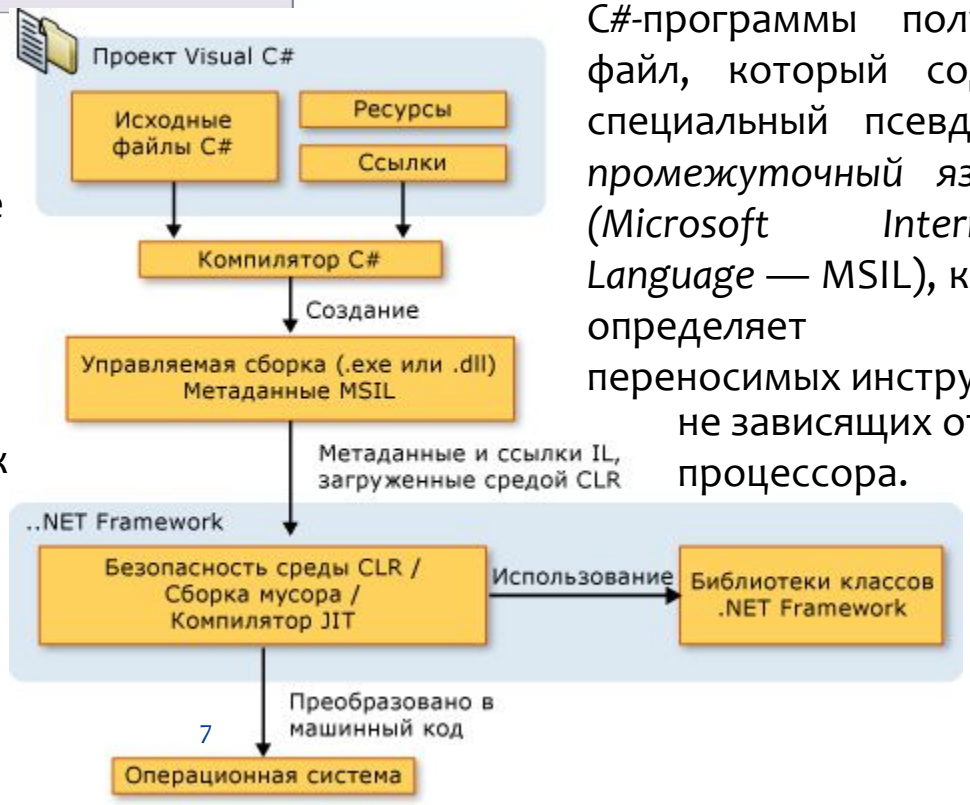


Оболочка .NET Framework определяет среду для разработки и выполнения сильно распределенных приложений, основанных на использовании компонентных объектов.

Common Language Runtime (CLR) - составная часть .NET Framework, управляющая выполнением .NET-кода, поддерживает многоязыковое программирование и обеспечивает переносимость и безопасность.

Библиотека классов .NET-оболочки, предоставляет программам доступ к среде выполнения.

Класс — это объектно-ориентированная конструкция, с помощью которой организуются программы.



В результате компиляции C#-программы получается файл, который содержит специальный псевдокод - *промежуточный язык MS (Microsoft Intermediate Language* — MSIL), который определяет набор переносимых инструкций, не зависящих от типа процессора.

Объектно-ориентированное программирование

C# неотделимо от ООП, позволяющего усовершенствовать процесс программирования

Набор машинных инструкций в двоичном коде на передней панели компьютера (несколько сотен инструкций)

Язык ассемблер, использующий мнемоники

который позволил программисту писать гораздо большие и более сложные программы, используя символическое представление машинных инструкций. Языки высокого уровня (например, FORTRAN и COBOL)

Структурное программирование

Объектно-ориентированное программирование, вобравшее в себя лучшие идеи структурного программирования и объединившее их с новыми концепциями.

Программа

Действия в программе

Данные, подвергающиеся воздействию

Методы структурного программирования (код, воздействующий на данные)

Методы ООП (данные должны управлять доступом к коду)

Характерные черты ООП

Инкапсуляция

Механизм программирования, который связывает код (действия) и данные, которыми он манипулирует, и при этом предохраняет их от вмешательства извне и неправильного использования. В объектно-ориентированном языке код и данные можно связать таким образом, что будет создан автономный *черный ящик - объект.*

Основной единицей инкапсуляции в C# является *класс* (набор шаблонных элементов). Класс определяет форму объекта (задает данные и код). В C# класс используется для создания объектов. Объекты — это экземпляры класса.

Код и данные, которые составляют класс, называются *членами* класса. Данные, определенные в классе, называются *переменными экземпляра* (instance variable), а код, который оперирует этими данными, — *методами-членами* (member method), или просто *методами.*

Характерные черты ООП

Полиморфизм

Качество, которое позволяет одному интерфейсу получать доступ к целому классу действий.

Пример: руль автомобиля. Руль (интерфейс) остается рулем независимо от того, какой тип рулевого механизма используется в автомобиле. Поворот руля влево заставит автомобиль поехать влево независимо от типа используемого в нем рулевого управления.

Концепцию полиморфизма часто выражают такой фразой: "один интерфейс — много методов". Это означает, что для выполнения группы подобных действий можно разработать общий интерфейс. Полиморфизм позволяет понизить степень сложности программы, предоставляя программисту возможность использовать один и тот же интерфейс для задания *общего класса действий.* Конкретное (нужное в том или ином случае) действие (метод) выбирается компилятором. Программисту нет необходимости делать это вручную. Его задача — правильно использовать общий интерфейс.

Характерные черты ООП

Наследование

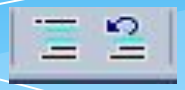
Процесс, благодаря которому один объект может приобретать свойства другого. Благодаря наследованию поддерживается концепция иерархической классификации. В виде управляемой иерархической (нисходящей) классификации организуется большинство областей знаний. Например, яблоки *Антоновка* являются частью классификации *яблоки,* которая в свою очередь является частью класса *фрукты,* а тот — частью еще большего класса *пища.* Таким образом, класс *пища* обладает определенными качествами (съедобность, питательность и пр.), которые применимы и к подклассу *фрукты.* Помимо этих качеств, класс *фрукты* имеет специфические характеристики (сочность, сладость и пр.), которые отличают их от других пищевых продуктов.

Если не использовать иерархическое представление признаков, для каждого объекта пришлось бы в явной форме определить все присущие ему характеристики. Но благодаря наследованию объекту нужно доопределить только те качества, которые делают его уникальным внутри его класса, поскольку он (объект) наследует общие атрибуты своего родителя. Следовательно, именно механизм наследования позволяет одному объекту представлять конкретный экземпляр более общего класса.

Структура программы C#

```
1.  // A Hello World! program in C#
2.  using System;
3.  namespace Hello World
4.  {
5.  class Hello
6.     {
7.  static void Main()
8.        {
9.  System.Console.WriteLine("Hello World!");
10. Console.WriteLine("Press any key to exit.");
11. Console.ReadKey();
12.       }
13.    }
14. }
```

Комментарии
// A Hello World! program in C#
/* A "Hello World!" program in C#.
This program displays the string "Hello World!" on the screen. */



Для использования в программе классов из других пространств имен необходимо указать их с директивой *using*.

Пространство имен позволяет сгруппировать вместе классы и структуры, что ограничивает их область действия и позволяет избежать конфликта имен с другими классами и структурами.

Пример создания собственного пространства имён

```
1.  namespace StatisticalData {
2.   class FileHandling    {
3.     public void Load() {}  // code to load statistical data
4.    }
5.  }
6.
7.  namespace Images  {
8.   class FileHandling    {
9.     public void Load() {}  // code to load an image file
10.   }
11. }
12.
13. class Program {
14.  static void Main()    {
15.    StatisticalData.FileHandling data = new StatisticalData.FileHandling();
16.    data.Load();
17.    Images.FileHandling image = new Images.FileHandling();
18.     image.Load();
19.  }
20. }
```

Метод Main ()

```
static void Main()
{
// возвращает значение void
}
```

```
static int Main(string[] args)
{
 // принимает массив строковых аргументов
 return 0;
}
```

```
static int Main()
{
// возвращает значение типа int
return 0;
}
```

```
class TestClass
{
  static void Main(string[] args) {
    // Display the number of command line
arguments:
    System.Console.WriteLine(args.Length);
    for(int i=0; i < args.Length; i++)
      Console.WriteLine(args[i]) ;
  }
}
```

Консольный ввод и вывод

Console.WriteLine() - записывает указанные данные с текущим признаком конца строки в стандартный выходной поток

Console.Write() - записывает текстовое представление заданного значения или значений в стандартный выходной поток

Console.ReadLine() - считывает следующую строку символов из стандартного входного потока.

Console.Read() - считывает следующий символ из входного потока или значение минус единица (-1), если доступных для чтения символов не осталось.

```
string  str = Console.ReadLine();
```

Консольный ввод и вывод

Некоторые варианты вывода данных:

Console.WriteLine(“Вывод” + “строки с использованием” + “символа” + “+.”);

На экране: Вывод строки с использованием символа +.

Console.WriteLine(“Текст {номер_ аргумента, минимальная_ширина: формат} Текст {номер_ аргумента, минимальная_ширина: формат} Текст”, arg1, arg2);

int year = 1066;

string battle = "Battle of Hastings";

Console.WriteLine("The {0} took place in {1}.", battle, year);

Результат будет выглядеть следующим образом: The Battle of Hastings took place in 1066.

Console.WriteLine("В феврале {0,10} или {1,5} дней.", 28, 29); // минимальная ширина первого аргумента 10 символов, второго – 5 символов

На экране: В феврале ____28 или __29 дней.

Console.WriteLine ("При делении 10/3 получаем: {0:#.##}", 10.0/3.0);

На экране: При делении 10/3 получаем: 3.33

Console.WriteLine("{0:###,###.##}", 123456.56);

На экране: 123,456.56

Литералы

Литерал - фиксированное значение, представленное в понятной форме.

Строковой литерал - строка символов, заключённая в кавычки. В строковом литерале могут использоваться управляющие последовательности символов (Escape-знаки).

Esc	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Подача страницы (для перехода к началу следующей страницы)
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\’	Одинарная кавычка (апостроф)
\”	Двойная кавычка
\\	Обратная косая черта

Пример:

```
Console.WriteLine("Первая строка\nВторая строка\nТретья строка");
Console.WriteLine ("Один\tДва\tТри") ;
Console.WriteLine (“Четыре\tПять\tШесть") ;
Console.WriteLine("\"Зачем?\", спросил он.");
```

На экране:

```
Первая строка
Вторая строка
Третья строка
Один    Два     Три
Четыре  Пять    Шесть
"Зачем?", спросил он.
```

Литералы

Буквальный строковой литерал (@)

Буквальный строковой литерал (@) выводит строку как есть. Пример:

```
Console.WriteLine(@"Воспользуемся табуляцией:
1		2	3	4
5		6	7	8
“);
Console.WriteLine(@"Отзыв программиста: "Мне нравится C #"");
```

На экране:

```
Воспользуемся табуляцией:
1	2	3	4
5	6	7	8
Отзыв программиста: "Мне нравится C#."
```

Литералы

Численный литерал

Правила определения типа литерала:

* целочисленным литералам присваивается наименьший целочисленный тип, который сможет его хранить, начиная с типа int. Таким образом, целочисленный литерал, в зависимости от конкретного значения, может иметь тип int, uint, long или ulong;
* все литералы с плавающей точкой имеют тип double.

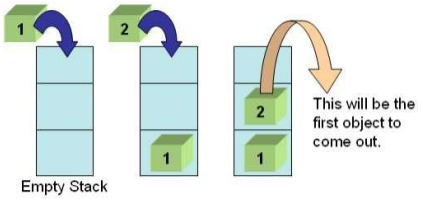
Для явного определения типа литерала к нему добавляется соответствующий суффикс.

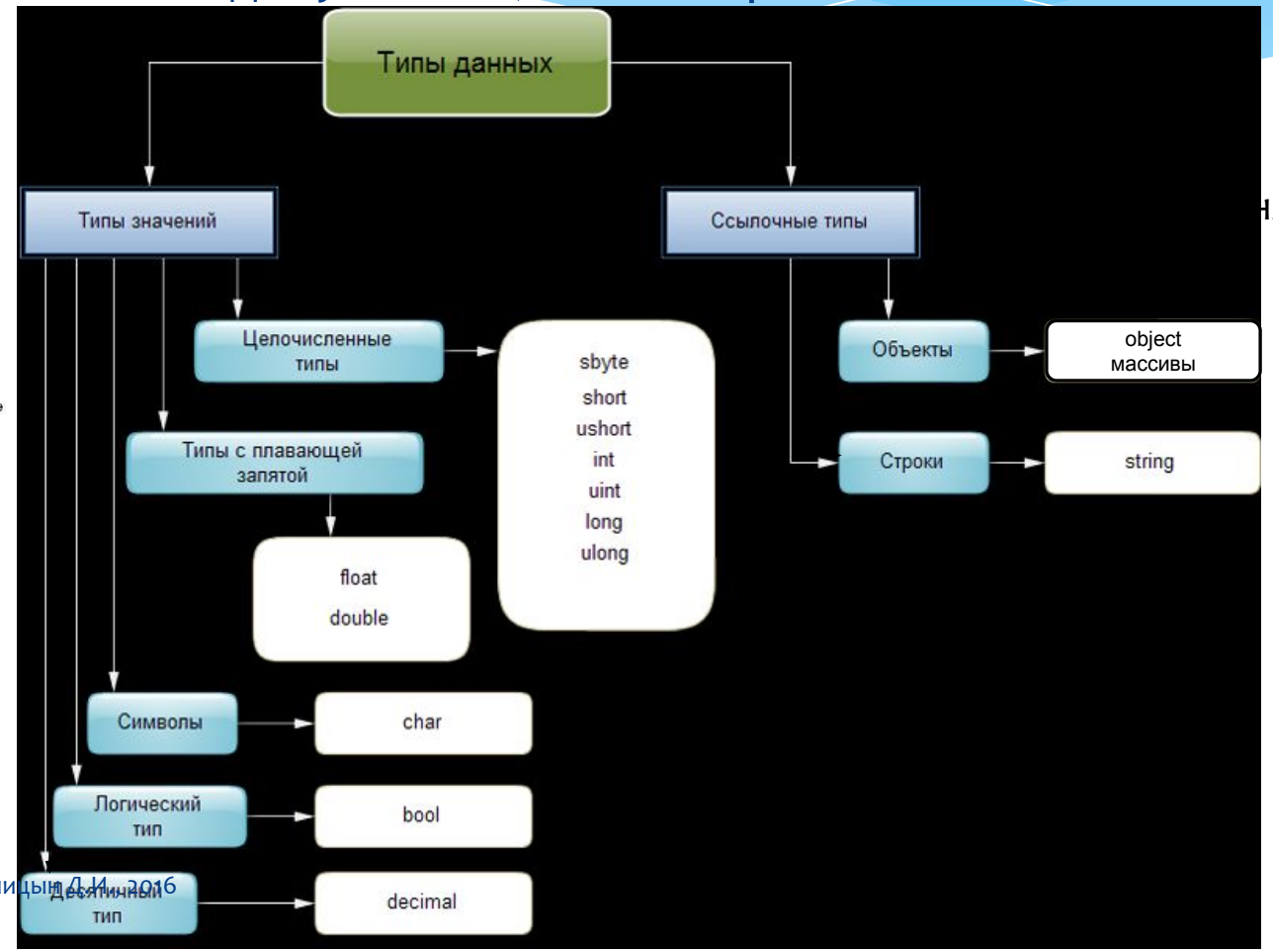
Желаемый тип литерала	Добавляемый суффикс	Пример
long	L или l	100L
uint	U или u	100U
ulong	UL или ul	100UL
float	F или f	100F
decimal	M или m	100M
шестнадцатиричный	0x	0xFF; //255 в десятичной системе

Типы данных

C# — строго типизированный язык. Это значит, что все операции проверяются компилятором на соответствие типов. Некорректные операции не компилируются. В C# не допускается, чтобы переменная не имела типа.

Значения хранятся в *стеке*





Типы данных. Типы значений

Тип	Описание	Размер, бит	Диапазон	Примечание
Целочисленные				
byte	8-разрядный целочисленный без знака	8	0 - 255	
sbyte	8-разрядный целочисленный со знаком	8	-128 - 127	
short	Тип для представления короткого целого числа	16	-32 768 - 32 767	
ushort	Тип для представления короткого целого числа без знака	16	0 - 65 535	
int	Целочисленный	32	-2 147 483 648 - 2147 483 647	
uint	Целочисленный без знака	32	0 - 4 294 967 295	
long	Тип для представления длинного целого числа	64	-9 223 372 036 854 775 808 - 9 223 372 036 854 775 807	
ulong	Тип для представления длинного целого числа без знака	64	0 - 18 446 744 073 709 551615	
С плавающей точкой				
float	С плавающей точкой	32	1,5E-45 - 3,4E+38	
double	С плавающей точкой двойной точности	64	5E-324 - 1,7E+308	
Денежный				
decimal	Числовой тип для финансовых вычислений	128	1E-28 - 7,9E+28	Для констант требуется указать m (decimal a=5m)
Символьный				
char	Символьный (Unicode)	16	0 - 65 535	char ch=’q’
Логический				
bool	Логический	1	true и false	

Типы данных

Переменная - числовое или строковое значение или объект класса. Значение, хранящееся в переменной, может измениться, однако имя остается прежним. Переменные объявляются с определенным типом данных.

```
int x = 1;
x = 2;
int a, b=2, c;
a=1;
c=3;
if(a!=b) {
   int d=5;
   b=22;
   int c=33; // ОШИБКА!
}
```

Типы данных

Преобразование типов

Автоматическое преобразование типов

Верно

```
public static void Main() {
   long L;
   double D;
   L = 100123285L;
   D = L;
   Console.WriteLine("L и D: " + L + " " + D);
}
```

Неверно

```
public static void Main() {
   long L;
   double D;
   D = 100123285.0;
   L = D; // Неверно!!!
   Console.WriteLine("L и D: " + L + " " + D);
}
```

Приведение несовместимых типов

(тип_приемника) выражение

```
double x, y;
// . . .
(int) (x / y) ;
```

Внимание! Может произойти потеря данных

Типы данных

Константы

Константа является другим типом поля. Она хранит значение, присваиваемое по завершении компиляции программы, и никогда после этого не изменяется. Константы объявляются помощью ключевого слова **const**. Их использование способствует повышению удобочитаемости кода.

```
const int speedLimit = 55;
const double pi = 3.14159265358979323846264338327950;
```

Ключевое слово **readonly** отличается от ключевого слова **const**. Поле с модификатором **const** может быть инициализировано только при объявлении поля. Поле с модификатором **readonly** может быть инициализировано при объявлении или в конструкторе. Следовательно, поля с модификатором **readonly** могут иметь различные значения в зависимости от использованного конструктора.

Типы данных

Строки

Строка C# представляет собой группу одного или нескольких знаков, объявленных с помощью ключевого слова **string**, которое является ускоренным методом языка C# для класса **System.String**. В отличие от массивов знаков в C или C++, строки в C# гораздо проще в использовании и менее подвержены ошибкам программирования.

Можно извлекать подстроки и объединять строки, как показано в следующем примере.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";
s1 += s2;
 System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Строковые объекты являются неизменяемыми: после создания их нельзя изменить. Методы, работающие со строками, возвращают новые строковые объекты. Поэтому в целях повышения производительности большие объемы работы по объединению строк или другие операции следует выполнять в классе **StringBuilder**.

Операторы

Арифметические		
Оператор	**Описание**	**Примечание**
+	Сложение	
-	Вычитание	
*	Умножение	
/	Деление	После применения оператора деления (/) к целому числу остаток будет отброшен. Например, 10/3 будет равен 3
%	Деление по модулю	Остаток от деления (10 % 3 = 1)
--	Декремент	Уменьшение на 1 (x=x-1)
++	Инкремент	Увеличение на 1 (x=x+1)

Инкремент (Декремент)	
Префиксная форма	**Постфиксная форма**
x = 10; y = ++x;	x = 10; y = x++;
Результат: x=11, y= 11	Результат: x=11, y= 10

Операторы

Присваивания

переменная = выражение;

```
int x, y, z;
x = y = z = 100; // устанавливаем переменные x, y  и z равными 100
x=x+5; // добавляем к переменной x число 5
x+=5; // добавляем к переменной x число 5
```

Возможны следующие варианты объединения операторов:

+=
-=
*=
/=
%=
&=
!=

Операторы

Логические и операторы отношений		
Оператор	**Описание**	**Примечание**
==	Равно	
!=	Не равно	
>	Больше	
<	Меньше	
>=	Больше или равно	
<=	Меньше или равно	
&	И	
|	ИЛИ	
^	Исключающее ИЛИ	
&&	Сокращенное (условное) И	Второй операнд проверяется только, если есть необходимость. Пример: *int n, d;* *n = 10;* *d = 2;* *if(d != 0 && (n % d) == 0)* *Console.WriteLine(d + " - множитель числа " + n);*
||	Сокращенное (условное) ИЛИ	
!	НЕ	

Операторы

Поразрядные

Поразрядные операторы предназначены для тестирования, установки или сдвига битов (разрядов), из которых состоит целочисленное значение. Поразрядные операторы очень часто используются для решения широкого круга задач программирования системного уровня, например, при опросе информации о состоянии устройства или ее формировании.

Они определены только для целочисленных операндов и не могут быть использованы для операндов типа *bool, float или double*.

Оператор	Описание
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево

```
// Использование поразрядного
      оператора И для "превращения"
      любого числа в четное
public static void Main() {
 ushort num;
 ushort i;
 for(i = 1; i <= 10; i++) {
  num = i ;
  Console.WriteLine("num: " + num);
  num = (ushort) (num & OxFFFE);
  // num & 1111 1111 1111 1110
  Console.WriteLine("num после сброса
       младшего бита: "+ num + "\n");
 }
}
```

Операторы

Поразрядные

Оператор **XOR** обладает интересным свойством, которое позволяет использовать его для кодирования сообщений

R1 = X ^ Y; // шифрование сообщения X ключом Y
R2 = R1 ^ Y; // дешифрование сообщения R1 ключом Y
При этом R2 будет равен X.

Пример использования оператора XOR для шифрования и дешифрирования сообщения.

```
public static void Main() {
   char ch1 = 'H';
   char ch2 = 'i';
   char ch3 = '!';
   int key = 88;
   Console.WriteLine("Исходное сообщение: " + ch1 + ch2 + ch3);
// Шифруем сообщение,
   ch1 = (char) (ch1 ^ key);
   ch2 = (char) (ch2 ^ key);
  ch3 = (char) (ch3 ^ key);
  Console.WriteLine("Зашифрованное сообщение: " + ch1 + ch2 + ch3);
// Дешифрируем сообщение,
  ch1 = (char) (ch1 ^ key);
  ch2 = (char) (ch2 ^ key);
  ch3 = (char) (ch3 ^ key);
  Console.WriteLine("Дешифрованное сообщение: " + ch1 + ch2 + ch3);
}
```

Результат выполнения программы:
Исходное сообщение: Hi!
Зашифрованное сообщение: >1y
Дешифрованное сообщение: Hi!

Операторы

Поразрядные

Операторы сдвига

В C# можно сдвигать значение влево или вправо на заданное число разрядов. Для это в C# определены следующие операторы поразрядного сдвига:

<< сдвиг влево

>> сдвиг вправо

Общий формат записи этих операторов такой:

значение >> число_битов;

Здесь *значение* - это объект операции сдвига, а элемент *число_битов* указывает, на сколько разрядов должно быть сдвинуто *значение.* При сдвиге влево на один разряд все биты, составляющее значение, сдвигаются влево на одну позицию, а в младший разряд записывается нуль. При сдвиге вправо все биты сдвигаются, соответственно, вправо. Если сдвигу вправо подвергается значение без знака, в старший разряд записывается нуль. Если же сдвигу вправо подвергается значение со знаком, значение знакового разряда сохраняется.

При сдвиге как вправо, так и влево крайние биты теряются. Следовательно, при этом выполняется нециклический сдвиг, и содержимое потерянного бита узнать невозможно.

Операторы

Поразрядные

Тернарный оператор ?

Выражение1 ? Выражение2 : Выражение3;

Значение ?-выражения определяется следующим образом. Вычисляется *Выражение1.* Если оно оказывается истинным, вычисляется *Выражение2,* и результат его вычисления становится значением всего ?-выражения. Если результат вычисления элемента *Выражение1* оказывается ложным, значением всего ?-выражения становится результат вычисления элемента *Выражение3.*

Выражение1 должно иметь тип bool. Типы элементов *Выражение2* и *Выражение 3* должны быть одинаковы.

Пример:

```
absval = val < 0 ? -val : val; // Получаем абсолютное значение val
```

Операторы

Приоритет C#-операторов

Наивысший

( ) [ ] . ++(постфиксный) --(постфиксный) checked new sizeof typeof unchecked

! ~ *Операторы_приведения_типа* +(унарный) -(унарный) ++(префиксный) --(префиксный)

***** / %

\+ -

<< >>

< <= > >= is

== !=

&

^

|

&&

||

?:

= op=

Низший

Инструкции управления

1. *Инструкции выбора*
   (if, switch)

2. *Итерационные инструкции*
   (for-, while-, do-while-, foreach- циклы)

3. *Инструкции перехода*
   (break, continue, goto, return, throw)

Инструкции выбора

Инструкция if

```
if (условие) инструкция;
else инструкция;
```

или

```
if (условие)
{
  последовательность инструкций
}
else
{
  последовательность инструкций
}
```

или

```
if (условие)
  инструкция;
else if (условие)
  инструкция;
else if (условие)
  инструкция;
else
  инструкция;
```

Инструкции выбора

Инструкция switch

Позволяет делать выбор одной из множества альтернатив. Значение выражения последовательно сравнивается с константами из заданного списка. При обнаружении совпадения для одного из условий сравнения выполняется последовательность инструкций, связанная с этим условием.

```
switch (выражение) {
  case константа1:
    последовательность инструкций;
    break;
  case константа2:
    последовательность инструкций;
    break;
  case константа3:
    последовательность инструкций;
    break;
  default:
    последовательность инструкций;
    break;
}
```

Элемент *выражение* инструкции switch должен иметь целочисленный тип (например, char, byte, short или int) или тип string). Выражения, имеющие тип с плавающей точкой, не разрешены.

Итерационные инструкции

Цикл for

for (*инициализация; условие; итерация*) *инструкция;*
или
for (*инициализация; условие; итерация*)
{
последовательность инструкций
}

Пример:

```
for (int num = 2; num < 20; num++) {
    isprime = true;
     // Узнаем, делится ли num на i без остатка.
    for(int i=2; i <= num/2; i++) {
      if ((num % i) == 0) {
        // Если num делится на i без остатка, значит num — число не простое
      }
    }
```

Для управления циклом for можно использовать две или больше переменных.

```
// Использование запятых в цикле for.
using System;
class Comma {
  public static void Main() {
    int i, j ;
    for(i=0, j=10; i < j; i++, j--)
      Console.WriteLine("i и j: " + i + " " + j);
  }
}
```

Результаты выполнения этой программы:

```
i и j: 0 10
i и j: 1 9
i и j: 2 8
i и j: 3 7
i и j: 4 6
```

Итерационные инструкции

Цикл for

Отсутствие элементов в определении цикла

```
 int i;
for (i = 0; i < 10; ) { //
  Console.WriteLme ("Проход №" + i) ;
  i++;              // Инкрементируем управляющую переменную цикла
}
```

```
int i ;
i = 0;              // Убираем из цикла раздел инициализации.
for(; i < 10; ) {
  Console .WriteLine ("Проход №" + i) ;
  i++;
}
```

К размещению выражения инициализации за пределами цикла, как правило, прибегают только в том случае, когда начальное значение генерируется сложным процессом, который неудобно поместить в определение цикла.

```
for ( ; ; )         // Специально созданный бесконечный цикл, который будет работать без конца.
```

Для выхода используется break;

Итерационные инструкции

Цикл while

Общая форма цикла while имеет такой вид:

while (*условие*) *инструкция*;

Цикл do-while

Общий формат имеет такой вид:

```
do {
  инструкции;
}
while (условие);
```

Цикл foreach

Цикл foreach предназначен для опроса элементов *коллекции*. Коллекция — это группа объектов. В C# определено несколько типов коллекций, среди которых можно выделить массив. Однако его не следует использовать для добавления или удаления элементов исходной коллекции.

```
int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
    foreach (int element in fibarray)
    {
        System.Console.WriteLine(element);
    }
```

Инструкции перехода

Инструкция break

Break служит для досрочного выхода из любого C#-цикла (*for, while, do-while u foreach*).
Если *break* находится в цикле, расположенном в ветви *switch*, то *break* будет относиться к *switch* и выход из цикла производиться не будет.

Инструкция continue

Помимо средства "досрочного" выхода из цикла, существует средство "досрочного" выхода из текущей его итерации. Этим средством является инструкция *continue*. Она принудительно выполняет переход к следующей итерации, опуская выполнение оставшегося кода в текущей.

Инструкция return

Инструкция *return* обеспечивает возврат из метода. Ее можно использовать для возвращения методом значения.

Инструкция goto

Инструкция безусловного перехода. При ее выполнении управление программой передается инструкции, указанной с помощью метки. Инструкция *goto* требует наличие в программе метки. Метка — это действительный в C# идентификатор, за которым поставлено двоеточие. Метка должна находиться в одном методе с инструкцией *goto*, которая ссылается на эту метку.
Инструкцию *goto* можно также использовать для перехода к *case-* или *default-ветви* внутри инструкции *switch*. Но в этом случае инструкция *goto* должна обязательно находиться в "рамках" той же инструкции *switch*. Это значит, что с помощью какой-либо "внешней" инструкции *goto* нельзя попасть в инструкцию *switch*.

Классы

Класс

Класс - это логическая абстракция, шаблон, который определяет форму объекта.

Он задает как данные, так и код, который оперирует этими данными и по сути, является чертежом для пользовательского типа данных. Определив класс, его можно использовать, загрузив в память. Класс, загруженный в память, называется объектом или экземпляром. О реализации класса нет смысла говорить до тех пор, пока не создан объект класса, и в памяти не появилось физическое его представление. Методы и переменные, составляющие класс, называются *членами* класса. Каждая программа C# имеет, по крайней мере, один класс.

Экземпляр класса создается с помощью ключевого слова C# new.

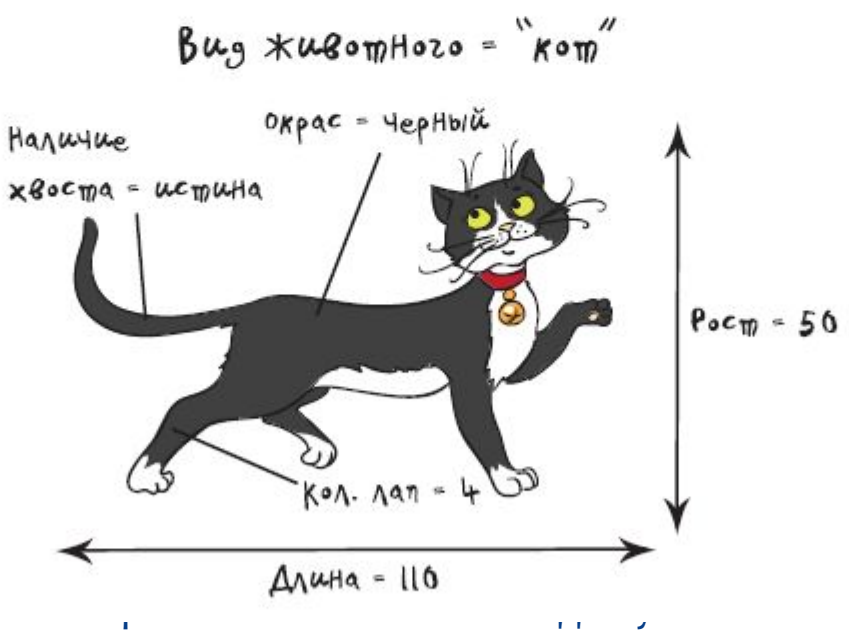
Общая форма определения класса

```
class имя_класса {
    // Объявление переменных экземпляров
    доступ тип переменная1;
    доступ тип переменнаяN;

    // Объявление методов
    доступ тип_возврата метод1 (параметры) {
        // тело метода1
    }
    доступ тип_возврата методM (параметры) {
        // тело методаM
    }
}
```

Свойства объектов

Перечислим несколько свойств животных:

* Вид животного (Kind of animal)
* Рост (Height)
* Длина (Length)
* Количество лап (Number of legs)
* Окрас (Color)
* Наличие хвоста (Has a tail)
* Является ли млекопитающим (Is a mammal)



При рассмотрении свойств этих животных мы м
них:

* Kind of animal = «Cat»
* Height = 50 cm
* Length = 110 cm
* Number of legs = 4
* Color = «Black»
* Has tail = true
* Is mammal = true

Поле

```
class Animal
{
    string kindOfAnimal;
    string name;
    int numberOfLegs;
    int height;
    int length;
    string color;
    bool hasTail;
    bool isMammal;
    bool spellingCorrect;
}
```

```
Animal Barsik;
Barsik = new Animal();
Barsik.kindOfAnimal = "Cat";
Barsik.name = "Кот Барсик";
Barsik.numberOfLegs = 4;
Barsik.height = 50;
Barsik.length = 110;
Barsik.color = "Black";
Barsik.hasTail = true;
Barsik.isMammal = true;
```

```
Animal X;
X = new Animal();

X.kindOfAnimal = "Cat";
X.height = 50;
X.length = 110;
X.color = "Black";
X.hasTail = true;
X.isMammal = true;
```

Создание объектов

Animal Barsik; Barsik=new Animal();	Animal Barsik=new Animal();
int x; x=5;	int x=5;

Barsik	Переменная ссылочного типа	Переменная *house* содержит ссылку на этот объект, а не объект
x	Переменная значимого типа	Переменная *x* содержит значение 5

Инкапсуляция

Группировка членов в классы имеет не только логический смысл, она также позволяет скрывать данные и функции, которые должны быть недоступны для другого кода. Этот принцип называют *инкапсуляцией*.

Спецификаторы доступа:

* **private** – объекты только этого класса могут обращаться к данному полю. При отсутствии спецификатора доступа член класса является закрытым (private) по умолчанию;
* **public** – объекты любого класса могут обращаться к этому полю;
* **protected** – только объекты классов-наследников могут обращаться к полю. Если построен класс *Animal*, то другой класс, например, класс *Mammal* (Млекопитающее), может объявить себя наследником класса *Animal*;
* **internal** - позволяет заявить о том, что некоторый член известен во всех файлах, входящих в состав компоновочного (известен только программе), но неизвестен вне его. Модификатор *internal* полезен при создании программных компонентов.

Инкапсуляция

Общие принципы, которыми следует руководствоваться при программировании классов

1. Члены, которые используются только внутри класса, следует определить как закрытые.
2. Данные экземпляров, которые должны находиться в пределах заданного диапазона, следует определить как закрытые, а доступ к ним обеспечить через открытые методы, выполняющие проверку вхождения в диапазон.
3. Если изменение члена может вызвать эффект, распространяющийся за пределы самого члена (т.е. действует на другие аспекты объекта), этот член следует определить как закрытый и обеспечить к нему контролируемый доступ.
4. Члены, при некорректном использовании которых на объект может быть оказано негативное воздействие, следует определить как закрытые, а доступ к ним обеспечить через открытые методы, предохраняющие эти члены от некорректного использования.
5. Методы, которые получают или устанавливают значения закрытых данных, должны быть открытыми.
6. Объявление переменных экземпляров открытыми допустимо, если нет причин делать их закрытыми.

```
class Animal
{
    public string kindOfAnimal;
    public string name;
    public int numberOfLegs;
    public int height;
    public int length;
    public string color;
    bool hasTail;
    protected bool isMammal;
    private bool spellingCorrect;
}

class Zoo
{
    Animal a = new Animal ();
        // Следующая строка будет выполнена успешно, поскольку классу “Zoo”
        // разрешено обращаться к открытым полям в классе “Animal”
    a.kindOfAnimal = “Kangaroo”;
        // Обе следующие строки НЕ будут выполнены, поскольку классу “Zoo”
        // не разрешено обращаться к закрытым или защищенным полям
    a.isMammal = false;         // Попытка обращения к защищенному методу
    a.spellingCorrect = true;  // Попытка обращения к закрытому методу
}
```

Методы

Переменные экземпляров и методы — две основные составляющие классов.

Методы — это процедуры (подпрограммы), которые манипулируют данными, определенными в классе, и во многих случаях обеспечивают доступ к этим данным. В хорошей программе один метод выполняет только одну задачу. Методами они называются потому, что именно в них описывается метод выполнения действий – пошаговые инструкции, задающие порядок выполнения операций.

Доступ к методам, аналогично доступу к полям класса, регулируется с помощью ключевых слов. По умолчанию все методы будут рассматриваться как **private** (закрытые).

```
доступ тип_возврата имя(список_параметров) {
        // тело метода
}
```

Методы

Пример добавления метода в класс

```
// Добавление метода в класс Building.
using System;
class Building {
    public int floors; // количество этажей
    public int area; // общая площадь здания
    public int occupants; // количество жильцов
    // Отображаем значение площади,
          приходящейся на одного человека
    public void areaPerPerson() {
        Console.WriteLine(" " + area / occupants +"
            приходится на одного человека");
    }
}
```

```
// Используем метод areaPerPerson()
class BuildingDemo {
    public static void Main() {
        Building house = new Building();
        Building office = new Building();
        // Присваиваем значения полям в объекте house
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;
        // Присваиваем значения полям в объекте office
        office.occupants = 25;
        office.area = 4200;
        office.floors = 3;
        Console.WriteLine("Дом имеет:\n " +house.floors + "
            этажа\n " +house.occupants + " жилыда\n "
            +house.area +" квадратных метров общей площади,
            из них");
        house.areaPerPerson();
        Console.WriteLine();
        Console.WriteLine("Офис имеет:\n " +office.floors + "
            этажа\n " +office.occupants + " работников\n "
            +office.area + квадратных метров общей площади, из
            них");
        office.areaPerPerson();
    }
```

Методы
Возврат значения

Методы могут возвращать значения вызывающим их процедурам:

return значение;

```
using System;
class Building {
   public int floors; // количество этажей
   public int area; // общая площадь здания
   public int occupants; // количество
      жильцов
   // Возврат значения площади, которая
      приходится на одного человека,
   public int areaPerPerson() {
       return area / occupants;
   }
}
```

```
class BuildingDemo {
   public static void Main() {
      Building house = new Building();
      // Присваиваем значения полям в объекте house
      house.occupants = 4;
      house.area = 2500;
      house.floors = 2;
      // Получаем для объекта house площадь,
         которая приходится на одного человека
      Console.WriteLine("Дом имеет:\n " +
         house.floors + " этажа\n " + house.occupants +
         " жильца\n " + house.area + " квадратных
         метров общей площади, из них\n
         house.areaPerPerson() + " приходится на
         одного человека");
      Console.WriteLine();
   }
}
```

Методы

Использование параметров

При вызове методу можно передать одно или несколько значений. Как упоминалось выше, значение, передаваемое методу, называется *аргументом*. Переменная внутри метода, которая принимает значение аргумента, называется *параметром*.

```
/* Метод класса Building возвращающий максимальное количество человек, если на каждого должна
   приходиться заданная минимальная площадь. */
public int maxOccupant(int minArea) {
    return area / minArea;
}
```

```
//Вызов метода maxOccupant() из класса BuildingDemo
Console.WriteLine("Максимальное число человек для офиса, \n" + "если на каждого
   должно приходиться " + 300 + " квадратных метров: " + office.maxOccupant(300));
```

Свойство

Свойство — это член, предоставляющий гибкий механизм для чтения, записи или вычисления значения частного (private) поля. Свойства можно использовать, как если бы они являлись открытыми членами данных, хотя в действительности они являются специальными методами, называемыми *методами доступа*. Это обеспечивает простой доступ к данным и позволяет повысить уровень безопасности и гибкости методов.

```
class TimePeriod
{
 private double seconds;
    public double Hours
    {
        get {
            return seconds / 3600;
        }
        set {
            seconds = value * 3600;
        }
    }
}
```

```
class Program
{
 static void Main()
 {
  TimePeriod t = new TimePeriod();
   t.Hours = 24;
  Console.WriteLine("Time in hours: " +
                t.Hours);
  }
 }
 // Output: Time in hours: 24
```

Конструктор

Очень часто встречаются классы с особым типом метода, называемым «**конструктором**». С точки зрения синтаксиса (правил языка) его особенность состоит в том, что имя *метода-конструктора совпадает с именем класса* и в объявление конструктора не включается *тип возвращаемого значения*. Содержательная специфика связана с предназначением конструктора — он нужен для создания (конструирования) объекта.

```
доступ имя_класса() {
    // тело конструктора
}
```

```
class Building {
    public int area; // общая площадь здания
    public int occupants; // количество жильцов
    public Building() {  //добавлен конструктор
        area = 4200;
        occupants = 25;
    }
    // Метод возвращает площадь на одного чел.
    public int areaPerPerson() {
        return area / occupants;
    }
}
```

```
// Используем параметризованный конструктор
    Building().
class BuildingDemo {
    public static void Main() {
        Building house = new Building();
        Console.WriteLine(«Площадь на одного
                человека  " +  house.areaPerPerson()) ;
    }
}
```

Параметризированный конструктор

Как и методы, конструкторы также могут перегружаться. Это дает возможность конструировать объекты самыми разными способами

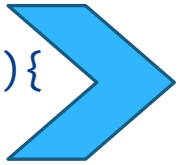
```
using System;
class Building {
    int floors; // количество этажей
    int area; // общая площадь здания
    int occupants; // количество жильцов
    public Building(int f, int o) {  //добавлен
        параметризованный конструктор Building
        floors = f;
        area = areaPerPerson() ;
        occupants = o;
    }
    // Метод возвращает площадь на одного чел.
    int areaPerPerson() {
        return floors *occupants;
    }
    // Метод возвращает макс.кол-во чел. в
        зданииpublic int maxOccupant(int minArea) {
        return area / minArea;
    }
}
```

```
// Используем параметризованный конструктор
    Building().
class BuildingDemo {
    public static void Main() {
        Building house = new Building(2, 4);
        Building office = new Building(3, 25);
        Console.WriteLine("Максимальное число
            человек для дома, \n" + "если на каждого
            должно приходиться " + 300 + " квадратных
            метров: " + house.maxOccupant(300)) ;
        Console.WriteLine("Максимальное число
            человек для офиса, \n" + "если на каждого
            должно приходиться " + 300 + " квадратных
            метров: " + office.maxOccupant(300));
    }
}
```

Ключевое слово this

При вызове метода ему автоматически передается неявно заданный аргумент, который представляет собой ссылку на вызывающий объект (т.е. объект, для которого вызывается метод). Эта ссылка и называется ключевым словом **this**.

```
using System;
class Rect {
    int width;
    int height;
    public Rect(int w, int h) {
        width = w;
        height = h;
    }
    public int area() {
        return width * height;
    }
}
```



```
using System;
class Rect {
    int width;
    int height;
    public Rect(int w, int h) {
        this.width = w;
        this.height = h;
    }
    public int area() {
        return this.width * this.height;
    }
}
```

```
class UseRect {
    public static void Main() {
        Rect rl = new Rect(4, 5);
        Rect r2 = new Rect(7, 9);
        Console.WriteLine("Пло
            щадь
            прямоугольника rl: "
            + rl.area());
        Console.WriteLine("Пло
            щадь
            прямоугольника r2:
            " + r2.area());
    }
}
```

Ключевое слово this

При вызове метода ему автоматически передается неявно заданный аргумент, который представляет собой ссылку на вызывающий объект (т.е. объект, для которого вызывается метод). Эта ссылка и называется ключевым словом **this**.

```
using System;
class Rect {
    int width;
    int height;
    public Rect(int width, int height)
    {
        this.width = width;
        this.height = height ;
    }
    public int area() {
        return width * height;
    }
}
```

```
class UseRect {
    public static void Main() {
        Rect rl = new Rect(4, 5);
        Rect r2 = new Rect(7, 9);
        Console.WriteLine("Площадь
            прямоугольника rl: " +
            rl.area());
        Console.WriteLine("Площадь
            прямоугольника r2: " +
            r2.area());
    }
}
```

Сбор "мусора" и использование деструкторов

При использовании оператора ***new*** объектам динамически выделяется память из пула свободной памяти.

Для освобождения ранее выделенной памяти в C# используется система сбора мусора (в C++ эту функцию выполняет оператор *delete*).

Система сбора мусора C# автоматически возвращает память для повторного использования, действуя незаметно и без вмешательства программиста. Если не существует ни одной ссылки на объект, то предполагается, что этот объект больше не нужен, и занимаемая им память освобождается. Эту (восстановленную) память снова можно использовать для размещения других объектов.

Система сбора мусора действует нерегулярно, случайным образом, во время выполнения отдельной программы. Эта система может и бездействовать: она не "включается" лишь потому, что существует один или несколько объектов, которые больше не используются в программе. Поскольку на сбор мусора требуется определенное время, динамическая система C# активизирует этот процесс только по необходимости или в специальных случаях. Таким образом, вы даже не будете знать, когда происходит сбор мусора, а когда - нет.

Сбор "мусора" и использование деструкторов

Деструкторы

Деструктор – метод, который вызывается непосредственно перед тем, как объект будет окончательно разрушен системой сбора мусора.

```
~имя_класса() {
// код деструктора
}
```

```
class Destruct {
    public int x;
    public Destruct(int i) {
        x = i;
    }
    // Вызывается при утилизации объекта
    ~Destruct() {
        Console.WriteLine("Деструктуризация " + x) ;
    }
    // Метод создает объект, который немедленно разрушается
    public void generator(int i) {
        Destruct o = new Destruct(i);
    }
}
```

```
class DestructDemo {
    public static void Main() {
        int count;
        Destruct ob = new Destruct(0);
        /* Теперь сгенерируем большое число
           объектов. В какой-то момент начнется
           сбор мусора. Замечание: возможно,
           для активизации этого процесса
           придется увеличить количество
           генерируемых объектов. */
        for(count=1; count < 100000; count++)
        ob.generator(count);
        Console.WriteLine("Готово!\a");
        Console.ReadLine();
    }
}
```

Наследование

Класс может наследовать от другого класса, что означает, что он включает все члены — открытые и закрытые — исходного класса, а также дополнительные определяемые им члены. Исходный класс называется *базовым классом*, а новый класс — *производным классом*. Производный класс создается для представления особых возможностей базового класса.

```
// Класс для двумерных объектов,
class TwoDShape {
    public double Width;
    public double Height;
    public void ShowDim() {
        Console.WriteLine("Ш и В равны " + Width + " и " + Height);
    }
}
// Класс Triangle, производный от класса TwoDShape.
class Triangle : TwoDShape {
    public string Style; // тип треугольника
    public double Area() {
        return Width * Height / 2;
    }
    // Показать тип треугольника,
    public void ShowStyle() {
        Console.WriteLine("Треугольник " + Style);
    }
}
```

```
class Shapes {
    static void Main() {
    Triangle t1 = new Triangle();
    t1.Width = 4.0;
    t1.Height = 4.0;
    t1.Style = "равнобедренный";
    Console.WriteLine("Сведения об объекте t1: ");
    t1.ShowStyle() ;
    t1.ShowDim() ;
    Console.WriteLine("Площадь равна" + tl.Area() ) ;
    }
}
```

Наследование

Доступ к членам класса и наследование

Для любого производного класса можно указать только один базовый класс. В C# не предусмотрено наследование нескольких базовых классов в одном производном классе.

Доступ к закрытым членам класса не наследуется.

```
class TwoDShape {
    double Width; // теперь это закрытая переменная
    double Height; // теперь это закрытая переменная
    public void ShowDim() {
        Console.WriteLine("Ширина и высота равны " + Width + " и " + Height);
    }
}

class Triangle : TwoDShape {
    public string Style; // тип треугольника
    // Возвратить площадь треугольника,
    public double Area() {
        return Width * Height /2; // Ошибка, доступ к закрытому члену класса запрещен
    }
}
```

Наследование

Доступ к членам класса и наследование

Ограничение на доступ к частным членам базового класса из производного класса снимается двумя способами:

1) применение открытых свойств для доступа к закрытым данным,
2) использование защищенных (***protected***) членов класса.

```
// Класс для двумерных объектов,
class TwoDShape {
    double pri_width; // теперь это закрытая переменная
    double pri_height; // теперь это закрытая переменная
    // Свойства ширины и высоты двумерного объекта,
    public double Width {
        get { return pri_width; }
        set { pri_width = value < 0 ? -value : value; }
    }
    public double Height {
        get { return pri_height; }
        set { pri_height = value < 0 ? -value : value; }
    }
    public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " + Width + " и " + Height);
    }
}
```

```
class Triangle : TwoDShape {
    public string Style; // тип треугольника
    public double Area() {
        return Width * Height / 2;
    }
}
```

Наследование

Доступ к членам класса и наследование

Ограничение на доступ к частным членам базового класса из производного класса снимается двумя способами:

1) применение открытых свойств для доступа к закрытым данным,
2) использование защищенных (***protected***) членов класса.

```
class B {
    protected int i, j; // члены, закрытые для класса B, но доступные для класса D
    public void Set(int a, int b) {
        i = a;
        j = b;
    }
    public void Show() {
        Console.WriteLine (i + " " + j);
    }
}
class D : B {
    int k; // закрытый член
    public void Setk() {
        k = i * j;
    }
    public void Showk() {
        Console.WriteLine(k) ;
    }
}
```

```
class ProtectedDemo {
    static void Main() {
    D ob = new D();
    ob.Set(2,3); // допустимо
    ob.Show(); // допустимо
    ob.Setk(); // допустимо
    ob.Showk(); // допустимо
    }
}
```

Статический класс

Статический член представляет собой метод или поле, доступ к которым можно получить без ссылки на определенный экземпляр класса. Самым общим статическим методом является Main, который представляет точку входа для всех программ C#.

Примером статического метода является **WriteLine()** в классе Console. При доступе к статическим методам необходимо обратить внимание на отличие в синтаксисе, с левой стороны оператора dot вместо имени экземпляра используется имя класса: Console.WriteLine().

В статическом классе все элементы также статические. Использование статических классов, методов и полей целесообразно в ряде случаев для повышения производительности и эффективности.

Статический класс может использоваться как обычный контейнер для наборов методов, работающих на входных параметрах, и не должен возвращать или устанавливать каких-либо внутренних полей экземпляра. Например, в библиотеке классов платформы .NET Framework статический класс System.Math содержит методы, выполняющие математические операции, без требования сохранять или извлекать данные, уникальные для конкретного экземпляра класса Math.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
```

Статический класс

Следующий список предоставляет основные характеристики статического класса:

* содержит только статические члены
* создавать его экземпляры нельзя
* он закрыт
* не может содержать конструкторов экземпляров
* его нельзя наследовать

По сути, создания статического класса аналогично созданию класса, содержащего только статические члены и закрытый конструктор. Закрытый конструктор не допускает создания экземпляров класса.

Статические классы не могут содержать конструктор экземпляров, но могут содержать статический конструктор. Нестатический класс также должен определять статический конструктор, если класс содержит статические члены, для которых нужна нетривиальная инициализация. Статический конструктор вызывается только один раз, и статический класс остается в памяти на время существования домена приложения, в котором находится программа.

Статический класс

Статический класс создается по приведенной ниже форме объявления класса, видоизмененной с помощью ключевого слова static

static class имя_класса { // ...

В таком классе все члены должны быть объявлены как static.

Статические классы применяются главным образом в двух случаях:

1) при создании метода расширения, которые связаны в основном с языком LINQ
2) для хранения совокупности связанных друг с другом статических методов. Именно это его применение и рассматривается ниже

Пример статического класса, содержащего два метода, преобразующих температуру по Цельсию в температуру по Фаренгейту и наоборот

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}
```

```
class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Статические методы

Для объявления статических методов класса используется ключевое слово **static** перед возвращаемым типом члена.

Статические члены инициализируются перед первым доступом к статическому члену и перед вызовом статического конструктора, если он имеется. Для доступа к члену статического класса следует использовать имя класса, а не имя переменной, указывая расположение члена.

Если класс содержит статические поля, должен быть предоставлен статический конструктор, который инициализирует эти поля при загрузке класса.

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Обработка ошибок и исключений

Try и Catch

```
try {
// Блок кода, подлежащий проверке на наличие ошибок.
}
catch (ExcepType1 exOb) {
// Обработчик для исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик для исключения типа ExcepType2.
}
finally  {
// Блок кода, подлежащий выполнению независимо от ошибки
}
```

ExcepType — это тип сгенерированного исключения

Обработка ошибок и исключений

Пример

```
class ExcDemo {
  public static void Main() {
    int [] numer = { 4, 8, 16, 32, 64, 128 };
    int [] denom = { 2, 0, 4, 4, 0, 8 };
    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " + denom[i] + " равно " + numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехватываем исключение.
        Console.WriteLine("Делить на нуль нельзя!");
      }
    }
  }
}
```

Результаты:
4 / 2 равно 2
Делить на нуль нельзя!
1 6 / 4 равно 4
3 2 / 4 равно 8
Делить на нуль нельзя!
128 / 8 равно 16

Обработка ошибок и исключений

Наиболее употребительные исключения

Исключение	Значение
Exception	Исключение общего вида (можно не указывать вообще)
ArrayTypeMismatchException	Тип сохраняемого значения несовместим с типом массива
DivideByZeroException	Попытка деления на нуль
IndexOutOfRangeException	Индекс массива оказался вне диапазона
InvalidCastException	Неверно выполнено динамическое приведение типов
OutOfMemoryException	Обращение к оператору new оказалось неудачным из-за недостаточного объема свободной памяти
OverflowException	Имеет место арифметическое переполнение
StackoverflowException	Переполнение стека

Обработка ошибок и исключений

Генерирование исключения вручную

```
class ThrowDemo {
  public static void Main() {
    try {
      Console.WriteLine("До генерирования исключения.");
      throw new DivideByZeroException();
    }
    catch (DivideByZeroException) {
      // Перехватываем исключение.
      Console.WriteLine("Исключение перехвачено.");
    }
Console.WriteLine("После try/catch-блока.");
  }
}
```

Обработка ошибок и исключений

Изменение стандартных сообщений для исключений

```
class ProgramThrow
{
static void DoWork(int x)
   {
if (x > 5)
      {
throw new System.ArgumentOutOfRangeException("X is too large");
      }
   }
static void Main()
   {
try
      {
DoWork(10);
      }
catch (System.ArgumentOutOfRangeException ex)
      {
System.Console.WriteLine(ex.Message);
}
   }
}
```

Строки

Строка C# представляет собой группу одного или нескольких знаков, объявленных с помощью ключевого слова **string**, которое является ускоренным методом языка C# для класса System.String.

```
string s1 = "The string ";
```

Строковые объекты являются неизменяемыми: после создания их нельзя изменить. Методы, работающие со строками, возвращают новые строковые объекты.

Все встроенные типы данных C# предоставляют метод **ToString()**, преобразующий значение в строку. Этот метод может быть использован для преобразования числовых значений в строки.

```
int year = 1999;
string msg = "Eve was born in " + year.ToString();
System.Console.WriteLine(msg);
```

Строки
Работа со строками

Доступ к отдельным знакам

К отдельным знакам, содержащимся в строке, можно получить доступ с помощью таких методов как Substring(), Replace(), IndexOf().

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Строки
Работа со строками

Смена регистра

Чтобы изменить регистр букв в строке (сделать их заглавными или строчными) следует использовать **ToUpper()** или **ToLower()**

```
string s6 = "Battle of Hastings, 1066";

System.Console.WriteLine(s6.ToUpper());
// outputs "BATTLE OF HASTINGS 1066"

System.Console.WriteLine(s6.ToLower());
// outputs "battle of hastings 1066"
```

Строки

Наиболее часто используемые методы обработки строк

Метод	Описание
static string Copy (string str)	Возвращает копию строки str
int CompareTo (string str)	Возвращает отрицательное значение, если вызывающая строка меньше строки str, положительное значение, если вызывающая строка больше строки str , и нуль, если сравниваемые строки равны
int IndexOf (string str)	Выполняет в вызывающей строке поиск подстроки, заданной параметром str. Возвращает индекс первого вхождения искомой подстроки или - 1 , если она не будет обнаружена
int LastindexOf (string str)	Выполняет в вызывающей строке поиск подстроки, заданной параметром str. Возвращает индекс последнего вхождения искомой подстроки или - 1 , если она не будет обнаружена
string ToLower()	Возвращает строчную версию вызывающей строки
string Toupper()	Возвращает прописную версию вызывающей строки

Строки

В целях повышения производительности большие объемы работы по объединению строк или другие операции следует выполнять в классе **StringBuilder**. Класс StringBuilder также позволяет заново присваивать отдельные знаки, что не поддерживается встроенным строковым типом данных.

В следующем примере создается объект StringBuilder и его содержимое последовательно добавляется с помощью метода **Append()**.

```
class TestStringBuilder {
  static void Main(){
     System.Text.StringBuilder sb = new System.Text.StringBuilder();
     for (int i = 0; i< 10; i++)
      {
        sb.Append(i.ToString());
      }
      System.Console.WriteLine(sb);  // displays 0123456789
       sb[0] = sb[9];
      system.Console.WriteLine(sb);  // displays 9123456789
   }
}
```

Массивы

Массивы являются коллекциями объектов одного типа. Поскольку длина массивов практически не ограничена, они могут использоваться для хранения тысяч или даже миллионов объектов, но размер массива должен быть указан при его создании. Каждый элемент массива доступен по числовому индексу, указывающему позицию или ячейку, в которой объект хранится в массиве. Массивы могут хранить ссылочные типы и типы значений.

Одномерные массивы

Одномерный массив объектов объявляется следующим образом:

```
type[] arrayName;
```

Элементы в массиве могут инициализироваться при объявлении:

```
int[] array = new int[5];
```

Значение по умолчанию числовых элементов массива задано равным нулю, а элементы ссылок имеют значение null, но значения можно инициализировать при создании массива следующим образом:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

или

```
int[] array2 = {1, 3, 5, 7, 9};
```

Индексация массивов начинается с нуля, поэтому номер первого элемента массива равен 0.

```
string[] days = {"Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat"};
System.Console.WriteLine(days[0]);  // Outputs "Sun"
```

Многомерные массивы

Концептуально, многомерный массив с двумя измерениями напоминает сетку. Многомерный массив с тремя измерениями напоминает куб.

```
тип[, . . ., ] имя_массива = new тип[размер1, размер2, . . . размеры] ;
```

```
int[,,] multidim = new int[4, 10, 3];
```

```
тип[,] имя_массива = {
{val, val, val, ..., val},
{val, val, val, ..., val},
{val, val, val, ..., val}
};
```

```
int[,] sqrs = {
{ 1, 1 },
{ 2, 4 },
{3, 9 },
{ 4, 16 },
{ 5, 25 },
{ 6, 36 },
{ 7, 49 },
{ 8, 64 },
{ 9, 81 },
{ 10, 100 }
};
```

10 строк по 2 столбца

Многомерные массивы

Пример

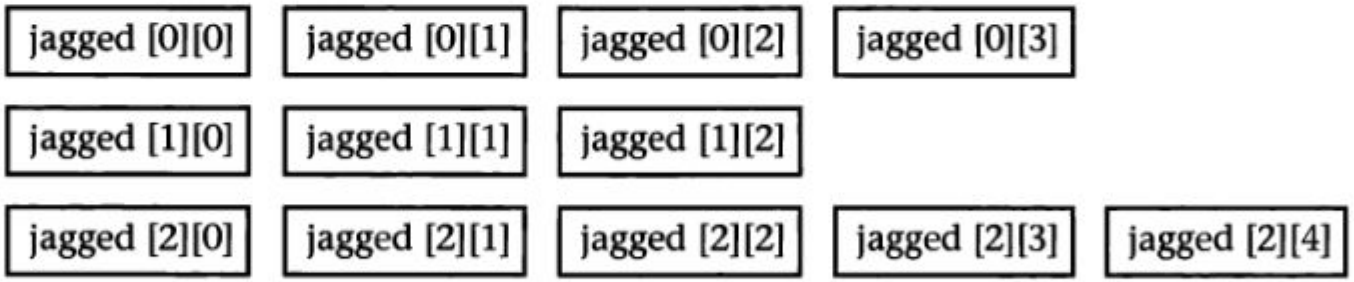
```
// declare multidimension array (two dimensions)
int[,] array2D = new int[2,3];

// declare and initialize multidimension array
int[,] array2D2 = { {1, 2, 3}, {4, 5, 6} };

// write elements in a multidimensional array
for (int i=0; i<2; i++)
{
 for (int j=0; j<3; j++)
   {
     array2D[i,j] = (i + 1) * (j + 1);
   }
}

// read elements in a multidimensional array
for (int i=0; i<2; i++)
{
  for (int j=0; j<3; j++)
    {
      System.Console.Write(array2D[i,j]);
    }
  System.Console.WriteLine();
}
```

Массивы массивов (ступенчатые массивы)

Представляет собой одномерный массив, в котором каждый элемент является массивом. Элементы массива не обязаны иметь одинаковый размер и тип. Объявить массив массивов можно следующим образом:

тип[] [] имя_массива = new тип [размер] [];

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```




```
jagged[2][1] = 10;
```

Массивы

Способы инициализации нескольких видов массивов: одномерного, многомерного и массива массивов

```
// Single-dimensional array (numbers)
int[] n1 = new int[4] {2, 4, 6, 8};
int[] n2 = new int[] {2, 4, 6, 8};
int[] n3 = {2, 4, 6, 8};

// Single-dimensional array (strings).
string[] s1 = new string[3] {"John", "Paul", "Mary"};
string[] s2 = new string[] {"John", "Paul", "Mary"};
string[] s3 = {"John", "Paul", "Mary"};

// Multidimensional array
int[,] n4 = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
int[,] n5 = new int[,] { {1, 2}, {3, 4}, {5, 6} };
int[,] n6 = { {1, 2}, {3, 4}, {5, 6} };

// Jagged array
int[][] n7 = new int[2][] { new int[] {2,4,6}, new int[] {1,3,5,7,9} };
int[][] n8 = new int[][] { new int[] {2,4,6}, new int[] {1,3,5,7,9} };
int[][] n9 = { new int[] {2,4,6}, new int[] {1,3,5,7,9} };
```

Массивы объектов

Создание массива объектов в отличие от создания массива простых типов данных, например целочисленных, происходит в два этапа. Сначала необходимо объявить массив, а затем создать объекты для хранения в нем. В этом примере создается класс, определяющий аудио компакт-диск. Затем создается массив для хранения 20 аудио компакт-дисков.

```
class CD {
    private string album;
    private string artist;
    public string Album
        {
            get {return album;}
            set {album = value;}
        }
    public string Artist
        {
            get {return artist;}
            set {artist = value;}
        }
}
```

```
class Program {
    static void Main(string[] args) {
        CD[] cdLibrary = new CD[20];
        for (int i=0; i<20; i++)
         {
            cdLibrary[i] = new CD();
         }
        cdLibrary[0].Album = "See";
        cdLibrary[0].Artist = "The Sharp ";
    }
}
```

Перечисления

Тип данных **enum** позволяет объявить набор имен или других значений литералов, определяющих все возможные значения, которые могут быть назначены переменной.

```
enum имя_типа {
    имя_1=значение,
    имя_n=значение
}
```

Использование этого типа данных способствует повышению удобочитаемости кода, кроме того, снижается вероятность назначения переменной недопустимого или неожиданного значения.

```
Public enum DayOfWeek
{
   Sunday = 0,
   Monday = 1,
   Tuesday = 2,
   Wednesday = 3,
   Thursday = 4,
   Friday = 5,
   Saturday = 6
}
```

```
class Program {
    static void Main() {
        DayOfWeek day = DayOfWeek.Monday;
        int i = (int) DayOfWeek.Monday;

        System.Console.WriteLine(day);  // Monday
        System.Console.WriteLine(i);    // 1
    }
}
```

Коллекции

Коллекция представляет собой совокупность объектов. Коллекции упрощают решение многих задач программирования благодаря тому, что предлагают готовые решения для создания целого ряда типичных, но порой трудоемких для разработки структур данных. Так, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Главное преимущество коллекций: они стандартизируют обработку групп объектов в программе.

Все коллекции разработаны на основе набора четко определенных интерфейсов. В среде .NET Framework поддерживаются пять типов коллекций:

1) необобщенные,
2) специальные,
3) с поразрядной организацией,
4) обобщенные,
5) параллельные.

В пространстве имен System.Collections.ObjectModel находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций.

Коллекции

Необобщенные коллекции

Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". Необобщенные коллекции оперируют данными типа object. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Очевидно, что такие коллекции не типизированы, поскольку в них хранятся ссылки на данные типа object. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен System.Collections.

Специальные коллекции

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк. Специальные коллекции объявляются в пространстве имен System.Collections.Specialized.

Поразрядная коллекция

Коллекция типа BitArray поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами (например, И, ИЛИ, исключающее ИЛИ) и существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа BitArray объявляется в пространстве имен System.Collections.

Коллекции

Обобщенные коллекции

Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен System.Collections.Generic.

Параллельные коллекции

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен System.Collections.Concurrent.

Основополагающим для всех коллекций является понятие *перечислителя.* С перечислителем непосредственно связано другое средство, называемое *итератором*. Это средство упрощает процесс создания классов коллекций, например специальных, поочередное обращение к которым организуется в цикле foreach.

Коллекции

ArrayList – относится к необобщённым коллекциям и определяет динамический массив, т.е. такой массив, который может при необходимости увеличивать и уменьшать свой размер

Некоторые методы ArrayList	
Метод	Описание
void AddRange (Icollection c)	Добавляет элементы из коллекции с в конец вызывающей коллекции типа ArrayList
int BinarySearch (object value).	Выполняет поиск в вызывающей коллекции значения value. Возвращает индекс найденного элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
void CopyTo (Array array)	Копирует содержимое вызывающей коллекции в массив array, который должен быть одномерным и совместимым по типу с элементами коллекции
void Reverse()	Располагает элементы вызывающей коллекции в обратном порядке
void Sort ()	Сортирует вызывающую коллекцию по нарастающей
array ToArray(Type type)	Возвращает массив, содержащий копии элементов вызывающего объекта. Тип элементов этого массива определяется параметром type
int Add(object value)	Добавляет объект в конец списка
int Remove(object value)	Удаляет первое вхождение объекта из списка

Коллекции

Пример 1 использования коллекции ArrayList

(создание, добавление, удаление , вывод)

```
using System;
using System.Collections;

class ArrayListDemo {
  static void Main() {
    // Создать коллекцию в виде динамического массива.
    ArrayList al = new ArrayList();

    Console.WriteLine("Исходное количество элементов: " + al.Count);

    Console.WriteLine();

    Console.WriteLine("Добавить 6 элементов");
    // Добавить элементы в динамический массив.
    al.Add('C');
    al.Add('A');
    al.Add('E');
    al.Add('B');
    al.Add('D');
    al.Add('F');

    Console.WriteLine("Количество элементов: " + al.Count);
```

```
Исходное количество элементов: 0

Добавить 6 элементов
Количество элементов: 6
```

Коллекции

Пример 1 использования коллекции ArrayList
(создание, добавление, удаление , вывод)

```
// Отобразить содержимое динамического массива,
// используя индексирование массива.
Console.Write("Текущее содержимое: ");
for(int i=0; i < al.Count; i++)
  Console.Write(al[i] + " ");
Console.WriteLine("\n");

Console.WriteLine("Удалить 2 элемента");
// Удалить элементы из динамического массива.
al.Remove('F');
al.Remove('A');

Console.WriteLine("Количество элементов: " + al.Count);

// Отобразить содержимое динамического массива, используя цикл foreach.
Console.Write("Содержимое: ");
foreach(char c in al)
  Console.Write(c + " ");
Console.WriteLine("\n");
```

```
Текущее содержимое: C A E B D F

Удалить 2 элемента
Количество элементов: 4
Содержимое: C E B D
```

Коллекции

Пример 1 использования коллекции ArrayList

(создание, добавление, удаление , вывод)

```
Console.WriteLine("Добавить еще 20 элементов");
// Добавить количество элементов, достаточное для
// принудительного расширения массива.
for(int i=0; i < 20; i++)
  al.Add((char)('a' + i));
Console.WriteLine("Текущая емкость: " + al.Capacity);   // Capacity кратно 8
Console.WriteLine("Количество элементов после добавления 20 новых: " +
                  al.Count);
Console.Write("Содержимое: ");
foreach(char c in al)
  Console.Write(c + " ");
Console.WriteLine("\n");

// Изменить содержимое динамического массива,
// используя индексирование массива.
Console.WriteLine("Изменить три первых элемента");
al[0] = 'X';
al[1] = 'Y';
al[2] = 'Z';
Console.Write("Содержимое: ");
foreach(char c in al)
  Console.Write(c + " ");
Console.WriteLine();
```

```
Добавить еще 20 элементов
Текущая емкость: 32
Количество элементов после добавления 20 новых: 24
Содержимое: C E B D a b c d e f g h i j k l m n o p q r s t

Изменить три первых элемента
Содержимое: X Y Z D a b c d e f g h i j k l m n o p q r s t
```

Коллекции

Пример 2 использования коллекции ArrayList
(сортировка, поиск)

```
class SortSearchDemo {
  static void Main() {
    // Создать коллекцию в виде динамического массива.
    ArrayList al = new ArrayList();
    // Добавить элементы в динамический массив.
    al.Add(55);
    al.Add(43);
    al.Add(-4);
    al.Add(88);
    al.Add(3);
    al.Add(19);
    Console.Write("Исходное содержимое: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");
    // Отсортировать динамический массив.
    al.Sort();
    // Отобразить содержимое динамического массива, используя цикл foreach
    Console.Write("Содержимое после сортировки: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");
    Console.WriteLine("Индекс элемента 43: " +
    al.BinarySearch(43));
  }
}
```

```
Исходное содержимое: 55 43 -4 88 3 19
```

```
Содержимое после сортировки: -4 3 19 43 55 88
```

```
Индекс элемента 43: 3
```

Коллекции

Пример 3 использования коллекции ArrayList
(получение массива из коллекции)

```
class ArrayListToArray {
  static void Main() {
    ArrayList al = new ArrayList();
    al.Add(1);
    al.Add(2);
    al.Add(3);
    al.Add(4);
    Console.Write("Содержимое: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine();

    // Получить массив.
    int[] ia = (int[]) al.ToArray(typeof(int));
    int sum = 0;

    // Просуммировать элементы массива.
    for(int i=0; i<ia.Length; i++)
      sum += ia[i];
    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

```
Содержимое: 1 2 3 4
```

```
Сумма равна: 10
```

Коллекции

List<T> - представляет простейший список однотипных объектов

Некоторые методы List<T>	
Метод	Описание
void Add(T item)	добавление нового элемента в список
void AddRange(ICollection collection)	добавление в список коллекции или массива
int BinarySearch(T item)	бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован
int IndexOf(T item)	возвращает индекс первого вхождения элемента в списке
void Insert(int index, T item)	вставляет элемент *item* в списке на позицию *index*
bool Remove(T item)	удаляет элемент *item* из списка, и если удаление прошло успешно, то возвращает *true*
void RemoveAt(int index)	удаление элемента по указанному индексу *index*
void Sort()	сортировка списка

Коллекции

List<T> - представляет простейший список однотипных объектов

```
using System;
using System.Collections.Generic;

namespace Collections
{
 class Program
    {
        static void Main()
        {
         List<int> num = new List<int>() { 1, 2, 3, 45 };
         num.Add(10); // добавление элемента ’10’:  1, 2, 3, 45, 10
         num.AddRange(new int[] { 7, 8, 9 }); //  1, 2, 3, 45, 10, 7, 8, 9
         num.Insert(0, 77); // вставляем на первое место в списке число 77: 77, 1, 2, 3, 45, 10, 7, 8, 9
         num.RemoveAt(1); //  удаляем второй элемент: 77, 2, 3, 45, 10, 7, 8, 9
         foreach (int i in num)
          {
             Console.WriteLine(i); //  вывод в консоль: 77, 2, 3, 45, 10, 7, 8, 9

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

Коллекции

List<T> - представляет простейший список однотипных объектов

```
using System;
using System.Collections.Generic;

namespace Collections
{
  class Person
    {
        public string Name { get; set; }
    }

class Program
    {
        static void Main()
        {
            List<Person> people = new List<Person>(3);
            people.Add(new Person() { Name = "Том" });
            people.Add(new Person() { Name = "Билл" });

            foreach (Person p in people)
            {
              Console.WriteLine(p.Name);
            }

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

Структуры

Структура в C# аналогична классу, однако в структурах отсутствуют некоторые возможности, например наследование. Кроме того, поскольку структура является типом значения, ее можно создать быстрее, чем класс. При наличии непрерывного цикла, где создается большое количество новых структур данных, вместо класса рекомендуется использовать структуру. Структуры используются для инкапсуляции групп полей данных. Структуры являются Типами Значений, а классы — Ссылочными Типами.

В отличие от классов, структуры можно создавать без использования оператора *new*. Структура определяется с помощью ключевого слова **struct**

Структуры

Пример

```
public struct CoOrds
{
   public int x, y;

   public CoOrds(int p1, int p2)
   {
      x = p1;
      y = p2;
   }
}
```

```
class TestCoOrds
{
   static void Main()
   {
      CoOrds coords1 = new CoOrds();
      CoOrds coords2 = new CoOrds(10, 10);
      CoOrds coord3;
      coord3.x=20;
      coord3.y=20;
      Console.WriteLine("x = {0}, y = {1}", coords1.x, coords1.y);
      Console.WriteLine("x = {0}, y = {1}", coords2.x, coords2.y);
      Console.WriteLine("x = {0}, y = {1}", coords3.x, coords3.y);
   }
}
```

```
/* Output:
 CoOrds 1: x = 0, y = 0
 CoOrds 2: x = 10, y = 10
CoOrds32: x = 20, y = 20
*/
```

Полиморфизм

Полиморфизмом называют возможность производного класса изменять или *переопределять* методы, которые он наследует от базового класса. Эта функция используется, если в методе, который имеет отличия либо не определен в базовом классе, нужно выполнить какие-то особые действия.

Перегрузка методов

В C# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса, при условии, что их параметры объявляются по-разному. В этом случае говорят, что методы перегружаются, а сам процесс называется перегрузкой методов. Перегрузка методов относится к одному из способов реализации полиморфизма в C#. Методы должны также отличаться типами или числом своих параметров.

```
class Overload {
  public void OvlDemo() {
    Console.WriteLine("Без параметров");
  }
 public void OvlDemo(int a) {
    Console.WriteLine("Один параметр: " + a);
  }
 public void OvlDemo(int a, int b) {
    Console.WriteLine("Два параметра: " + a + " " + b);
     }
public double OvlDemo(int d, int c) {
   return d + c;
  }
}
```

```
class OverloadDemo {
  static void Main() {
    Overload ob = new Overload();
    ob.OvlDemo() ;
    ob.OvlDemo(2);
    resI = ob.OvlDemo(4, 6);
    resD = ob.OvlDemo(1.1, 2.32);
  }
}
```

Необязательные аргументы

В версии C# 4.0 внедрено новое средство, повышающее удобство указания аргументов при вызове метода. Это средство называется необязательными аргументами и позволяет определить используемое по умолчанию значение для параметра метода. Данное значение будет использоваться по умолчанию в том случае, если для параметра не указан соответствующий аргумент при вызове метода.

```
void OptArgMeth(int alpha, int beta=10, int gamma = 20) {
//beta и gamma являются необязательными аргументами
}

// Передать все аргументы явным образом.
OptArgMeth(1, 2, 3);
// Сделать аргумент gamma необязательным.
OptArgMeth(1, 2);
// Сделать оба аргумента beta и gamma необязательными.
OptArgMeth(1);
```

Именованные аргументы

Именованные аргументы были внедрены в версии C# 4.0. Именованный аргумент позволяет указать имя того параметра, которому присваивается его значение. И в этом случае порядок следования аргументов уже не имеет никакого значения. Для указания аргумента по имени служит следующая форма синтаксиса.

имя_параметра : значение

```
// Выяснить, делится ли одно значение нацело на другое,
 static bool IsFactor(int val, int divisor) {
  if((val % divisor) == 0) return true;
    return false;
 }
```

```
static void Main() {
 if(IsFactor (10, 2))
    Console.WriteLine ("2 - множитель 10");
 if(IsFactor(val: 10, divisor: 2))
    Console.WriteLine("2 - множитель 10");
 if(IsFactor(divisor: 2, val: 10))
    Console.WriteLine("2 - множитель 10");
 if(IsFactor(10, divisor: 2))
    Console.WriteLine ("2 - множитель 10");
}
```

Применение оператора new к переменным типа значений

В C# переменная типа значения содержит собственное значение. Во время компиляции программы компилятор автоматически выделяет память для хранения этого значения. Следовательно, нет необходимости использовать оператор *new* для явного выделения памяти. И напротив, в переменных ссылочного типа хранится ссылка на объект, а память для хранения этого объекта выделяется динамически, т.е. во время выполнения программы. Тем не менее вполне допустимо использовать оператор *new* и с типами значений.

```
int i = new int () ;
```

В этом случае вызывается конструктор по умолчанию для типа int , который инициализирует переменную i нулем. Без оператора *new* переменная i осталась бы неинициализированной, и попытка использовать ее, например, в методе WriteLine() без явного присвоения ей конкретного значения привела бы к ошибке.

Передача объектов методам

```
class MyClass {
  int alpha, beta;
  public MyClass(int i, int j) {
    alpha = i;
    beta = j ;
  }
public bool sameAs(MyClass ob) {
   if((ob.alpha == alpha) & (ob.beta == beta))
     return true;
    else
     return false;
  }
public void copy(MyClass ob) {
    alpha = ob.alpha;
    beta = ob.beta;
  }
  public void show() {
    Console.WriteLine("alpha: {0}, beta: {1}",
alpha, beta);
  }
}
```

```
class PassOb {
  public static void Main() {
   MyClass ob1 = new MyClass(4, 5);
   MyClass ob2 = new MyClass(6, 7);
    ob1.show();
    ob2.show();
   if(ob1.sameAs(ob2))
     Console.WriteLine("ob1 и ob2 одинаковы");
   else
     Console.WriteLine("ob1 и ob2 разные ");
  ob1.copy(ob2);
  ob1.show();
   if(ob1.sameAs(ob2))
      Console.WriteLine("ob1 и ob2 одинаковы");
   else
     Console.WriteLine("ob1 и ob2 разные ");
  }
}
```

Передача аргументов по значению и по ссылке

Передача аргументов по значению (call-by-value)

В этом случае *значение* аргумента копируется в формальный параметр метода. Следовательно, изменения, внесенные в параметр метода, не влияют на аргумент, используемый при вызове.

```
class Test {
  // Этот метод не оказывает
влияния на аргументы,
используемые в его вызове
  public void noChange(int i, int j)
{
    i = i + j;
    j = - j ;
  }
}
```

```
class CallByValue {
  public static void Main() {
    Test ob = new Test();
    int a = 15, b = 20;
    Console.WriteLine("a и b до: " + a + " " + b) ;
    ob.noChange(a, b ) ;
    Console.WriteLine("a и b после : " + a + " " + b) ;
  }
}
```

Результаты выполнения этой программы выглядят так:

```
a и b до: 15  20
a и b после: 15  20
```

Передача аргументов по значению и по ссылке

Передача аргументов по ссылке (call-by-reference)

Здесь для получения доступа к реальному аргументу, заданному при вызове, используется ссылка на аргумент. Это значит, что изменения, внесенные в параметр, окажут воздействие на аргумент, использованный при вызове метода

```
class Test {
  public int a, b;
  public Test(int i, int j) {
    a = i;
    b = j;
  }
public void change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}
```

```
class CallByRef {
  public static void Main() {
    Test ob = new Test(15, 20);
    Console.WriteLine("ob.a и ob.b до:"+ ob.a + " " +
ob.b);
    ob.change(ob);
    Console.WriteLine("ob.a и ob.b после:" +ob.a + "
" + ob.b);
  }
}
```

Результаты выполнения этой программы выглядят так:

ob.a и ob.b до: 15 20

ob.a и ob.b после: 35 -20

Использование ref- и out-параметров

Используя ключевые слова **ref** и **out**, можно передать значение любого нессылочного типа по ссылке. Тем самым мы позволим методу изменить аргумент, используемый при вызове.

Модификатор **ref** заставляет C# организовать вместо вызова по значению вызов по ссылке. Аргументу, передаваемому методу "в сопровождении" модификатора **ref**, должно быть присвоено значение до вызова метода.

Модификатор **out** позволяет методу вернуть более одного значения. Метод может возвращать посредством out-параметров столько значений, сколько нужно.

Использование модификатора ref

Пример: изменение передаваемого аргумента в методе

```
class RefTest {
   // Этот метод изменяет свои аргументы. Обратите внимание на использование модификатора ref
  public void sqr(ref int i) {
    i = i * i;
  }
}

class RefDemo {
  public static void Main() {
    RefTest ob = new RefTest();
    int a = 10;
    Console.WriteLine("a перед вызовом: " + a );          // 10
    ob.sqr(ref a); // Обратите внимание на использование модификатора ref
    Console.WriteLine("a после вызова: " + a ) ;          //  100
  }
}
```

Использование модификатора ref

Пример: обмен значениями двух аргументов

```
class Swap {
  public void swap(ref int a, ref int b) {
    int t;
    t = a;
    a = b;
    b = t;
  }
}

class SwapDemo {
  public static void Main() {
    Swap ob = new Swap();
    int x = 10, y = 20;
    Console.WriteLine("x и y перед вызовом: " +x + " " + y) ;      // 10   20
    ob.swap(ref x, ref y) ;
    Console.WriteLine("x и y после вызова: " + x + " " + y) ;        // 20   10
  }
}
```

Использование модификатора out

Инструкция return позволяет методу возвратить значение тому, кто сделал вызов. Однако метод может вернуть в результате одного вызова только одно значение. Модификатор **out** позволяет методу вернуть более одного значения. Метод может возвращать посредством out-параметров столько значений, сколько нужно.

Использование модификатора out

Пример: разделение числа на целую и дробную части

```
class Decompose {
  public int parts(double n, out double frac) {
    int whole;
    whole = (int) n;
    frac = n - whole; // Передаем дробную часть посредством параметра frac
    return whole; // Возвращаем целую часть числа
  }
}

class UseOut {
  public static void Main() {
    Decompose ob = new Decompose();
    int i;
    double f;
    i = ob.parts(10.125, out f);
    Console.WriteLine("Целая часть числа равна " + i);          //  10
    Console.WriteLine("Дробная часть числа равна " + f );       //  0.125
  }
}
```

Использование переменного количества аргументов

Для передачи в метод произвольного количества аргументов необходимо применить специальный тип параметра- модификатор **params**, который используется для объявления параметра-массива. Количество элементов в массиве будет равно числу аргументов, переданных методу.

Метод(**params** тип[] имя)

Параметр-массив может быть только один и должен стоять последним по порядку среди параметров.

Использование переменного количества аргументов
Пример: нахождение минимального числа

```
class Min {
 public int minVal(params int[] nums) {
  int m;
  if(nums.Length ==0) {
            Console.WriteLine("Ошибка: нет
                                аргументов");
   return 0;
  }
  m = nums [0];
  for (int i=1; i < nums.Length; i++)
   if(nums[i] < m)   m = nums[i];
  return m;
 }
}
```

```
class ParamsDemo {
 public static void Main() {
  Min ob = new Min();
  int min;
  int a = 10, b = 20;
   Console.WriteLine("Минимум равен
                      " + ob.minVal(a, b));
  Console.WriteLine("Минимум равен "
                    + ob.minVal(a, b, -1));
  Console.WriteLine("Минимум равен "
                + ob.minVal(18, 23, 3, 14, 25));
   // Этот метод можно также вызвать
                                с int-массивом
  int[] args = { 45, 67, 34, 9, 112, 8 };
  Console.WriteLine("Минимум равен "
                    + ob.minVal(args));
 }
}
```

Возвращение методами массивов

Пример: Метод возвращает массив, содержащий множители параметра num. Out-параметр numfactors будет содержать количество найденных множителей

```
class Factor {
      public int[] findfactors(int num, out int
                                        numfactors) {
   int[] facts = new int[80];
   int i, j;
    // Находим множители и помещаем их
                                        в массив facts
   for(i=2, j=0; i < num/2 + 1; i++)
    if( (num%i) == 0 ) {
      facts[j] = i;
      j++;
    }
    numfactors = j;
    return facts;
  }
}
```

```
class FindFactors {
 public static void Main() {
   Factor f = new Factor();
   int numfactors;
   int[] factors;
            factors = f.findfactors(1000, out
                                        numfactors);
    Console.WriteLine("Множители числа
                                        1000: " ) ;
   for(int i=0; i < numfactors; i++)
     Console.WriteLine (factors[i]) ;
  }
}
```

```
Множители числа 1000:
2 4 5 8 10 20 25 40 50 100 125 200 250 500
```

Наследование

Вызов конструкторов базового класса

```
class TwoDShape {
 double pri_width;
 double pr.i_height;
 public double Width {
  get { return pri_width; }
  set { pri_width = value < 0 ?
                    -value : value; }
 }
 public double Height {
  get { return pri_height; }
  set { pri_height = value < 0 ?
                    -value : value; }
 }
 public void ShowDim() {
  Console.WriteLine("Ширина
     и длина равны " + Width +
     " и " + Height);
 }
}
```

```
class Triangle : TwoDShape {
 string Style;
 public Triangle(string s,
                 double w,
                 double h) {
  Width = w;
  Height = h;
  Style = s;
 }
 public double Area() {
  return Width * Height / 2;
 }
 public void ShowStyle() {
  Console.WriteLine("Треуго
                 льник " +
                 Style);
}
```

```
class Shapes3 {
 static void Main() {
  Triangle t1 = new
      Triangle("равнобедренны
      й", 4.0, 4.0);
  Triangle t2 = new
      Triangle("прямоугольный
      ", 8.0, 12.0);
  Console.WriteLine(«Объект
      t1: ");
  t1. ShowStyle() ;
  t1.ShowDim();
  Console.WriteLine («S= " +
      t1.Area());
  Console.WriteLine ();
  Console.WriteLine(“Объект
      t2: ");
  t2.ShowStyle();
  t2.-ShowDim() ;
  Console.WriteLine(“S= " +
      t2.Area());
 }
}
```

Вызов конструкторов базового класса

В иерархии классов допускается, чтобы у базовых и производных классов были свои собственные конструкторы.

Конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта.

Для обращения к конструктору базового класса используется ключевое слово : base, которое находит двоякое применение:

1) вызова конструктора базового класса;
2) для доступа к члену базового класса, скрывающегося за членом производного класса.

Общая форма расширенного объявления:

```
конструктор_производного_класса ( список_параметров) : base (список_аргументов) {
  // тело конструктора
}
```

Вызов конструкторов базового класса

```
class TwoDShape {
  double pri_width;
  double pri_height;
  public TwoDShape(double w, double h)
  {
    Width = w;
    Height = h;
  }
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value :
    value; }
  }
  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value :
    value; }
  }
  public void ShowDim() {
    Console.WriteLine(Width+"и"+Height);
  }
}
```

```
class Triangle : TwoDShape {
  string Style;
  public Triangle(string s,
      double w, double h) :
      base(w, h) {
    Style = s;
  }
  public double Area() {
    return Width * Height / 2;
  }
  public void ShowStyle() {
    Console.WriteLine("Треуго
      льник " + Style);
  }
}
```

```
class Shapes4 {
  static void Main() {
    Triangle t1 = new
      Triangle("равнобедренн
      ый", 4.0, 4.0);
    Triangle t2 = new
      Triangle("прямоугольный
      ", 8.0, 12.0);
    Console.WriteLine(«Объект
      t1: ");
    t1. ShowStyle() ;
    t1.ShowDim();
    Console.WriteLine("S="+
      t1.Area();
    Console.WriteLine();
    Console.WriteLine("Объект
      t2: ");
    t2. ShowStyle() ;
    t2.ShowDim();
    Console.WriteLine("S="+
      t2.Area());
  }
}
```

Вызов конструкторов базового класса

Когда в производном классе указывается ключевое слово **base**, вызывается конструктор из его непосредственного базового класса. Следовательно, ключевое слово **base** всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии классов.

Аргументы передаются базовому конструктору в качестве аргументов метода **base**(). Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

В иерархии классов конструкторы вызываются по порядку выведения классов: от базового к производному. Более того, этот порядок остается неизменным независимо от использования ключевого слова **base**. Если ключевое слово **base** не используется, то выполняется конструктор по умолчанию, т.е. конструктор без параметров.

Виртуальные методы и их переопределение

Виртуальным называется такой метод, который объявляется как **virtual** в базовом классе. Виртуальный метод отличается тем, что он может быть переопределен в одном или нескольких производных классах.

Если базовый класс содержит виртуальный метод и от него получены производные классы, то при обращении к разным типам объектов по ссылке на базовый класс выполняются разные варианты этого виртуального метода.

Метод объявляется как виртуальный в базовом классе с помощью ключевого слова **virtual**, указываемого перед его именем. Когда же виртуальный метод переопределяется в производном классе, то для этого используется модификатор **override**.

Виртуальные методы и их переопределение

```
class Base {
  public virtual void Who() {
    Console.WriteLine("Метод Who() в классе
                                Base");
  }
}
 class Derived1 : Base {
  // Переопределить метод Who()
  public override void Who() {
     Console.WriteLine("Метод Who() в классе
                          Derived1");
  }
}
class Derived2 : Base {
// Переопределить метод Who()
public override void Who() {
    Console.WriteLine("Метод Who() в классе
                          Derived2");
  }
}
```

```
Метод Who() в классе Base
Метод Who() в классе Derived1
Метод Who() в классе Derived2
```

```
class OverrideDemo {
   static void Main() {
     Base baseOb = new Base();
     Derived1 dOb1 = new Derived();
     Derived2 dOb2 = new Derived2();
     Base baseRef; // ссылка на базовый класс

     baseRef = baseOb;
     baseRef.Who();

     baseRef = dOb1;
     baseRef.Who() ;

     baseRef = dob2;
     baseRef.Who();
   }
}
```

Виртуальные методы и их переопределение

Переопределять виртуальный метод не обязательно. Если в производном классе не предоставляется собственный вариант виртуального метода, то используется его вариант из базового класса.

Если при наличии многоуровневой иерархии виртуальный метод не переопределяется в производном классе, то выполняется ближайший его вариант, обнаруживаемый вверх по иерархии.

Свойства также подлежат модификации ключевым словом **virtual** и переопределению ключевым словом **override**.

Переопределение методов - это еще один способ воплотить в C# главный принцип полиморфизма: один интерфейс - множество методов.

Переопределение метода

Пример применения виртуальных методов

```
class TwoDShape {
  double pri_width;
  double pri_height;
  public TwoDShape() {
    Width = Height = 0.0;
    name = "null";
  }
  public TwoDShape(double w, double h,
          string n) {
    Width = w;
    Height = h;
    name = n;
  }
  public TwoDShape(double x, string n) {
    Width = Height = x;
    name = n;
  }
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
    name = ob.name;
  }
  // Свойства ширины и высоты объекта,
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }
  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }
  public string name { get; set; }
  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
  public virtual double Area() {
    Console.WriteLine("Метод Area() должен быть
                       переопределен");
    return 0.0;
  }
}
```

Переопределение метода

Пример применения виртуальных методов

```
class Triangle : TwoDShape {
  string Style;
  public Triangle() {
    Style = "null";
  }
  public Triangle(string s, double w, double h) :
                base(w, h, "треугольник") {
    Style = s;
  }
  public Triangle(double x) : base(x, "треугольник")
                {
    Style = "равнобедренный";
  }
  public Triangle(Triangle ob) : base(ob) {
    Style = ob.Style;
  }
  public override double Area() {
    return Width * Height / 2;
  }
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}
```

```
class Rectangle : TwoDShape {
  public Rectangle(double w, double h) :
                base(w, h, "прямоугольник"){ }
  public Rectangle(double x) : base(x,
                "прямоугольник") { }
  public Rectangle(Rectangle ob) : base(ob) { }
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }
  public override double Area() {
    return Width * Height;
  }
}
```

Переопределение метода

Пример применения виртуальных методов

```
class DynShapes {
  static void Main() {
    TwoDShape[] shapes = new TwoDShape[5];
    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);
    shapes[4] = new TwoDShape(10, 20, "общая форма");
    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Объект - " + shapes[i].name);
      Console.WriteLine("Площадь равна " + shapes[i].Area());
      Console. WriteLine();
    }
    Console. ReadLine();
  }
}
```

Результат:

```
Объект - треугольник
Площадь равна 48

Объект - прямоугольник
Площадь равна 100

Объект - прямоугольник
Площадь равна 40

Объект - треугольник
Площадь равна 24.5

Объект - общая форма
Метод Area() должен быть переопределен
Площадь равна 0
```

Методы расширения

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает полезна, когда необходимо добавить в некоторый тип новый метод, не изменяя сам тип (класс или структуру).

Метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию:

this имя_типа название_параметра

Порядок создания метода расширения:

1) создать статический класс, который будет содержать требуемый метод;

2) объявить статический метод;

Метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, изначально определенный в типе.

Методы расширения действуют на уровне пространства имен. То есть, если добавить в проект другое пространство имен, то метод не будет применяться к строкам, и в этом случае надо будет подключить пространство имен метода через директиву *using*.

Методы расширения
Пример применения метода расширения

```
namespace  ExtensionMethods
{
public static class StringExtension
{
        public static int CharCount(this string str,
                                                char c)
   {
      int counter = 0;
      for (int i = 0; i<str.Length; i++)
      {
         if (str[i] == c)
            counter++;
      }
      return counter;
   }
}
}
```

```
namespace  ExtensionMethods
{
class Program
{
   static void Main()
   {
      string s = «Метод  расширения";
      char c = ‘e’;
      int a = s.CharCount(c);
      Console.WriteLine(a);

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

Абстрактные классы и методы

Абстрактный метод создается с помощью модификатора типа **abstract**. У абстрактного метода отсутствует тело, и поэтому он не реализуется в базовом классе. Это означает, что он должен быть переопределен в производном классе, поскольку его вариант из базового класса непригоден для использования.

Абстрактный метод автоматически становится виртуальным и не требует указания модификатора **virtual**. Совместное использование модификаторов **virtual** и **abstract** считается ошибкой.

Для определения абстрактного метода служит приведенная ниже общая форма.

abstract тип имя (список_параметров);

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный, и для этого перед его объявлением class указывается модификатор **abstract**. Поскольку реализация абстрактного класса не определяется полностью, то у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора new приведет к ошибке во время компиляции. Если в производном классе не будет определён метод, являющийся абстрактным в базовом, то во время компиляции возникнет ошибка.

В абстрактный класс допускается включать конкретные методы, которые в производном классе переопределять не требуется.

Абстрактные классы и методы

Пример применения абстрактного класса

```
abstract class TwoDShape {
  double pri_width;
  double pri_height;
  public TwoDShape() {
    Width = Height = 0.0;
    name = "null";
  }
  public TwoDShape(double w, double h,
          string n) {
    Width = w;
    Height = h;
    name = n;
  }
  public TwoDShape(double x, string n) {
    Width = Height = x;
    name = n;
  }
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
    name = ob.name;
  }
  // Свойства ширины и высоты объекта,
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }
  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }
  public string name { get; set; }
  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
  public abstract double Area()
}
```

Абстрактные классы и методы

Пример применения абстрактного класса

```
class Triangle : TwoDShape {
  string Style;
  public Triangle() {
    Style = "null";
  }
  public Triangle(string s, double w, double h) :
                base(w, h, "треугольник") {
    Style = s;
  }
  public Triangle(double x) : base(x, "треугольник")
                {
    Style = "равнобедренный";
  }
  public Triangle(Triangle ob) : base(ob) {
    Style = ob.Style;
  }
  public override double Area() {
    return Width * Height / 2;
  }
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}
```

```
class Rectangle : TwoDShape {
  public Rectangle(double w, double h) :
                base(w, h, "прямоугольник"){ }
  public Rectangle(double x) : base(x,
                "прямоугольник") { }
  public Rectangle(Rectangle ob) : base(ob) { }
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }
  public override double Area() {
    return Width * Height;
  }
}
```

Абстрактные классы и методы

Пример применения абстрактного класса

```
class DynShapes {
  static void Main() {
    TwoDShape[] shapes = new TwoDShape[4];
    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);
    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Объект - " + shapes[i].name);
      Console.WriteLine("Площадь равна " + shapes[i].Area());
      Console. WriteLine();
    }
    Console. ReadLine();
  }
}
```

Результат:

Объект - треугольник
Площадь равна 48

Объект - прямоугольник
Площадь равна 100

Объект - прямоугольник
Площадь равна 40

Объект - треугольник
Площадь равна 24.5

Предотвращение наследования с помощью ключевого слова ***sealed***

```
sealed class A {
// ...
}
class B : A { // ОШИБКА! Наследовать класс A нельзя
// ...
}
```

Предотвращение наследования с помощью ключевого слова ***sealed***

```
class B {
public virtual void MyMethod() { /* ... */ }
}
class D : B {
// Здесь герметизируется метод MyMethod() и  предотвращается его
//дальнейшее переопределение
sealed public override void MyMethod() { /* ... */ }
}
class X : D {
// Ошибка! Метод MyMethod() герметизирован!
public override void MyMethod() { /* ... */ }
}
```

Метод MyMethod () герметизирован в классе D, и поэтому не может быть переопределен в классе X.

Некоторые методы класса object

Метод	Назначение
public virtual bool Equals (object ob)	Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке ob
public virtual int GetHashCode()	Возвращает хеш-код, связанный с вызывающим объектом
public virtual string ToString()	Возвращает строку, которая описывает объект

Пример переопределения метода ToString()

```
class MyClass {
   static int count = 0;
   int id;
   public MyClass () {
      id = count;
      count++;
   }
   public override string ToString() {
      return "Объект #" + id + " типа
            MyClass";
   }
}
```

```
class Test {
   static void Main() {
      int a = 5;
      MyClass ob1 = new MyClass();
      MyClass ob2 = new MyClass();
      MyClass ob3 = new MyClass();
      Console.WriteLine(ob1);
      Console.WriteLine(ob2);
      Console.WriteLine(ob3);
      Console.WriteLine(a);
   }
}
```

При выполнении этого кода получается следующий результат:

```
Объект #0 типа MyClass          Override.MyClass
Объект #1 типа MyClass          Override.MyClass
Объект #2 типа MyClass          Override.MyClass
5                               5
```

Интерфейсы

Главный принцип полиморфизма: один интерфейс - множество методов.
Интерфейс представляет собой логическую конструкцию, описывающую функциональные возможности без конкретной их реализации. Один и тот же интерфейс может быть реализован в двух классах по-разному.

```
interface ISwitchable
  {
    void on();
    void off();
  }

  interface IType
  {
    string type {get;}
  }
```

```
class monitor : ISwitchable,IType
{
   public void on()
   {
      Console.WriteLine(type + " ON");
   }
   public void off()
   {
      Console.WriteLine(type + " OFF");
   }
   public string type
   {
      get {
         return "Monitor";
      }
   }
}
```

```
class printer : ISwitchable, IType
{
   public void on()
   {
      Console.WriteLine("Printer ON");
   }
   public void off()
   {
      Console.WriteLine("Printer OFF");
   }
}
```

Ошибка! Интерфейс IType не реализован

Интерфейсы

```
class Program
  {
     static void Main(string[] args)
     {
        monitor m = new monitor();
        printer pr = new printer();
        m.off();
        pr.off();
        m.on();
        ISwitchable sw1 = new monitor();
        sw1.off();
        ISwitchable sw2 = new printer();
        sw2.on();
//-------------------------------------------------
        List<ISwitchable> isw = new List<ISwitchable>();
        isw.Add(new monitor()  );
        isw.Add(new monitor());
        isw.Add(new printer());

        isw[0].off();
        isw[2].off() ;
        isw[1].on();
        isw[2].on();
}
```