Введение в Объектно-ориентированное Программирование

Понятие **объекта**

- В реальном мире каждый **предмет** или **процесс** обладает набором **статических** и **динамических** характеристик (**свойствами** и **поведением**). ***Поведение*** *объекта* зависит от его ***состояния*** и *внешних* ***воздействий***.
- Понятие **объекта** в **программе** совпадает с обыденным смыслом этого слова: ***объект*** *представляется как совокупность* ***данных***, *характеризующих его* ***состояние***, *и* ***функций*** *их* ***обработки***, *моделирующих его* ***поведение***. **Вызов функции** на выполнение часто называют ***посылкой сообщения*** **объекту**.
- При создании **объектно-ориентированной** программы предметная область представляется в виде совокупности **объектов**. **Выполнение** программы состоит в том, что **объекты** обмениваются **сообщениями**.

Столы ООП

- **Инкапсуляция** – как объекты прячут своё внутренне устройство (blackbox)
- **Наследование** – как в этом языке поддерживается повторное использование кода
- **Полиморфизм** - как в этом языке реализована поддержка выполнения нужного действия в зависимости от типа передаваемого объекта

Абстрагирование и **инкапсуляция**

- При представлении **реального объекта** с помощью **программного** необходимо выделить в первом его **существенные** особенности и игнорировать **несущественные**. Это называется ***абстрагированием***.
- Таким образом, **программный объект** — это **абстракция**.
- Детали реализации **объекта** скрыты, он используется через его ***интерфейс*** — совокупность **правил доступа**.
- **Скрытие деталей реализации** называется ***инкапсуляцией***. Это позволяет представить программу в укрупненном виде — на уровне **объектов** и их **взаимосвязей**, а следовательно, управлять большим объемом информации.
- *Итак,* **объект** — это **инкапсулированная абстракция** с четко определенным **интерфейсом**.

Инкапсуляция свойств и методов

```
class deposit    {
        private int money;
        private string owner;
        private bool locked;


      public deposit(string owner, int money)
        {this.owner = owner; this.money = money; }
      public  bool lockaccount() {locked = true;}
      private bool unlockaccount() {locked = false;}
}}
```

Наследование

- Важное значение имеет возможность **многократного использования кода**. Для **объекта** можно определить **наследников**, **корректирующих** или **дополняющих** его **поведение**.

- ***Наследование*** применяется для:
  - исключения из программы **повторяющихся фрагментов** кода;
  - **упрощения модификации** программы;
  - **упрощения создания** новых программ на основе существующих.

- Благодаря **наследованию** появляется возможность **использовать объекты**, **исходный** код которых **недоступен**, но в которые требуется внести **изменения**.

Наследование

```
class deposit    {
        private int money;
        private string owner;

       public deposit(string owner, int money)
           {this.owner = owner; this.money = money; }
}

class vipdeposit : deposit
{
        private string currency;

        public vipdeposit(string cur, string owner, int money) : base(owner,
    money)
        { currency = cur; }

}
```

Полиморфизм

- **ООП** позволяет писать **гибкие**, **расширяемые** и читабельные программы.

- Во многом это обеспечивается благодаря «**полиморфным» методам**, когда есть возможность определения **единого** по **имени действия**, **применимого** ко всем **объектам иерархии**, причем каждый **объект** реализует это **именованное действие** (**полиметод**) собственным способом.

- Чаще всего понятие **полиморфизма** связывают с механизмом **виртуальных методов**.

Поликонструкторы

```
class deposit    {
      public deposit() { owner = "Ghost"; money = 0; }
       public deposit(string owner) { this.owner = owner; money = -
   100; }
       public deposit(string owner, int money)
          {this.owner = owner; this.money = money; }
}}
 class Program
   {
      static void Main(string[] args)
      {
         deposit Mike = new deposit("Mike", 100);
         deposit Andy = new deposit("Andy");
         deposit Igy = new deposit();
     }
   }
```

Достоинства ООП

- **использование** при программировании **понятий**, **близких** к **предметной области**;
- **возможность** успешно **управлять большими объемами исходного кода** благодаря инкапсуляции, то есть скрытию деталей реализации объектов и **упрощению структуры программы**;
- **возможность многократного использования кода** за счет **наследования**;
- сравнительно **простая** возможность **модификации** программ;
- возможность **создания** и **использования библиотек объектов**.

Недостатки ООП

- некоторое **снижение быстродействия** программы, связанное с использованием **виртуальных методов**;
- **идеи** ООП **не просты** для понимания и в особенности для практического использования;
- для **эффективного использования** существующих **объектно-ориентированных систем** требуется **большой объем** первоначальных знаний;
- **неграмотное применение ООП** может привести к значительному **ухудшению характеристик** разрабатываемой **программы**.

Понятие **класса**

- ***Класс*** является **типом данных**, определяемым пользователем. Он должен представлять собой одну логическую сущность, например, являться моделью реального объекта или процесса. ***Элементами*** класса являются ***данные*** и ***функции***, предназначенные для их обработки.
- Все **классы** .NET имеют общего **предка** — класс **object**, и организованы в единую **иерархическую структуру**.
- Внутри структуры **классы** логически сгруппированы в **пространства имен**, которые служат для упорядочивания имен классов и предотвращения **конфликтов имен**
- Любая программа использует **пространство** имен **System**.

Описание **класса**

[спецификаторы] class имя_класса [: предки]
тело_класса

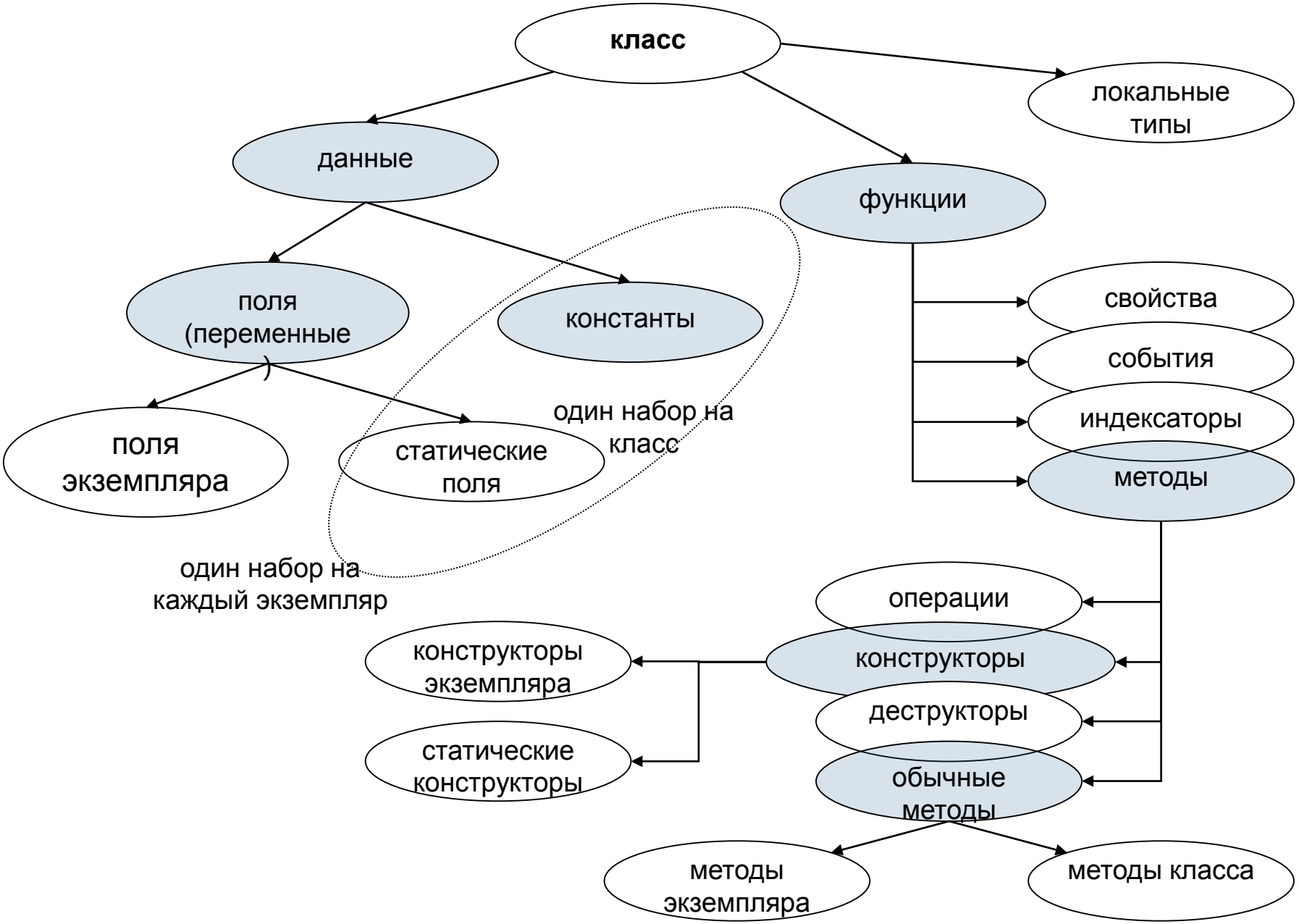
- **Имя класса** задается по общим правилам.
- **Тело класса** — список описаний его **элементов**, заключенный в фигурные скобки.
- **Спецификаторы** определяют **свойства класса**, а также **доступность класса** для других элементов программы.
- Простейший пример описания **класса**:

  class Demo {}

Спецификаторы класса

Спецификатор	Описание
new	Используется для вложенных классов. Задает новое описание класса взамен унаследованного от предка. Применяется в иерархиях
public	Доступ не ограничен
protected	Используется для вложенных классов. Доступ только из элементов данного и производных классов
internal	Доступ только из данной программы (сборки)
protected internal	Доступ только из данного и производных классов или из данной программы (сборки)
private	Используется для вложенных классов. Доступ только из элементов класса, внутри которого описан данный класс
abstract	Абстрактный класс. Применяется в иерархиях

Элементы класса



Описание объекта

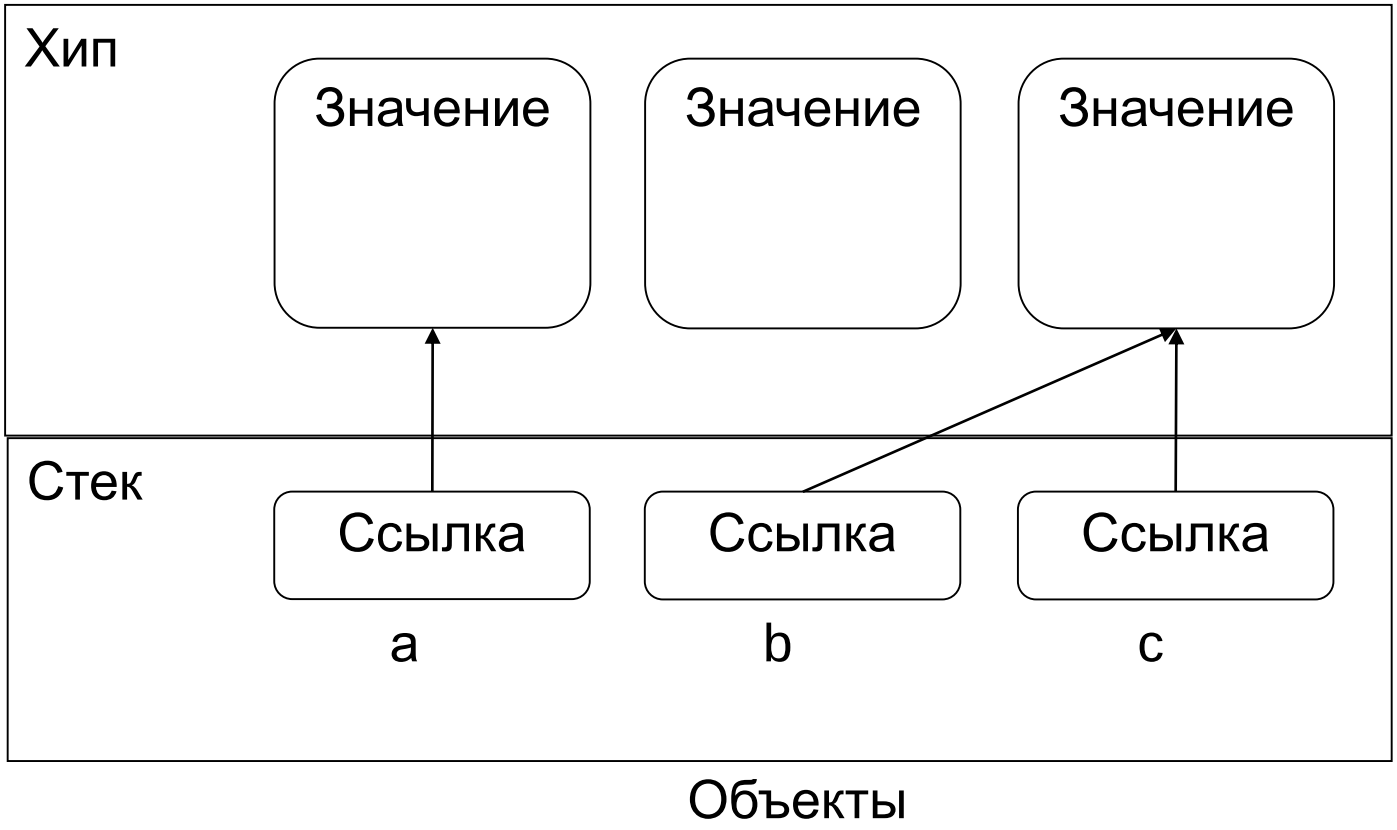
- **Класс** является обобщенным понятием, определяющим характеристики и поведение **множества конкретных объектов** этого класса, называемых **экземплярами** (**объектами**) **класса**.
- **Объекты** создаются **явным** или **неявным** образом (либо программистом, либо системой). Программист создает экземпляр класса с помощью операции **new**:

```
Demo a = new Demo();

Demo b = new Demo();
```

- Для каждого **объекта** при его создании в **памяти** выделяется **отдельная область** для хранения его данных.
- Кроме того, в **классе** могут присутствовать **статические элементы**, которые существуют в **единственном экземпляре** для **всех объектов класса**.
- Функциональные элементы класса всегда хранятся в единственном экземпляре.

Присваивание и сравнение объектов



- b = c
- Величины ссылочного типа равны, если они ссылаются на одни и те же данные (b == c, но a != b даже при равенстве их значений или если обе ссылки равны null).

Данные: поля и константы

- **Данные**, содержащиеся в **классе**, могут быть **переменными** или **константами**.
- **Переменные**, описанные в **классе**, называются **полями класса**.
- При описании **полей** можно указывать спецификаторы, задающие различные характеристики элементов:

[спецификаторы] [const] тип имя

[= начальное_значение]

- Все **поля** сначала **автоматически инициализируются нулем** соответствующего **типа** (например, полям типа int присваивается 0, а ссылкам на объекты — значение null).
- После этого полю присваивается значение, заданное при его явной инициализации.

Спецификаторы полей и констант класса

Спецификатор	Описание
new	Новое описание поля, скрывающее унаследованный элемент класса
public	Доступ к элементу не ограничен
protected	Доступ только из данного и производных классов
internal	Доступ только из данной сборки
protected internal	Доступ только из данного и производных классов и из данной сборки
private	Доступ только из данного класса
static	Одно поле для всех экземпляров класса
readonly	Поле доступно только для чтения
volatile	Поле может изменяться другим процессом или системой

Пример описания класса

```
using System;
namespace CA1
{    class Demo
   {  public int a = 1;                        // поле данных
      public const double  c = 1.66;     // константа
      public static string s = "Demo";  // статическое поле класса
      double y;                                  // закрытое поле данных
   }
   class Class1
   {   static void Main()
      {
         Demo x = new Demo();       // создание экземпляра класса Demo
         Console.WriteLine( x.a );        // x.a - обращение к полю класса
         Console.WriteLine( Demo.c );  // Demo.c - обращение к константе
         Console.WriteLine( Demo.s );  // обращение к статическому полю
  }}}
```

Методы

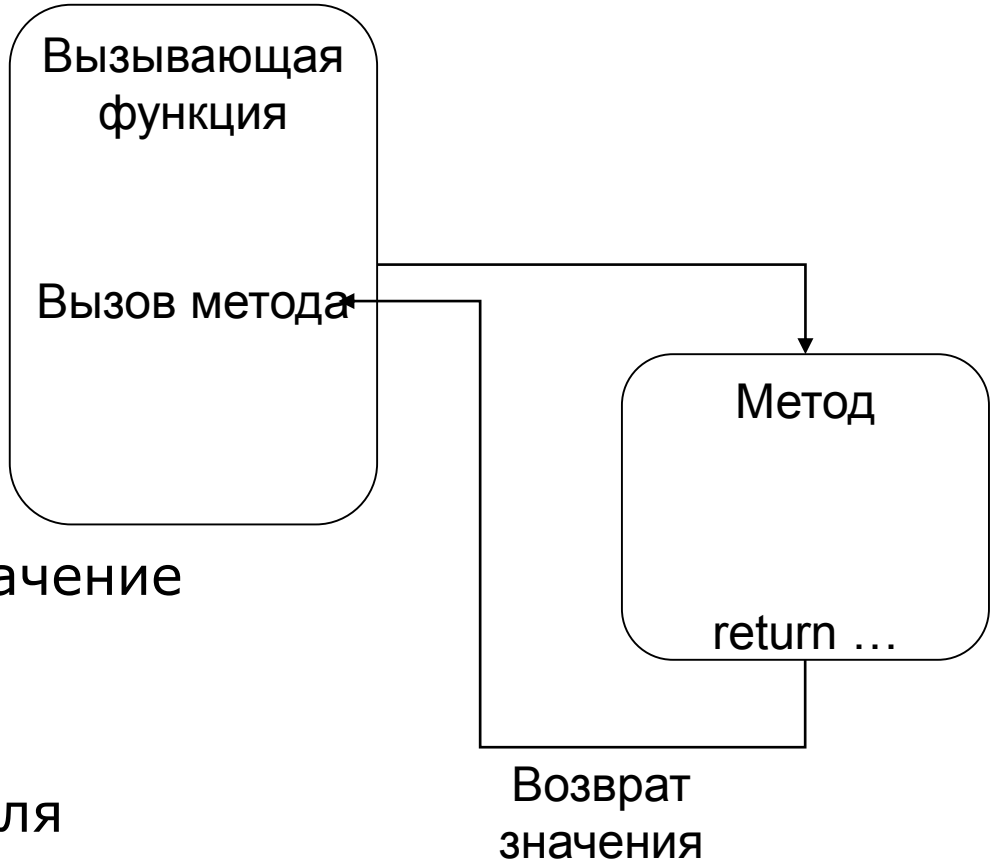
- **Метод** — **функциональный** элемент **класса**, реализующий **вычисления** или другие **действия**. **Методы** определяют **поведение класса** и составляют его интерфейс.
- **Метод** — законченный **фрагмент кода**, к которому можно **обратиться** по **имени**. Он описывается один раз, а вызываться может столько раз, сколько необходимо.
- Один и тот же метод может обрабатывать различные данные, переданные ему в качестве аргументов.
- Синтаксис метода:

[спецификаторы] тип имя_метода ([параметры])
тело_метода

- При **описании методов** можно использовать первые 7 **спецификаторов** полей, а также virtual, sealed, override, abstract и extern. Чаще всего для методов задается **public**.

Примеры методов

```
public void Sety(double z)
{ y = z; }
public double Gety()
{ return y; }
```



- Тип метода определяет, значение какого типа вычисляется с помощью метода
- Параметры используются для обмена информацией с методом. Параметр - локальная переменная, которая при вызове метода принимает значение соответствующего аргумента.

```
x.Sety(3.12);
double t = x.Gety();
```

```
double a = 0.1;
double b = Math.Sin(a);
Console.WriteLine(a);
```

Параметры методов

- **Параметры** определяют **множество значений аргументов**, которые можно передавать в **метод**.
- **Список аргументов** при вызове как бы накладывается на список **параметров**, поэтому они должны попарно соответствовать друг другу.
- Для каждого **параметра** должны задаваться его **тип**, **имя** и, возможно, **вид параметра**.
- **Имя** метода вкупе с **количеством**, **типами** и **спецификаторами** его **параметров** представляет собой **сигнатуру метода** — то, по чему один метод отличают от других.
- В **классе** не должно быть методов с **одинаковыми сигнатурами**.
- **Метод**, описанный со **спецификатором** static, должен обращаться только к **статическим** полям класса.
- **Статический** метод вызывается через **имя класса**, а обычный — через **имя экземпляра**.

Пример

```
class Demo    {
        public int a = 1;
        public const double c = 1.66;
        static string s = "Demo";
        double y;
        public double Gety() { return y;  }            // метод получения y
        public void Sety( double y_ ){  y = y_; }    // метод установки y
        public static string Gets() { return s; }      // метод получения s
}
class Class1 {
       static void Main()
        {  Demo x = new Demo();
           x.Sety(0.12);                          // вызов метода установки y
           Console.WriteLine(x.Gety());         // вызов метода получения y
           Console.WriteLine(Demo.Gets());  // вызов метода получения s
}}
```

Вызов метода

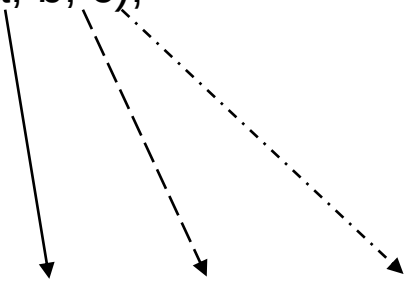
При вызове метода:

1. Вычисляются **выражения**, стоящие на месте **аргументов**.
2. Выделяется **память** под **параметры метода**.
3. Каждому из **параметров** сопоставляется соответствующий **аргумент**. При этом проверяется соответствие **типов аргументов** и **параметров** и при необходимости **выполняется** их **преобразование**. При несоответствии типов выдается диагностическое сообщение.
4. Выполняется тело **метода**.
5. Если **метод** возвращает **значение**, оно передается в **точку вызова**; если **метод** имеет тип **void**, управление передается на **оператор**, следующий после **вызова**.

Иллюстрация передачи параметров

Описание аргументов: int b; double a, c;

Вызов метода: obj.P(a, b, c);



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

Пример передачи параметров

```
class Class1
   {   static int Max(int a, int b)       // выбор макс. значения
       {
          if ( a > b ) return a;
          else         return b;
       }
       static void Main()
       {
          int a = 2, b = 4;
          int x = Max( a, b );                  // вызов метода Max
          Console.WriteLine( x );               // результат: 4
          short t1 = 3, t2 = 4;
          int y = Max( t1, t2 );                // вызов метода Max
          Console.WriteLine( y );               // результат: 4
          int z = Max( a + t1, t1 / 2 * b );    // вызов метода Max
          Console.WriteLine( z );               // результат: 5
}}
```

Способы передачи параметров и их типы

Способы передачи параметров: по значению и по ссылке.

- *При передаче по значению* метод получает копии значений аргументов, и операторы метода работают с этими копиями.
- *При передаче по ссылке* (*по адресу*) метод получает копии адресов аргументов и осуществляет доступ к аргументам по этим адресам.

В C# четыре типа параметров:

- параметры-значения;
- параметры-ссылки (ref);
- выходные параметры (out);
- параметры-массивы (params).

Ключевое слово предшествует описанию типа параметра. Если оно опущено, параметр считается параметром-значением. Пример:

```
public int Calculate( int a, ref int b, out int c, params int[] d ) …
```

Пример: параметры-значения и ссылки

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void P( int a, ref int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
            int a = 2, b = 4;
            Console.WriteLine( "до вызова     {0} {1}", a, b );
            P( a, ref b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
}}}
```

Результат работы программы:
до вызова 2 4
внутри метода 44 33
после вызова 2 33

Применение параметров-значений и ссылок

1. Для параметров-значений используется передача по значению. Этот способ применяется для исходных данных метода.

- При вызове метода на месте параметра, передаваемого по значению, может находиться выражение (а также его частные случаи — переменная или константа). Должно существовать неявное преобразование типа выражения к типу параметра.

2. Параметры-ссылки передаются по адресу. Этот способ применяется для передачи побочных результатов метода.

- При вызове метода на месте параметра-ссылки может находиться только имя инициализированной переменной точно того же типа. Перед именем параметра указывается ключевое слово ref.

Ключевое слово this

- Чтобы обеспечить работу метода с полями того объекта, для которого он был вызван, в метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию объект.
- В явном виде параметр this применяется:

```
// чтобы возвратить из метода ссылку на вызвавший объект:
class Demo
{
        double y;
        public Demo T()    { return this; }
// для идентификации поля, если его имя совпадает с именем
// параметра метода:
        public void Sety( double y ) { this.y = y; }
}
```

Конструкторы

Конструктор предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса.

Свойства конструкторов:

- 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, полям значимых типов присваивается нуль, полям ссылочных типов — значение null.
- Конструктор, вызываемый без параметров, называется конструктором по умолчанию.

Пример класса с конструктором

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

   class Class1
   {   static void Main()
      {
         Demo a = new Demo( 300, 0.002 );     // вызов конструктора
         Demo b = new Demo( 1, 5.71 );        // вызов конструктора
         …
    } }
```

Пример класса с двумя конструкторами

```
class Demo
   {
      public Demo( int a )                  // конструктор 1
      {
         this.a = a;
         this.y = 0.002;
      }
      public Demo( double y )                 // конструктор 2
      {
         this.a = 1;
         this.y = y;
      }
      ...
   }
      ...
         Demo x = new Demo( 300 );          // вызов конструктора 1
         Demo y = new Demo( 5.71 );         // вызов конструктора 2
```

Свойства

- Свойства служат для организации доступа к полям класса. Как правило, свойство определяет методы доступа к закрытому полю.
- Синтаксис свойства:

[спецификаторы] тип имя_свойства

{

[get код_доступа]

[set код_доступа]

}

- Чаще всего свойства объявляются как открытые (public).
- Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке (set) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно.
- Если отсутствует часть set, свойство доступно только для чтения (read-only), если отсутствует часть get, свойство доступно только для записи (write-only).

Пример описания свойств

```
public class Button: Control
{  private string caption;      // поле, с которым связано свойство
   public string Caption {      // свойство
      get { return caption; }  // способ получения свойства

      set                              // способ установки свойства
         { if (caption != value) { caption = value; }
}}    ...
```

В программе свойство выглядит как поле класса:

```
Button ok = new Button();
ok.Caption = "OK";              // вызывается метод установки свойства
string s = ok.Caption;          // вызывается метод получения свойства
```

При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки.

Рекомендации по программированию

- Интерфейс класса должен быть интуитивно ясен, непротиворечив и обозрим.
- Интерфейс должен быть полным (предоставлять возможность выполнять любые разумные действия с классом), и минимально необходимым (без дублирования и пересечения возможностей методов).
- Поля предпочтительнее делать закрытыми (private). Поля, характеризующие класс в целом, то есть имеющие одно и то же значение для всех экземпляров, следует описывать как статические.
- Необходимо стремиться к максимальному сокращению области действия каждой переменной.
- Все литералы, связанные с классом (числовые и строковые константы), описываются как поля-константы с именами, отражающими их смысл.
- Каждый метод класса должен решать только одну задачу.