Лекция 21

Организация пользовательских подпрограмм, методов класса

Подпрограммы

Подпрограмма (функция) представляет собой законченный фрагмент кода, к которому можно обратиться **по имени**. Она описывается один раз, а вызываться может столько раз, сколько необходимо. Одна и та же функция может обрабатывать различные данные, переданные ей в качестве аргументов.

Подпрограммы

Преимущества подпрограмм (функций):

- содержат многократно используемый код.
- делают код более читабельным, поскольку могут применяться для группирования связанных между собой фрагментов кода.
- могут применяться для создания универсального кода, что позволяет им выполнять одни и те же операции над варьирующимися данными (обобщенные функции).

Подпрограммы

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

[модификаторы] тип_возвращаемого_значения название_функции (метода) ([параметры])

```
{
    // 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 A и методам классов, производных от класса A.

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

private (закрытые), доступны только методам класса A.

Модификаторы

virtual (виртуальный) — метод может переопределяться.

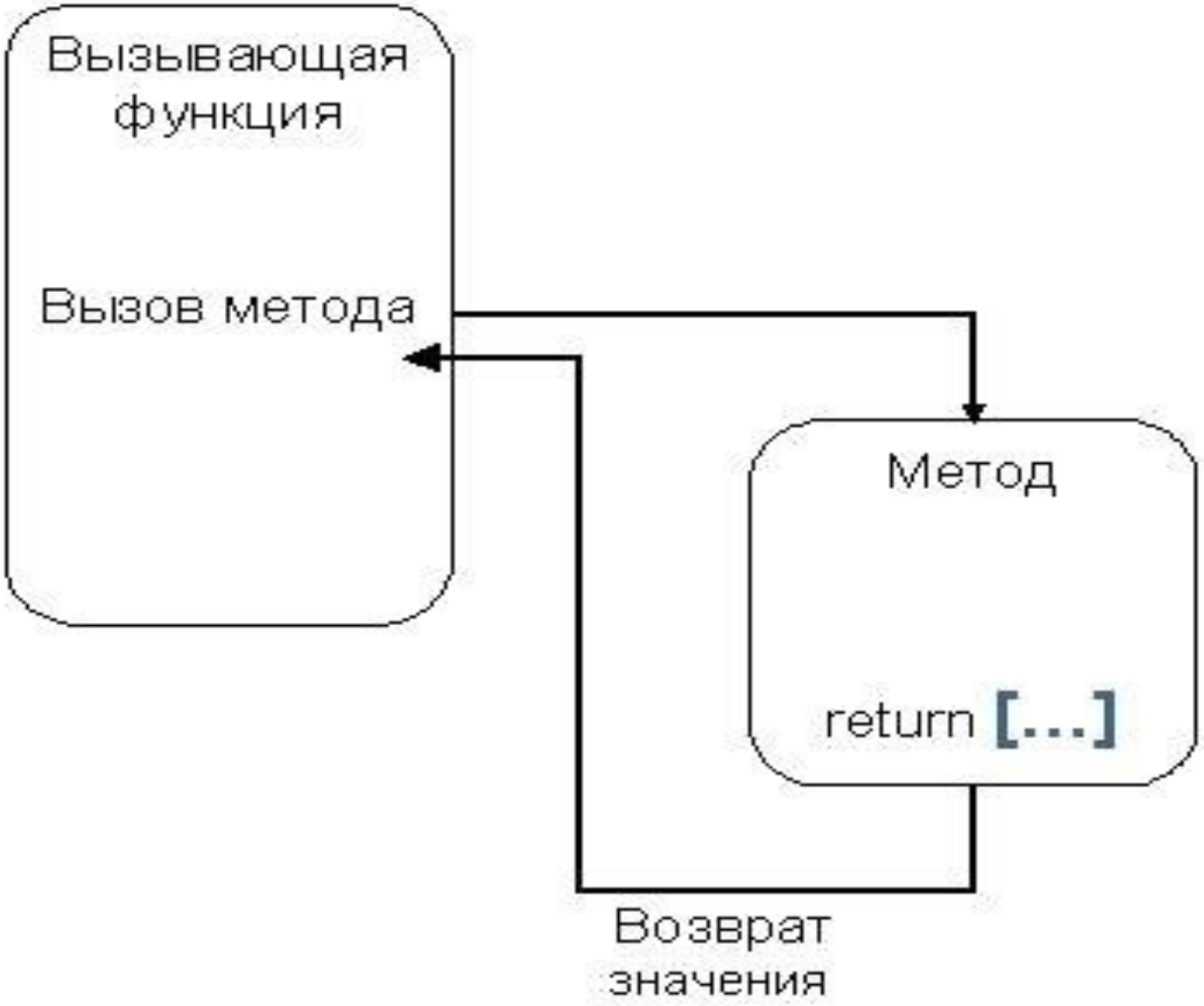
abstract (абстрактный) — метод должен обязательно переопределяться в не абстрактных производных классах (может использоваться только в абстрактных классах).

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


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

Методы

Определение метода в консольном приложении:

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

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

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

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

Ограничения на применение методов типа **static:**

• 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 1

```
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 1

```
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 1

```
    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 1

```
            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

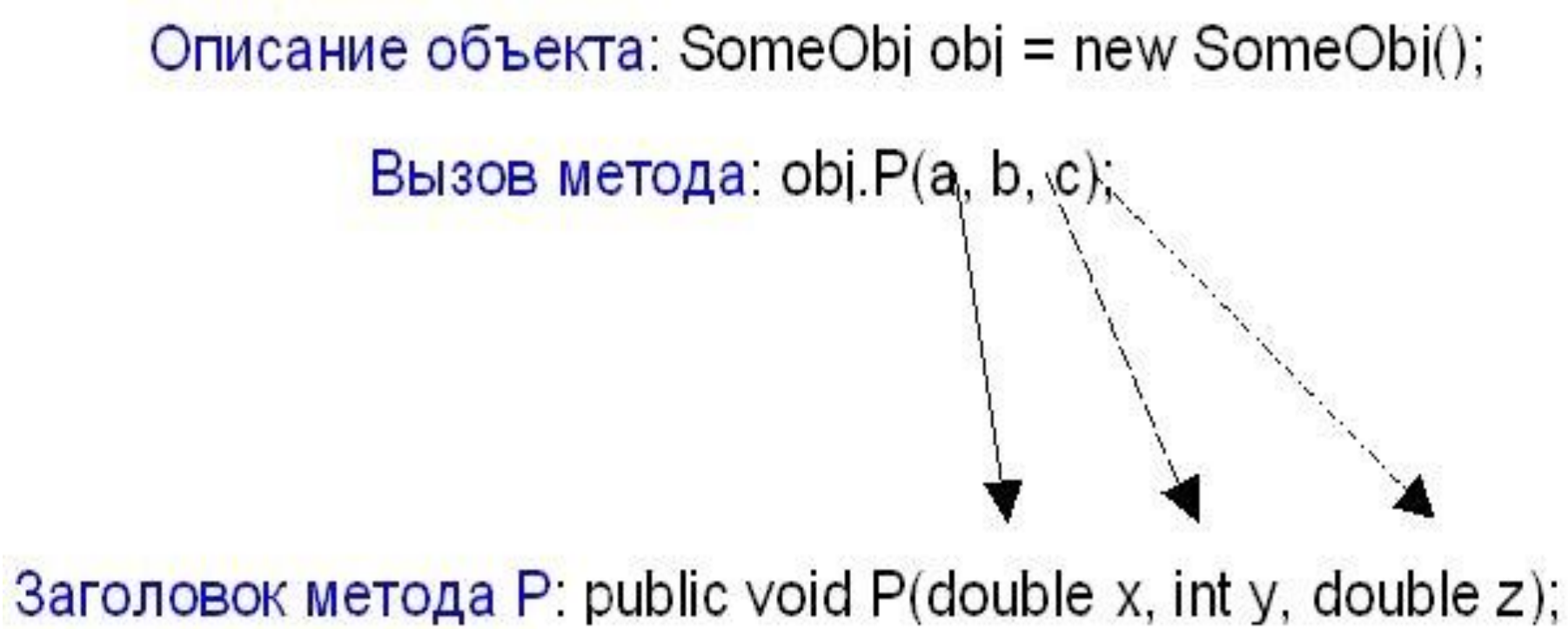
Параметры используются для обмена информацией с методом. **Параметры**, описываемые в заголовке метода, определяют множество значений **аргументов**, которые можно передавать в метод. Для каждого параметра должны задаваться его **тип** и **имя**.

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

Параметры методов

Правила соответствия формальных и фактических параметров: главное требование при передаче параметров состоит в том, что аргументы при вызове метода должны записываться в том же порядке, что и в заголовке метода, и должно существовать неявное преобразование типа каждого аргумента к типу соответствующего параметра. Количество аргументов должно соответствовать количеству параметров.

Параметры методов




```
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 2

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

Пример 2

```
/* 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 2

```
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 3

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

Пример 3

```
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 4

```
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 4

```
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 5

```
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 5

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

Пример 5

```
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 5

```
    // 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 6

```
static void Main(string[] args){
    int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
    // 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();
}
```

Пример 6

```
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**. Псле 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 пераметры**.

Пример 7

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

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 8

```
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 8

```
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 8

```
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 9

```
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 9

```
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 10

```
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 10

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

Пример 10

```
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 10

```
    // 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;
    }
}
```

Свойства

[атрибуты] [спецификаторы] тип
имя_свойства{
[**get** код_доступа]
[**set** код_доступа]
}

public, **private** и т.д.
virtual, **override** и **abstract**

Свойства

В блоках **get** должно обязательно присутствовать возвращаемое значение типа свойства, т. е. он должен содержать оператор **return**.

Функция **set** присваивает значение закрытому полю. Здесь можно применять ключевое слово **value,** которое содержит устанавливаемое значение.

Можно опускать тот или иной блок и тем самым создавать свойства, доступные только для записи или только для чтения (в частности, **пропуск** блока **get** позволяет обеспечивать доступ только для записи, а пропуск блока **set** — доступ только для чтения).

Пример 11

```
public class MyClass {
    public readonly string Name;
    private int intVal;
    public int Val {
        get {  return intVal;  }
       set {
           if (value > = 0 && value <= 10) intVal = value;
           else  throw (new
ArgumentOutOfRangeException("Val = ", value, ",  Val
может присваиваться только значение в диапазоне от
0 до 10."));
       }
  }
```

Пример 11

```
    public override string ToString() {
        return "Name:  " + Name + "\nVal:  " + Val;
    }
    private MyClass() : this("Default Name") {  }
        // конструктор
    public MyClass(string newName)  {
        Name = newName;
        intVal = 0;
    }
}
```

Пример 11

```
public class Program {
   static void Main(string [ ] args) {
      Console.WriteLine("Создание объекта myObj...");
      MyClass myObj = new MyClass("My Object");
      Console.WriteLine("Объект myObj создан.");
      for (int i = -1; i <= 0; i++)        {
         try {
            Console.WriteLine("\n Попытка присвоить
   myObj.Val значение {0}...", i);
            myObj.Val = i;
            Console.WriteLine("Значение {0} присвоено
   myObj.Val.", myObj.Val);
         }
```

Пример 11

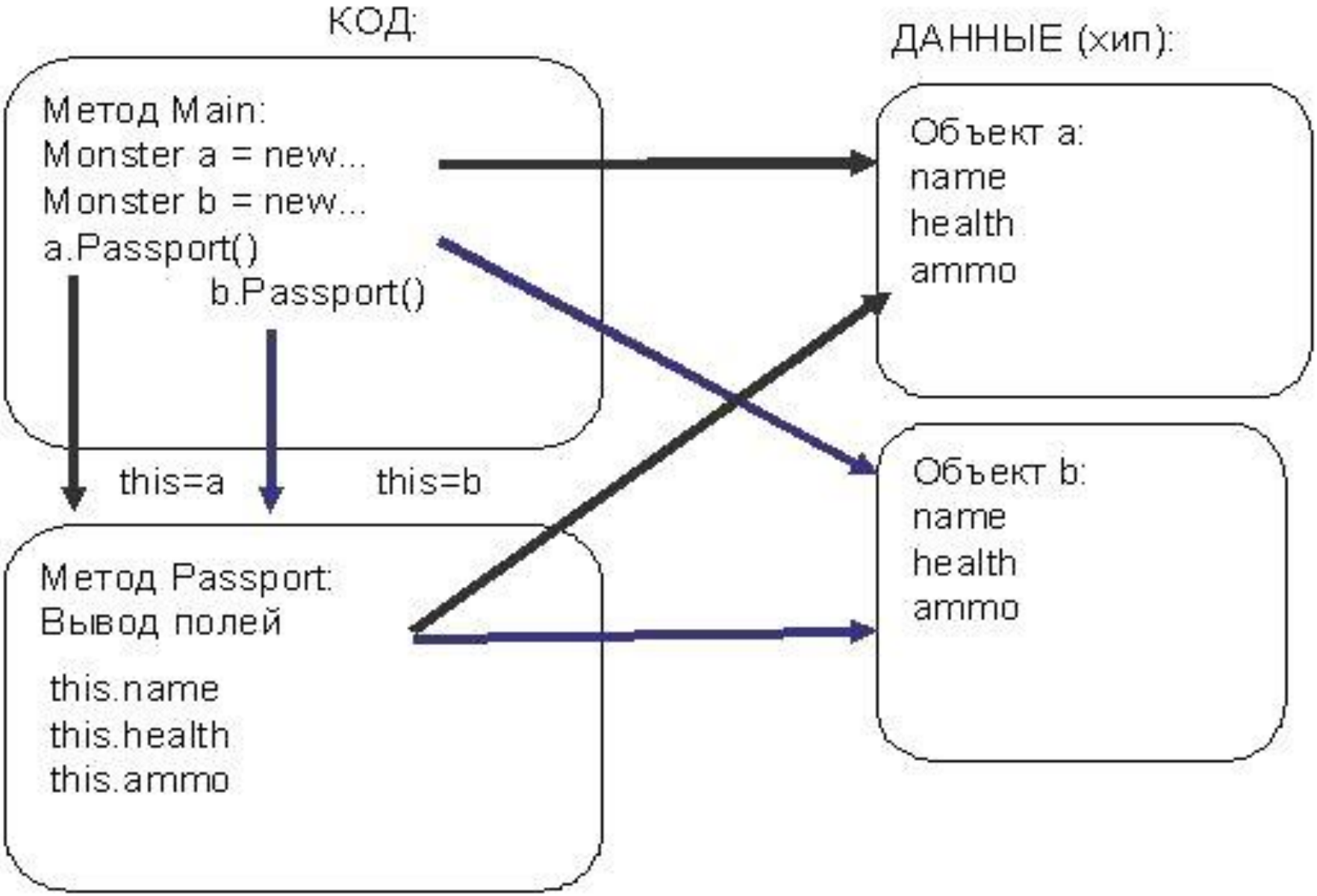
```
    catch (Exception e) {
        Console.WriteLine("Сгенерировано {0}
исключение.", e.GetType().FullName);
        Console.WriteLine("Сообщение:\n\"{0}\"",
e.Message);
    }
  }
```

```
  Console.WriteLine("\n Вывод
myObj.ToString()...");
  Console.WriteLine(myObj.ToString ());
  Console.WriteLine("myObj.ToString() Вывод.");
  Console.ReadKey();
}
```

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**.

Пример 12

```
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;
    double y;
  }
```

Пример 12

```
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 13

```
class Demo {
    public Demo( int a ) {  // конструктор 1
        this.a = a;
    }
    public Demo( int a, double y ) : this( a )  {
                    // вызов конструктора 1
        this.y = y;
    }
     ...
}
```

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 14

```
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 14

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

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 15

```
class Destruct {
    public int x;
    public Destruct(int 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 15

```
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!");
    }
}
```

Виртуальные, переопределяющие и абстрактные методы

Если объявление метода экземпляра содержит модификатор **virtual**, метод является **виртуальным методом**.

Виртуальный метод может быть **переопределен** в производном классе. Если объявление метода экземпляра содержит модификатор **override**, метод переопределяет унаследованный виртуальный метод с такой же сигнатурой. Объявление виртуального метода определяет новый метод. Объявление переопределяющего метода уточняет существующий виртуальный метод, предоставляя его новую реализацию.

Виртуальные, переопределяющие и абстрактные методы

Вместе с ключевым словом **override** может также использоваться и ключевое слово **sealed** (герметизированный), указывающее, что в данный метод больше не могут вноситься изменения ни в каких производных классах, т.е. метод не может **переопределяться** в производных классах.

```
public class MyDerivedClass : MyBaseClass {
        public override sealed void DoSomething () {
          // Реализация в производном классе,
          // переопределяющая базовую реализацию.
        }
}
```

Виртуальные, переопределяющие и абстрактные методы

Абстрактным называется виртуальный метод без реализации.

Объявление абстрактного метода осуществляется с использованием модификатора **abstract** и допускается только в классе, объявленном как **abstract**.

В каждом неабстрактном производном классе необходимо переопределять абстрактный метод.

Пример 16

```
public abstract class Expression{
   public abstract double Evaluate(Hashtable vars);
}
public class Constant: Expression{
   double value;
   public Constant(double value) {
    this.value = value;
   }
  public override double Evaluate(Hashtable vars){
    return value;
   }
}
```

Пример 16

```
public class VariableReference: Expression{
   string name;
   public VariableReference(string name) {
    this.name = name;
   }
   public override double Evaluate(Hashtable vars){
    object value = vars[name];
    if (value == null)  throw new Exception("Не
   определена переменная: " + name);
    return Convert.ToDouble(value);
   }
}
```

Пример 16

```
public class Operation: Expression{
    Expression left;
    char op;
    Expression right;
   public Operation(Expression left, char op,
   Expression right) {
    this.left = left;
    this.op = op;
    this.right = right;
   }
```

Пример 16

```
public override double Evaluate(Hashtable vars){
 double x = left.Evaluate(vars);
 double y = right.Evaluate(vars);
 switch (op) {
     case '+': return x + y;
     case '-': return x - y;
     case '*': return x * y;
     case '/': return x / y;
 }
 throw new Exception("Не определен оператор");
}
}
```

Пример 16

Четыре приведенных выше класса могут использоваться для моделирования арифметических выражений. Например, с помощью экземпляров этих классов выражение x + 3 можно представить следующим образом.

```
Expression e = new Operation(new
  VariableReference("x"), '+', new
  Constant(3));
```

Пример 16

```
class Test{ //классы Expression используются для вычисления
//  выражения x * (y + 2) с различными значениями x и y
    static void Main() {
      Expression e = new Operation(
     new VariableReference("x"),
     '*',
     new Operation(
          new VariableReference("y"),
          '+',
          new Constant(2)
      )
    );
```

Пример 16

```
    Hashtable vars = new Hashtable();
    vars["x"] = 3;
    vars["y"] = 5;
    Console.WriteLine(e.Evaluate(vars)); // Вывод "21"
    vars["x"] = 1.5;
    vars["y"] = 9;
    Console.WriteLine(e.Evaluate(vars)); //Вывод "16.5"
    }
}
```

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?

Контрольные вопросы

6. Что такое деструктор? Гарантирует ли среда его выполнение?
7. Какие действия обычно выполняются в части set свойства?
8. Может ли свойство класса быть не связанным с его полями?
9. Можно ли описать разные спецификаторы доступа к частям get и set свойства?