Лекция 27

Организация пользовательских подпрограмм, методов класса

Классы

Класс представляет собой шаблон - тип данных, по которому определяется форма объекта.

Объект - это экземпляр класса, переменная типа класс.

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными.

Классы

Данные содержатся в переменных — **полях**-членах класса, а код — в подпрограммах – **функциях**-членах класса.

В C# предусмотрено несколько разновидностей полей и подпрограмм — функций-членов. Например, к полям относятся переменные экземпляра и статические переменные, а к функциям-членам — методы, конструкторы, деструкторы, индексаторы, события, операторы и свойства.

Классы

```
class имя_класса {
   // Объявление переменных экземпляра
   доступ тип переменная1;
    //...
    доступ тип переменнаяN;
   // Объявление методов
   доступ возращаемый_тип метод1 (параметры) {
       // тело метода
   }
    //...
    доступ возращаемый_тип методN (параметры) {
      // тело метода
   }
}
```

Пример 1

```
using System;
namespace ConsoleAppFunc{
class Building {
    public int Floors; // количество этажей
    public int Area; // общая площадь здания
    public int Occupants; // количество жильцов
}
class BuildingDemo {
     static void Main() {
        // создание объектов типа класса Building
        Building house = new Building();
        Building office = new Building();
        int areaPP; // площадь на одного человека
```

Пример 1

```
// Присвоить значения полям в объекте house
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
// Присвоить значения полям в объекте office
office.Occupants = 25;
office.Area = 4200;
office.Floors = 3;
// Вычислить площадь на одного человека в жилом доме
areaPP = house.Area / house.Occupants;
Console.WriteLine("Дом имеет:\n " +   house.Floors +
" этажа\n " + house.Occupants + " жильца\n " +
house.Area + " кв. футов общей площади, из них\n " +
areaPP + " приходится на одного человека");
```

Пример 1

```
// Вычислить площадь на одного человека в учреждении
 areaPP = office.Area / office.Occupants;
 Console.WriteLine("Учреждение имеет:\n " +
office.Floors + " этажа\n " + office.Occupants + "
работников\n " +  office.Area + " кв. футов общей
площади, из них\n " + areaPP + " приходится на
одного человека");
  }
 }
}  } // ***********************************
Building house = new Building();
// или
Building house; // объявить ссылку на объект
house = new Building(); // распределить память для объекта типа Building
```

Классы

Building house = **new** Building();

/* Объявление переменной **house** можно отделить от создания объекта, на который она ссылается */

Building house; // объявить ссылку на объект
// распределить память для объекта типа Building
house = **new** Building();

Building house1 = **new** Building();

Building house2 = house1;
/* переменная **house2** ссылается на тот же самый объект, что и переменная **house1** */

Методы

Переменные экземпляра (объекта) и **методы** (подпрограммы, функции) являются двумя основными составляющими классов.

Пользовательские функции (подпрограммы), определенные в классе, называются **методами**. В **C#** определить подпрограмму вне класса нельзя, поэтому все подпрограммы - это методы.

Методы

Метод — это *функциональный элемент* класса, который реализует вычисления или другие действия, выполняемые классом или экземпляром. Методы определяют поведение класса.

Метод описывается один раз и состоит из одного или нескольких операторов. В грамотно написанном коде C# каждый метод выполняет только одну функцию.

У каждого метода имеется свое **имя**, по которому он **вызывается сколько угодно раз**. После имени метода следуют круглые скобки, в которых могут быть указаны **переменные – формальные параметры**, получающие значение **аргументов – фактических параметров**, передаваемых методу при его вызове. Одна и та же функция может обрабатывать различные данные, переданные ей в качестве аргументов.

Методы

Общая форма определения метода:

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

Модификаторы

static делает метод доступным только через класс, в котором он определяется, но не через экземпляры объектов этого класса.

public (открытые), доступны любому методу любого класса.

protected (защищенные), доступны методам класса А и методам классов, производных от класса А.

internal (внутренние), доступны методам любого класса в сборке класса А**.**

private (закрытые), доступны только методам класса А – по умолчанию.

Модификаторы

virtual (виртуальный) — метод может переопределяться.

abstract (абстрактный) — метод должен обязательно переопределяться в не абстрактных производных классах (может использоваться только в абстрактных классах).

override (переопределенный) — метод переопределяет какой-то метод, определенный в базовом классе.

sealed (герметизированный) — в метод больше не могут вноситься изменения ни в каких производных классах, т.е. метод не может переопределяться в производных классах. Может использоваться вместе с ключевым словом **override**.

extern (внешний) — определение метода находится в каком-то другом месте.

Методы

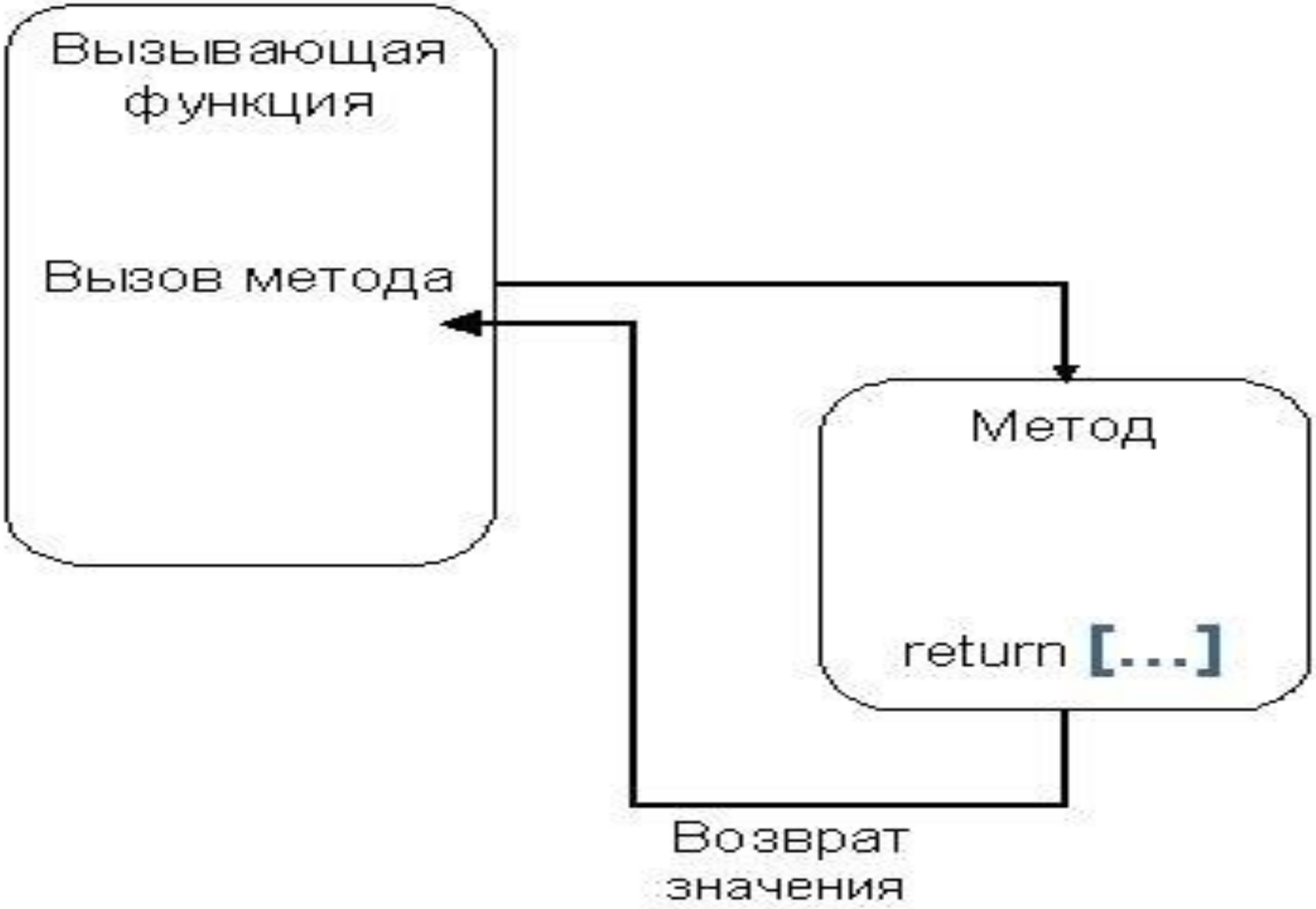
Определение метода в консольном приложении:

```
static <возвращаемый_тип> <имя_функции> (){
    return <возвращаемое_значение>;
}
```

В объектно-ориентированной среде программирования обработчик события имеет составное имя: имя компонента + имя события:

```
private void button1_Click(object sender,
  EventArgs e) {
        ...
}
```

Методы


Вызывающая функция
Вызов метода
Метод
return [...]
Возврат значения

Методы

```
public void MyMeth() {
    // ...
    if (done) return;
    // ...
}
```

```
int Sqr(int i) {
        return (i * i);
  }
```

Пример 2 (ООП)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsAppСвойства{
    public partial class Form1 : Form {
        public Form1()  {
            InitializeComponent();
        }
```

Пример 2 (ООП)

```
private void button1_Click(object sender, EventArgs e) {
    Building house = new Building();
    Building office = new Building();
    // Присвоить значения полям в объектах house и office
    if (textBox1.Text != "") house.Occupants =
Convert.ToInt32(textBox1.Text);
    if (textBox2.Text != "") house.Area =
Convert.ToInt32(textBox2.Text);
    if (textBox3.Text != "")  house.Floors =
Convert.ToInt32(textBox3.Text);
    if (textBox4.Text != "") office.Occupants =
Convert.ToInt32(textBox4.Text);
    if (textBox5.Text != "") office.Area =
Convert.ToInt32(textBox5.Text);
    if (textBox6.Text != "") office.Floors =
Convert.ToInt32(textBox6.Text);
```

Пример 2 (ООП)

```
richTextBox1.AppendText ("Дом имеет:\n " +
house.Floors.ToString() + " этажа\n " +
        house.Occupants.ToString() + " жильца\n " +
house.Area.ToString() +
        "кв. футов общей площади, из них");
    richTextBox1.AppendText (house.AreaPerPerson());
    richTextBox1.AppendText ("Учреждение имеет:\n "
+ office.Floors.ToString() + " этажа\n " +
        office.Occupants.ToString() + " работников\n " +
office.Area.ToString() +
        " кв. футов общей площади, из них");
    richTextBox1.AppendText (office.AreaPerPerson());
  }
}  // end class Form1
```

Пример 2 (ООП)

```
class Building {
    public int Floors;  // количество этажей
    public int Area;  // общая площадь здания
    public int Occupants;  // количество жильцов
    public string AreaPerPerson() {
      // Вывести площадь на одного человека
       string s = " " + (Area / Occupants).ToString() +"
приходится на одного человека\n ";
       return s;
    }
  } // end class Building
}
```

Методы

Если член класса объявляется как **static**, то он становится доступным до создания любых объектов своего класса и без ссылки на какой-нибудь объект. С помощью ключевого слова **static** можно объявлять как переменные, так и методы.

Для того чтобы воспользоваться членом типа **static** за пределами класса, достаточно указать **имя этого класса с оператором-точкой**. Но создавать объект для этого не нужно.

Методы

<u>Ограничения на применение методов типа **static:**</u>

• В методе типа **static** должна отсутствовать ссылка **this**, поскольку такой метод не выполняется относительно какого-либо объекта.

• В методе типа **static** допускается непосредственный вызов только других методов типа **static**, но не метода экземпляра из того самого же класса. Нестатический метод может быть вызван из статического метода только по ссылке на объект.

• Для метода типа **static** непосредственно доступными оказываются только другие данные типа **static**, определенные в его классе (метод, в частности, не может оперировать переменной экземпляра своего класса).

Пример 3

```
namespace ConsoleAppFunc{
  /* в пространстве имен нельзя размещать переменные и
  подпрограммы, но можно пользовательские типы данных:
  классы, структуры, ... */
   class Pr {
        public static int Val = 100;
        static public void Met(){ // необходим public
                                  // для видимости в другом классе
           Val = 200;
            Console.WriteLine("Met, Val = " + Val);
        }
       public string SMet(){      // не static
            return ("Stroka");
       }
   }
```

Пример 3

```
class Program {
    static void Method1()  {
        Console.WriteLine("Method1");
    }
    void Method2()  {            // не static
        Console.WriteLine("Method2");
    }
    static  int Sqr(int i) {
        return (i * i);
    }
     double Rez(int i) {
        return ((i*1.0) / 10);
    }
```

Пример 3

```
static void Main(string[] args) {
// вызов статических методов и переменных
    Pr.Val = 1;   // доступ к переменной через класс
    Pr.Met();  // доступ к методу через класс, Val = 200
    Method1();   // вызов метода отдельным оператором
    Console.Write("Введите целое число - ");
    int i = Convert.ToInt32(Console.ReadLine());
    int a = Sqr(i);
    Console.WriteLine(i + " в квадрате равно " + a);
// вызов методов с помощью ссылки на объект класса
    Program p = new Program();
    // т. к. Method2 не static,
    // то нужна p - ссылка на объект класса Program
```

Пример 3

```
        p.Method2();
        Console.Write("Введите целое число - ");
        int j = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine(j + " / 10 = " + p.Rez(j));
        Pr pp = new Pr();  // pp - ссылка на объект класса Pr
        string s = pp.SMet();
        Console.WriteLine("Результат метода SMet класса
  Pr:   " + s);
        Console.ReadKey();
    }
  }
}
```

Параметры методов

Параметры используются для обмена информацией с методом.

Параметры, описываемые в заголовке метода – **формальные параметры**, определяют множество значений **аргументов**, которые можно передавать в метод.

Для каждого параметра должны задаваться его **тип** и **имя**.

```
static  double Sum(int i, double j) {
      return (i + j);
}
```

Параметры методов

Параметры, которые передаются в метод вместо формальных параметров при вызове этого метода, называются **аргументами** или **фактическими параметрами.**

Правила соответствия формальных и фактических параметров: **порядок следования, типы** и **количество** формальных и фактических параметров должны **совпадать**.

Параметры методов

Описание объекта: SomeObj obj = new SomeObj();

Вызов метода: obj.P(a, b, c);

Заголовок метода P: public void P(double x, int y, double z);

```
int y = 3;
double x = 0.9;
double S = Sum(y, x);
Console.WriteLine(S);
```

Параметры методов

Существуют два способа передачи параметров: **по значению** и **по ссылке**.

При передаче по значению (**по умолчанию**) формальные параметры метода получают копии значений аргументов, и операторы метода работают с этими копиями. Доступа к исходным значениям аргументов у метода нет, а, следовательно, изменения, вносимые в параметр метода, не оказывают никакого влияния на аргумент, используемый для вызова.

Параметры методов

При передаче по ссылке (**по адресу**) метод получает копии **адресов аргументов**, он осуществляет **доступ** к ячейкам памяти по этим адресам и **может изменять исходные значения аргументов**, модифицируя параметры.

В **C#** предусмотрено четыре типа параметров:

- параметры-значения;
- параметры-ссылки — описываются с помощью ключевого слова **ref**;
- выходные параметры — описываются с помощью ключевого слова **out**;
- параметры-массивы — описываются с помощью ключевого слова **params**.

Пример 4

```
namespace ConsoleAppFunc{
  class Test {
     public int a, b;
     public Test() { }
     public Test(int i, int j) {
          a = i;
          b = j;
     }
```

Пример 4

```
/* Этот метод не оказывает никакого влияния на
   аргументы, используемые для его вызова. */
  public void NoChange(int i, int j){ //параметры-значения
      i = i + j;
      j = -j;
  }
```

```
  /* Передать объект. Теперь переменные ob.a и ob.b из объекта,
  используемого в вызове метода, будут изменены. */
  public void Change(Test ob) { // ссылочный параметр
      ob.a = ob.a + ob.b;
      ob.b = -ob.b;
  }
}
```

Пример 4

```
class CallByValue {
  static void Main() {
    Test ob = new Test();
    int a = 15, b = 20;
    Console.WriteLine ("a и b до вызова: " + a + "  " + b);
    ob.NoChange(a, b);
    Console.WriteLine("a и b после вызова: " + a + "  " + b);
    Test ob1 = new Test(15, 20);
    Console.WriteLine("ob1.a и ob1.b до вызова: " + ob1.a +"  "+
  ob1.b);   // 15   20
    ob1.Change(ob1);
    Console.WriteLine("ob1.a и ob1.b после вызова: " + ob1.a + "  "
  + ob1.b);    //  35    -20
  }
}
}
```

Параметры методов

Модификатор параметра **ref** принудительно организует вызов по ссылке, а не по значению. Этот модификатор указывается как при объявлении, так и при вызове метода.

Пример 5

```
namespace ConsoleAppFunc{
  class RefTest {
     public void Sqr(ref int i, int j) {
      // Метод изменяет свой аргумент
          i = i * i;
          j++;
     }
  }
```

Пример 5

```
class RefDemo {
   static void Main() {
    RefTest ob = new RefTest ();
    int a = 10, b = 0;
    Console.WriteLine("a до вызова: " + a);   // 10
    Console.WriteLine("b до вызова: " + b);   // 0
    ob.Sqr(ref a, b); // применение модификатора ref
    Console.WriteLine("a после вызова: " + a); // 100
    Console.WriteLine("b после вызова: " + b);    // 0
   }
 }
}
```

Параметры методов

Модификатор параметра **out** подобен модификатору **ref**, за одним исключением: он служит только для передачи значения за пределы метода. Поэтому переменной, используемой в качестве параметра **out**, не нужно (да и бесполезно) присваивать какое-то значение. Более того, в методе параметр **out** считается неинициализированным, т.е. предполагается, что у него отсутствует первоначальное значение. Это означает, что значение должно быть присвоено данному параметру в методе до его завершения.

Пример 6

```
namespace ConsoleAppFunc{
  class Decompose {
    public int GetParts(double n, out double frac){
        int whole;
        whole = (int) n;
        frac = n - whole; // передать дробную
        // часть числа через параметр frac
        return whole;    // возвратить целую
                         // часть числа
    }
  }
```

Пример 6

```
class UseOut {
  static void Main() {
      Decompose ob = new Decompose();
       int i;
      double f, a;
      Console.Write("Введите вещественное число - ");
      a = Convert.ToDouble(Console.ReadLine());
      i = ob.GetParts(a, out f);
  Console.WriteLine("Для вещественного числа " + a + ":");
  Console.WriteLine("Целая часть числа равна " + i); // 10
  Console.WriteLine("Дробная часть числа равна " + f);//0.125
   }
 }
}
```

Параметры методов

Применение модификаторов **ref** и **out** не ограничивается только передачей значений обычных типов. С их помощью можно также передавать **ссылки на объекты**. Если модификатор **ref** или **out** указывает на ссылку, то сама ссылка передается по ссылке. Это позволяет изменить в методе объект, на который указывает ссылка.

Пример 7

```
namespace ConsoleAppFunc{
  class RefSwap {
    int a, b;
    public RefSwap(int i, int j) {
      a = i;
      b = j;
    }
    public void Show() {
      Console.WriteLine ("a: {0}, b: {1}", a, b);
   }
```

Пример 7

```
// Этот метод изменяет свои аргументы
  public void Swap(ref RefSwap ob1, ref
RefSwap ob2) {
      RefSwap t;
      t = ob1;
      ob1 = ob2;
      ob2 = t;
  }
}
```

Пример 7

```
class RefSwapDemo {
  static void Main() {
    RefSwap x = new RefSwap(1, 2);
    RefSwap y = new RefSwap(3, 4);
    Console.Write("x до вызова: ");
    x.Show();  // x до вызова: a: 1, b: 2
    Console.Write("y до вызова: ");
    y. Show ();  // y до вызова: a: 3, b: 4
    Console.WriteLine ();
```

Пример 7

```
// Смена объектов, на которые
// ссылаются аргументы x и y.
 x.Swap(ref x, ref y);
 Console.Write("x после вызова: ");
 x.Show();  // x после вызова: a: 3, b: 4
 Console.Write("y после вызова: ");
 y.Show();   // y после вызова: a: 1, b: 2
  }
 }
}
```

Параметры методов

Ссылка может использоваться как **результат функции.** Для возвращения из функции ссылки в сигнатуре функции перед возвращаемым типом, а также после оператора **return** следует указать ключевое слово **ref**.

Пример 8

```
static void Main(string[] args){
    int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
   try {
      // найти число 4 в массиве
      ref int numberRef = ref Find(4, numbers);
      numberRef = 9; // заменить 4 на 9
      Console.WriteLine(numbers[3]); // 9
      Console.Read();
   }
   catch (IndexOutOfRangeException e) {
      Console.WriteLine(e.Message);
   }
}
```

Пример 8

```
static ref int Find(int number, int[] numbers){
  for (int i = 0; i < numbers.Length; i++) {
    if (numbers[i] == number) {
       return ref numbers[i];
   // возвращается ссылка на адрес,
   // а не само значение
    }
  }  throw new IndexOutOfRangeException("число
  не найдено");
}
```

Параметры методов

C# позволяет использовать **необязательные параметры**. **Для таких параметров необходимо объявить значение по умолчанию**. После необязательных параметров все последующие параметры также должны быть необязательными.

При вызове метода значения для параметров передаются в порядке объявления этих параметров в методе. Но можно нарушить подобный порядок, используя **именованные параметры**. **Именованы** должны быть **все параметры**.

Пример 9

```
static int OP(int x, int y, int z=5, int s=4){
    return x + y + z + s;
}
...
static void Main(string[] args){
    Console.WriteLine (OP(2, 3));        // 14
    Console.WriteLine (OP(2,3,10));    // 19
    // использование именованных параметров
   Console.WriteLine (OP(x:2, y:3));   // 14
    // необязательный параметр z использует
    // значение по умолчанию
   Console.WriteLine (OP(s:10, y:2, x:3)); // 20
}
```

Параметры методов

Язык **C#** позволяет указывать один (и только один последний в списке параметров) специальный параметр для функции **- массив параметров**.

Используя ключевое слово **params**, можно передавать в метод неопределенное количество параметров. Этот способ передачи параметров надо отличать от передачи **массива в качестве параметра**.

Пример 10

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

Пример 10

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

Пример 10

```
static void Main(string[] args){
    Addition(1, 2, 3, 4, 5);
    int[] array = new int[] { 1, 2, 3, 4 };
     Addition(array);
     Addition();
    AdditionMas(array, 2);
    Console.ReadLine();
}
```

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

Область видимости (контекст) переменных

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

Существуют различные контексты:

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

Пример 11

```
class Program{ // начало контекста класса
   static int a = 9; // переменная уровня класса - глорбальная
   static void Main(string[] args) {
      int b = a - 1; // локальная переменная
         ...
        { // начало контекста блока кода
          int c = b - 1; // переменная уровня блока кода
         }  // конец блока кода, переменная c уничтожается
       //  Console.WriteLine(c);  // ошибка!
       //  Console.WriteLine(d);  // ошибка!
        Display();
        Console.Read();
   } // конец Main, переменная b уничтожается
```

Пример 11

```
static void Display(){ // начало метода Display
    int a = 5;   // локальная переменная
    int d = Program.a + 1;  // использование
                            // глобальной переменной
    Console.WriteLine(d);
    d = a + 1; //использование локальной переменной
    Console.WriteLine(d);
  } // конец контекста метода Display,
    // переменная d уничтожается
} // конец контекста класса, переменная a уничтожается
```

Организация закрытого и открытого доступа

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.

Организация закрытого и открытого доступа

- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.

Организация закрытого и открытого доступа

- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.

Организация закрытого и открытого доступа

- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Пример 12

```
namespace ConsoleAppFunc{
 class MyClass {
    private int alpha; // закрытый доступ,
                       // указываемый явно
    int beta; // закрытый доступ по умолчанию
    public int gamma; // открытый доступ
    public void SetAlpha(int a) { // открытый доступ
    alpha = a; // Член класса может иметь доступ
               // к закрытому члену этого же класса
    public int GetAlpha() { // открытый доступ
     return alpha;
 }
```

Пример 12

```
public void SetBeta(int a) {// открытый доступ
    beta = a;
 }
 public int GetBeta() { // открытый доступ
    return beta;
 }
} // end class
```

Пример 12

```
class AccessDemo {
  static void Main() {
    MyClass ob = new MyClass();
    // Доступ к членам alpha и beta данного класса
    // разрешен только посредством его методов.
    ob.SetAlpha(-99);
    ob.SetBeta(9) ;
    Console.WriteLine("ob.alpha равно " +
ob.GetAlpha());
    Console.WriteLine("ob.beta равно " +
ob.GetBeta());
```

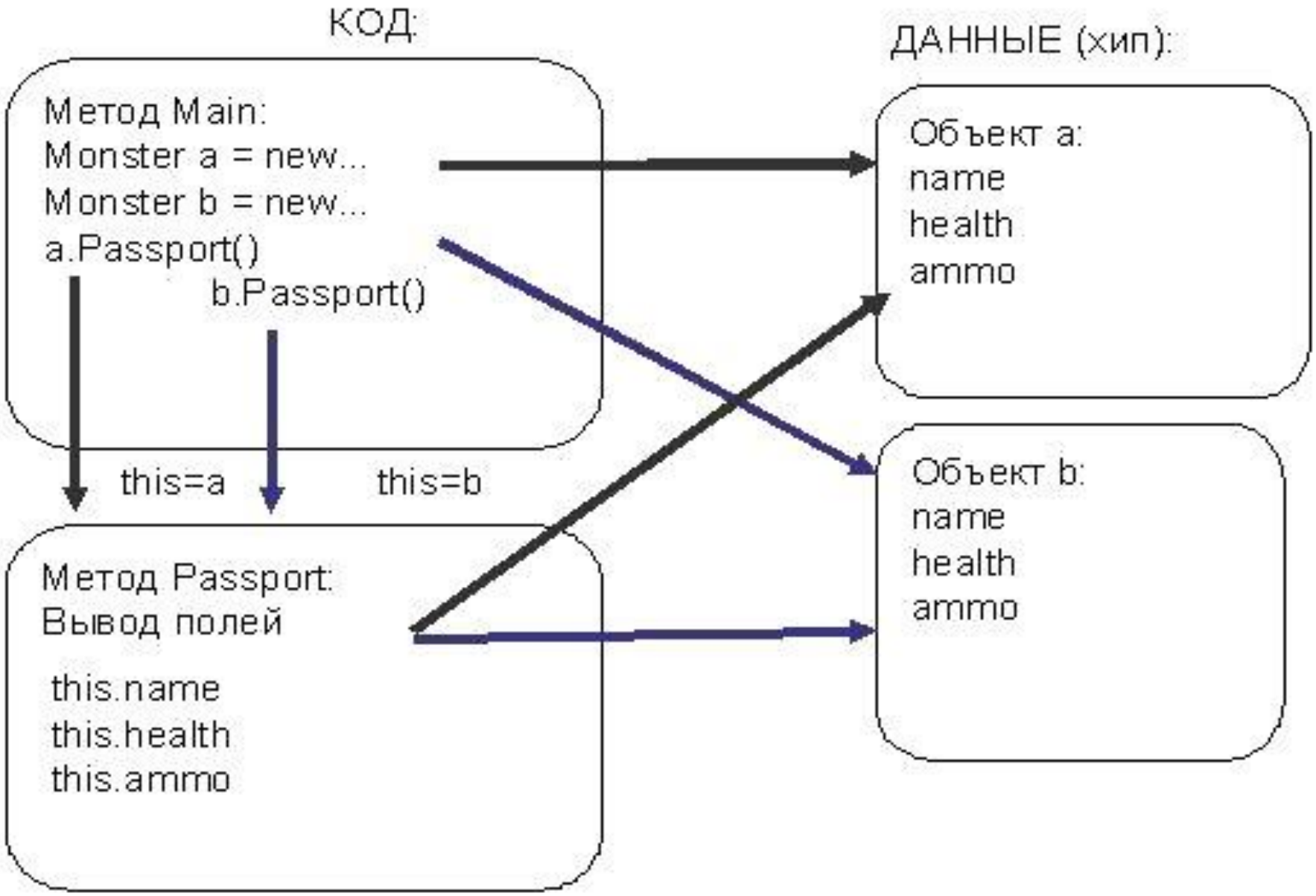
Пример 12

```
    // Следующие виды доступа к членам alpha
    // и beta данного класса не разрешаются.
    // ob.alpha = 10; // Ошибка! alpha - закрытый член!
    // ob.beta =9; // Ошибка! beta - закрытый член!
    // Член gamma данного класса доступен
    // непосредственно, поскольку он
    // является открытым.
        ob.gamma = 99;
   }
  }  // end class
}   // end namespace
```

Конструкторы

Каждый **объект** (переменная типа класс) содержит свой экземпляр полей класса. Методы находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр **this**, в котором хранится **ссылка** на вызвавший функцию экземпляр.

Конструкторы



Конструкторы

Конструктор - метод класса - предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции **new**:

- Имя конструктора совпадает с именем класса.
- Конструктор **не возвращает значение**, даже типа **void**.
- Класс может иметь **несколько конструкторов с разными параметрами** для разных видов инициализации.
- Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, автоматически вызывается **конструктор базового класса без параметров (конструктор по умолчанию)**, который полям **значимых типов** присваивает нуль, полям ссылочных типов — значение **null**.

Конструкторы

Общая форма конструктора:

```
[доступ] имя_класса([список_параметров])
{
    // тело конструктора
}
```

Пример 13

```
namespace ConsoleApplication1{
  class Demo {
    public Demo( int a, double y ) {
      // конструктор с параметрами
      this.a = a;
      this.y = y;
    }
    public double Gety(){ // метод получения поля
        return y;
    }
    int a;   // закрытые поля класса
    double y;
  }
```

Пример 13

```
class Class1{
  static void Main() {
    // вызов конструктора
     Demo a = new Demo( 300, 0.002 );
     Console.WriteLine( a.Gety() ); // результат: 0,002
    // вызов конструктора
     Demo b = new Demo( 1, 5.71 );
     Console.WriteLine( b.Gety() ); // результат: 5,71
  }
}
}
```

Пример 14

```
class Demo {
    public Demo() {} // конструктор 1 - по умолчанию
    public Demo(int a) {this.a = a; }  // конструктор 2
    public Demo(int a, double y) : this(a)  {
                    // конструктор 3 - вызов конструктора 2
      this.y = y;
   }
   public Demo(Demo ob) {
            // конструктор 4 - копирующий конструктор
     a = ob.a;       y = ob.y;
   }
   int a;
   double y;   ...
}
```

Пример 14

```
static void Main() {
   Demo a1 = new Demo();     // вызов конструктора 1
   Demo a2 = new Demo(300);     // вызов конструктора 2
   Demo a3 = new Demo(300, 0.002); // вызов конструктора 3
   Demo a4 = new Demo(a3);     // вызов конструктора 4
   ...
}
```

Конструкция, находящаяся после двоеточия, называется **инициализатором**.

Все классы в **C#** имеют общего предка — класс **object**. Конструктор любого класса, если не указан **инициализатор**, автоматически вызывает конструктор своего предка.

Конструкторы

В **C#** существует возможность описывать **статический класс**, то есть класс с модификатором **static**. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с модификатором **static** (*константы* и *вложенные типы* классифицируются как статические элементы автоматически).

Пример 15

```
namespace ConsoleApplication1{
  static class D{
    static int a = 200;
    static double b = 0.002;
    public static void Print (){
      Console.WriteLine( "a = " + a );
      Console.WriteLine( "b = " + b );
   }
 }
```

Пример 15

```
class Class1{
    static void Main(){
        D.Print();
    }
}
}
```

Пример 16

```
class MathLib{
    public const double PI=3.141;
    public const double E = 2.81;
    public const double K;
    public MathLib()    {  // конструктор
      K = 2.5; // ошибка - константа должна быть определена до компиляции
    }
}
class Program{
    static void Main(string[] args)    {
        MathLib.E=3.8; // константу нельзя установить несколько раз
    }
}
```

Пример 17

```
class MathLib{
   public readonly double K = 23; // инициализация
   public MathLib(double _k)  {
      K = _k; // поле для чтения может быть
     // инициализировано или изменено
     // в конструкторе после компиляции
   }
   public void ChangeField()  {
      // K = 34; // так нельзя
   }
}
```

Пример 17

```
class Program{
   static void Main(string[] args)  {
      MathLib mathLib = new MathLib(3.8);
      Console.WriteLine(mathLib.K); // 3.8
     // mathLib.K = 7.6; // поле для чтения нельзя
                                 // установить вне своего класса
      Console.ReadLine();
   }
}
```

Деструкторы

Система "сборки мусора" в **C#** освобождает память от лишних объектов автоматически, действуя незаметно и без всякого вмешательства со стороны программиста. "Сборка мусора" происходит лишь время от времени по ходу выполнения программы, нельзя заранее знать или предположить, когда именно произойдет "сборка мусора".

В языке **C#** имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Такой метод называется **деструктором** .

Деструкторы

Общая форма деструктора:

~имя_класса() {

// код деструктора

}

В деструкторе можно указать те действия, которые следует выполнить перед тем, как уничтожать объект. Деструктор вызывается непосредственно перед "сборкой мусора".

Пример 18

```
class Destruct {
    public int x;
    public Destruct(int i) { x = i; }
        // Вызывается при утилизации объекта.
      ~Destruct () {
            Console.WriteLine("Уничтожить " + x);
      }
      // Создает объект и тут же уничтожает его.
    public void Generator(int i) {
      Destruct o = new Destruct (i);
    }
}
```

Пример 18

```
class DestructDemo {
    static void Main() {
      int count;
      Destruct ob = new Destruct (10);
  /* Можно создать большое число объектов,
  чтобы в какой-то момент произошла "сборка
  мусора" */
     for (count=l; count < 100000; count++)
        ob.Generator(count);
      Console.WriteLine("Готово!");
    }
}
```

Пример 19

```
// Масив объектов
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

Пример 19

```
namespace ConAVar0{
[Serializable]
   class base0 {
    private double x, y;  // координаты точки
      private char name;   // название точки
      public virtual double space(base0 d) {
         return Math.Sqrt(d.x * d.x + d.y * d.y); }
      public void inx(double xx) { x = xx; }
      public void iny(double yy) { y = yy; }
      public void inn(char nn) { name = nn; }
      public double outx() { return x; }
      public double outy() { return y; }
      public char outn() { return name; }
   }
```

Пример 19

```
class child :  base0 {
    private  base0 []a = new base0 [5];
    private int m;  // количество точек, записанных в файл
                    // - оптимизация: int m = 0;
    private int n;  // количество элементов массива a
                    // - оптимизация: int n = 5;
    public int infile() {
      n = 5;   // - оптимизация
               // массив для сериализации
     a[0] = new base0();  a[0].inx(2.1);   a[0].iny(3.5);  a[0].inn('B');
     a[1] = new base0();  a[1].inx(-1.0);  a[1].iny(2.4);  a[1].inn('A');
     a[2] = new base0();  a[2].inx(2.7);   a[2].iny(1.5);  a[2].inn('D');
     a[3] = new base0();  a[3].inx(4.3);   a[3].iny(-3.5); a[3].inn('C');
     a[4] = new base0();  a[4].inx(-5.0);  a[4].iny(-2.8); a[4].inn('E');
```

Пример 19

```
try   {
      BinaryFormatter formatter = new
BinaryFormatter();
       using (FileStream fout = new
FileStream("in.txt", FileMode.OpenOrCreate)){
          formatter.Serialize(fout, a);
                // сериализация всего массива
      }
      using (FileStream fin = new
FileStream("in.txt", FileMode.OpenOrCreate))   {
    a = (base0[])formatter.Deserialize(fin);
                // десериализация
```

Пример 19

```
        foreach (base0 p in a)
            Console.WriteLine($"{p.outn()}
({p.outx()}; {p.outy()})");
        }
    } catch (IOException e)  {
        Console.WriteLine("Сгенерировано
исключение ввода-вывода!");
        Console.WriteLine(e.ToString());
        return 1;
    }
    return 0;
}
```

Пример 19

```
public void sort()   {
    /*Реализация алгоритма линейной
  сортировки по координате Y в порядке
  возрастания*/
    base0 X;
    n = 5;  // - оптимизация
    for (int i = 0; i < n; i++)
        for (int c = i + 1; c < n; c++)
            if (a[i].outy() > a[c].outy())    {
                X = a[i];        a[i] = a[c];        a[c] = X;
            }
    }
```

Пример 19

```
/* функция перегружает метод базового класса
 для определения точки, находящейся в 1
 четверти координатной оси */
    public override double space(base0 d)   {
      if (d.outx() > 0 && d.outy() > 0)
        return 1;
      else return 0;
    }
```

Пример 19

```
/*  функция выводит в файл "rezult.txt" характеристики точек,
  размещенных в первой четверти координатной оси */
    public int outfile()   {
      m = 0;   // - оптимизация
      n = 5;   // - оптимизация
      try   {
        BinaryFormatter formatter = new
  BinaryFormatter();
        using (FileStream fout = new
  FileStream("rezult.txt", FileMode.OpenOrCreate))
  { /* Записать  на диск только точки,
  размещенные в первой четверти *//
```

Пример 19

```
            for (int i = 0; i < n; i++)   {
                if (space(a[i]) == 1)    {
                    formatter.Serialize(fout, a[i]);
                     m++;
                }
            }
            fout.Close();
        }
    }
    catch (IOException e)   {
      Console.WriteLine("Исключение ввода-вывода!");
      Console.WriteLine(e.ToString());     return 1;
  }
  return 0;
}
```

Пример 19

```
public int inrezfile()  {
   int k;   // - оптимизация int k = 0;
   k = 0;   // - оптимизация
   Console.WriteLine("В файле \"rezult.txt\" записаны
следующие точки:");
   try    {
      BinaryFormatter formatter = new BinaryFormatter();
     using (FileStream fin = new FileStream("rezult.txt",
FileMode.OpenOrCreate))  {
        while (k < m) {
           base0 b = (base0)formatter.Deserialize(fin);
                                          // десериализация
```

Пример 19

```
            Console.WriteLine($"Точка {b.outn()} с
координатами {b.outx()} и {b.outy()}");
            k++;
          } // end while
           fin.Close();
        } // end using
      } catch (IOException e) {
         Console.WriteLine("Исключение ввода-вывода!");
         Console.WriteLine(e.ToString());    return 1;
      }
       return 0;
    } // end inrezfile()
} // end class child :  base0
```

Пример 19

```
class Program   {
    static void Main(string[] args)   {
        child ob = new child();
        ob.infile();
        ob.sort();
        ob.outfile();
        ob.inrezfile();
        Console.ReadKey();
    }
  }
}
```

Контрольные вопросы

1. Перечислите и опишите элементы класса в C#.
2. Опишите способы передачи параметров в методы.
3. Для чего в классе может потребоваться несколько конструкторов?
4. Как можно вызвать один конструктор из другого? Зачем это нужно?
5. Что такое this? Что в нем хранится, как он используется?
6. Что такое деструктор? Гарантирует ли среда его выполнение?