Классы: основные понятия

Основные элементы класса: поля, методы, конструкторы, свойства. Виды параметров методов.

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.

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.

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 позволяет создавать *иерархии объектов*. Иерархия представляется в виде дерева, в котором более общие объекты располагаются ближе к корню, а более специализированные — на ветвях и листьях.

Полиморфизм

- ООП позволяет писать гибкие, расширяемые и читабельные программы.
- Во многом это обеспечивается благодаря полиморфизму, под которым понимается возможность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа.
- Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

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.

Технология разработки ОО программ

В процесс проектирования добавляется еще один этап - разработка иерархии классов.

1. в предметной области выделяются понятия, которые можно использовать как классы. Кроме классов из прикладной области, появляются классы, связанные с аппаратной частью и реализацией
2. определяются операции над классами, которые впоследствии станут методами класса. Их можно разбить на группы:

   - связанные с конструированем и копированем объектов

   - для поддержки связей между классами, которые существуют в прикладной области

   - позволяющие представить работу с объектами в удобном виде.
3. Определяются функции, которые будут виртуальными.
4. Определяются зависимости между классами. Процесс создания иерархии классов - итерационный. Например, можно в двух классах выделить общую часть в базовый класс и сделать их производными.

Классы должны как можно ближе соответствовать моделируемым объектам из предметной области.

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 нее классы логически сгруппированы в пространства имен, которые служат для упорядочивания имен классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными.
- Любая программа использует пространство имен System.

Описание класса

[атрибуты] [спецификаторы] class имя_класса [: предки] тело_класса

- Имя класса задается по общим правилам.
- Тело класса — список описаний его элементов, заключенный в фигурные скобки.
- Атрибуты задают дополнительную информацию о классе.
- Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы.

- Простейший пример описания класса:

```
class Demo {}        // пустой класс
```

Сквозной пример класса

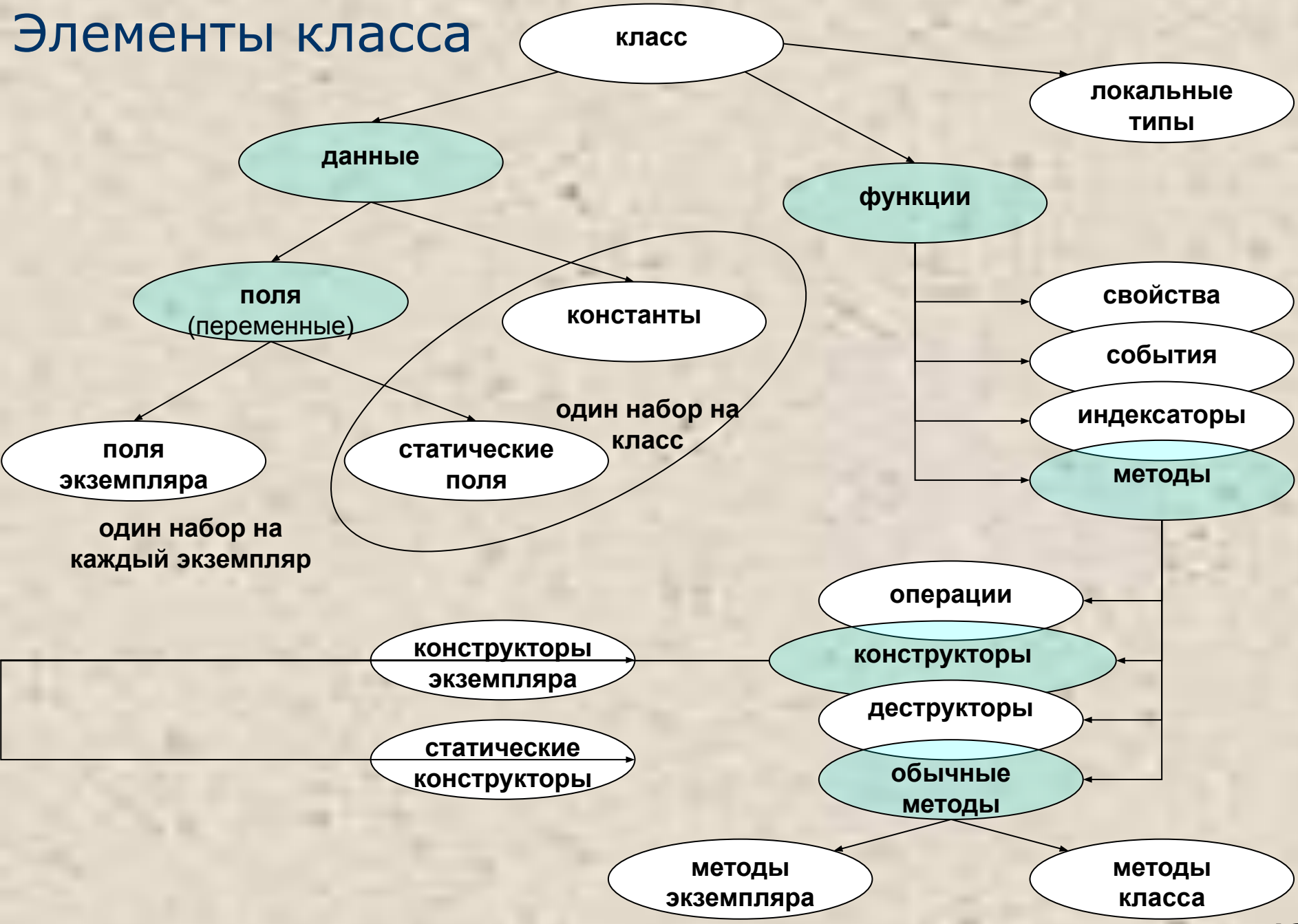
```
class Monster {
    public Monster()      // конструктор
    {
            this.name  = "Noname";
            this.health = 100;
            this.ammo = 100;
    }
  public Monster( string name ) : this()
    {
            this.name = name;
     }
  public Monster( int health, int ammo,
     string name )
     {
            this.name  = name;
            this.health = health;
            this.ammo = ammo;
     }
  public int GetName()          // метод
    {  return name;   }
  public int GetAmmo()        // метод
     {  return ammo;}
```

```
public int Health {                  // свойство
   get { return health;  }
   set { if (value > 0) health = value;
           else          health = 0;
         }
 }

 public void Passport()               // метод
     {   Console.WriteLine(
  "Monster {0} \t health = {1} \
    ammo = {2}", name, health, ammo );
      }

public override string ToString(){
     string buf = string.Format(
       "Monster {0} \t health = {1} \
     ammo = {2}", name, health, ammo);
    return buf;  }

  string name;                       // поле
  int health, ammo;                  // поле
}
```

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
sealed	Бесплодный класс. Применяется в иерархиях
static	Статический класс. Введен в версию языка 2.0.

Элементы класса



Описание объекта (экземпляра)

- 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();
```

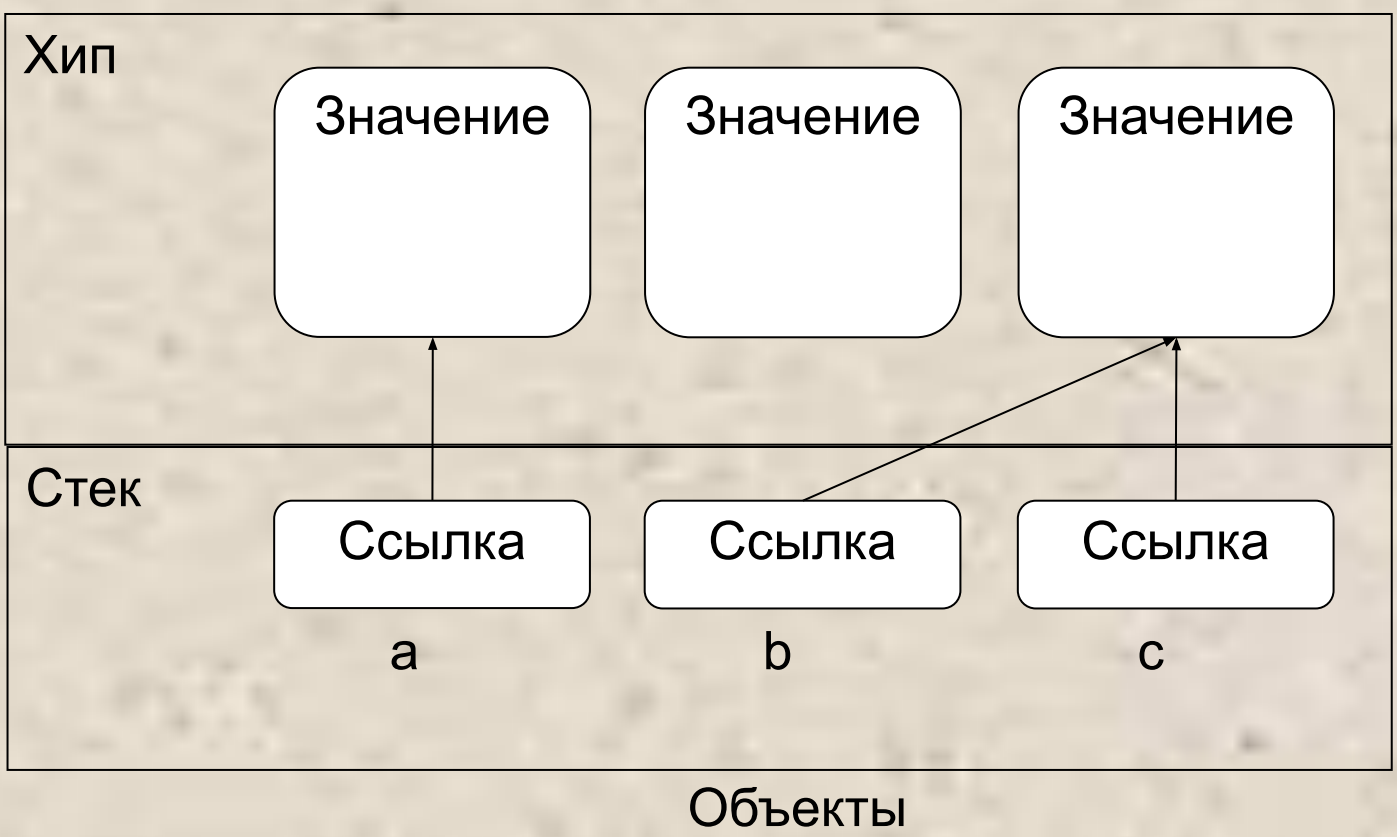
- Для каждого объекта при его создании в памяти выделяется отдельная область для хранения его данных.
- Кроме того, в классе могут присутствовать статические элементы, которые существуют в единственном экземпляре для всех объектов класса.
- Функциональные элементы класса всегда хранятся в единственном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

Пример создания объектов (экземпляров)

```
class Monster { ... }
 class Class1
{
        static void Main()
        {
            Monster X = new Monster();
            X.Passport();
            Monster Vasia = new Monster( "Vasia" );
            Vasia.Passport();
            Monster Masha = new Monster( 200, 200, "Masha" );
            Console.Writeline(Masha);
        }
    }
```

Результат работы программы:
Monster Noname health = 100 ammo = 100
Monster Vasia health = 100 ammo = 100
Monster Masha health = 200 ammo = 200

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 атрибуты и спецификаторы, задающие различные характеристики элементов:

[атрибуты] [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.

Пример 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
  }}}
```

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

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.

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

Синтаксис метода

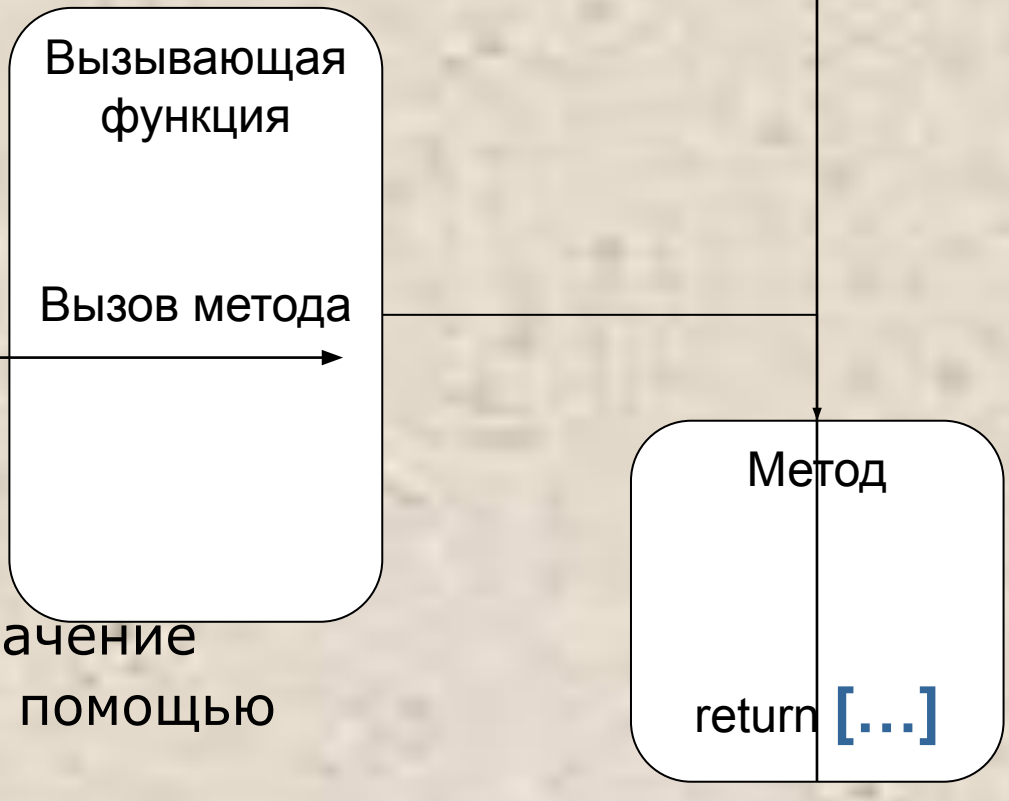
[атрибуты] [спецификаторы] тип имя_метода (**[параметры]**) тело_метода

- Спецификаторы: new, **public**, protected, internal, protected internal, private, static, virtual, sealed, override, abstract, extern.
- Метод класса имеет непосредственный доступ к его полям.
- Пример:

```
class Demo {
    double y;                          // закрытое поле класса

    public void Sety( double z ) {     // открытый метод класса
        y = z;
    }
}
… Demo x = new Demo();          // где-то в методе другого класса
  x.Sety(3.12); …               // вызов метода
```

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();
```

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
//          Console.WriteLine(Gets());                 // вариант вызова из того же
}}                                                     // класса
```

Вызов 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.

Описание объекта: SomeObj obj = new SomeObj();

Описание аргументов: int b; double a, c;

Вызов метода: obj.P(a, b, c);

Заголовок метода P: 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 );      // пар-ры совместимого типа
          Console.WriteLine( y );                 // результат: 4
          int z = Max( a + t1, t1 / 2 * b );   // выражения
          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 ref

```
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

Пример: выходные параметры out

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void P( int x, out int y )
        {
            x = 44; y = 33;
            Console.WriteLine( "внутри метода {0} {1}", x, y );
        }
        static void Main()
        {
            int a = 2, b;         // инициализация b не требуется

            P( a, out b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
}}}
```

Результат работы программы:

внутри метода 44 33
после вызова 2 33

Правила применения параметров

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.

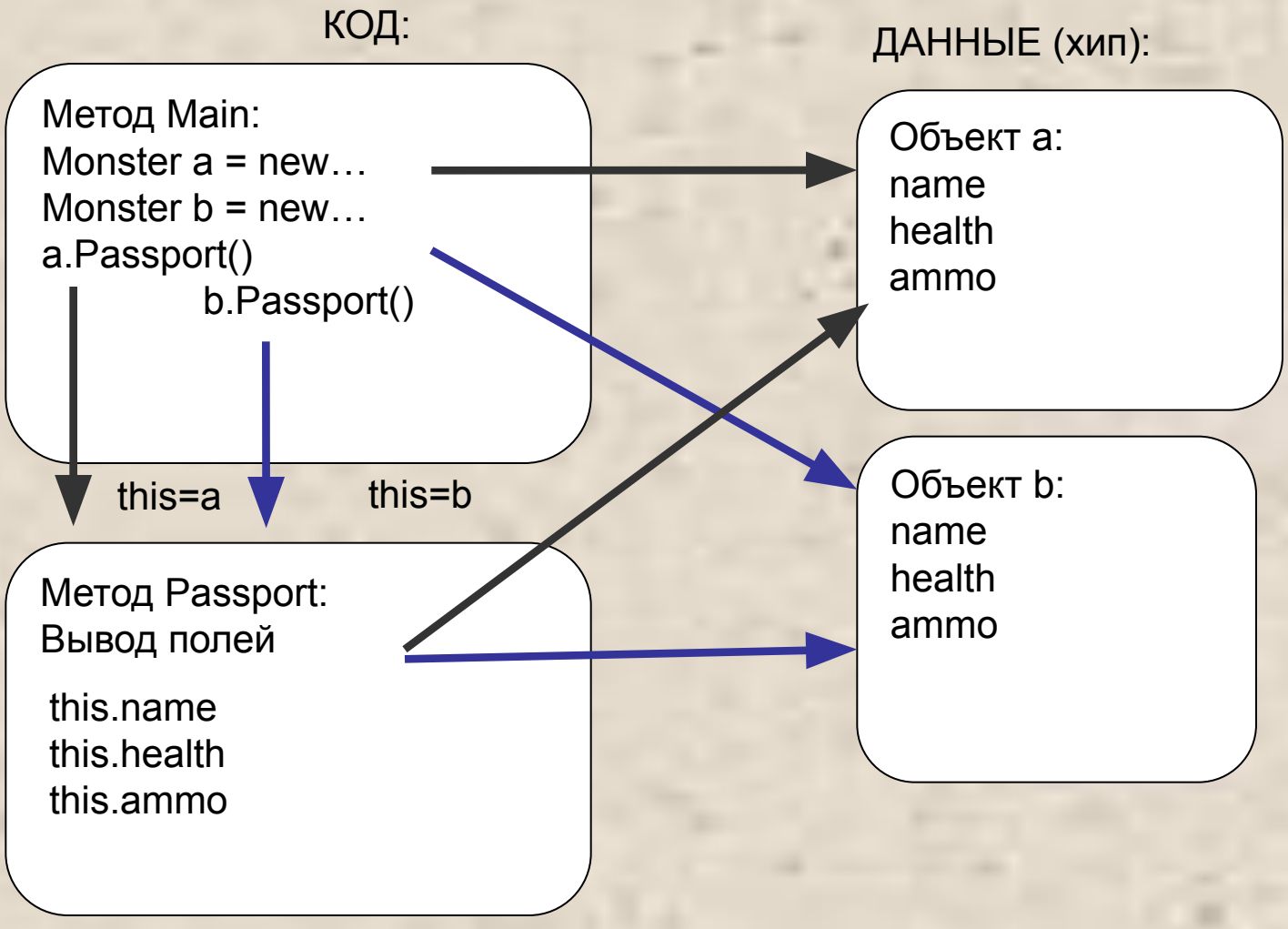
2. Параметры-ссылки и выходные параметры передаются по адресу. Этот способ применяется для передачи побочных результатов метода.

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

- При вызове метода на месте выходного параметра **out** может находиться только имя переменной точно того же типа. Ее инициализация не требуется. Перед именем параметра указывается ключевое слово out.

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.



Использование явного this

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
```

Сквозной пример класса

```
class Monster {
    public Monster()      // конструктор
    {
            this.name  = "Noname";
            this.health = 100;
            this.ammo = 100;
    }
  public Monster( string name ) : this()
    {
            this.name = name;
     }
  public Monster( int health, int ammo,
     string name )
     {
            this.name  = name;
            this.health = health;
            this.ammo = ammo;
     }
  public int GetName()          // метод
    {  return name;   }
  public int GetAmmo()         // метод
     {  return ammo;}
```

```
public int Health {                  // свойство
   get { return health;  }
   set { if (value > 0) health = value;
           else                 health = 0;
         }
 }

 public void Passport()                 // метод
     {   Console.WriteLine(
  "Monster {0} \t health = {1} \
    ammo = {2}", name, health, ammo );
      }

public override string ToString(){
     string buf = string.Format(
       "Monster {0} \t health = {1} \
     ammo = {2}", name, health, ammo);
    return buf;  }

  string name;
  int health, ammo;
}
```

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]

}

При обращении к свойству автоматически вызываются указанные в нем блоки чтения (**get)** и установки (**set)**.

- Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, свойство доступно только для чтения (read-only), если отсутствует get - 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
```

Сквозной пример класса

```
class Monster {
    public Monster()      // конструктор
    {
            this.name  = "Noname";
            this.health = 100;
            this.ammo = 100;
    }
  public Monster( string name ) : this()
    {
            this.name = name;
     }
  public Monster( int health, int ammo,
     string name )
     {
            this.name  = name;
            this.health = health;
            this.ammo = ammo;
     }
  public int GetName()          // метод
    {  return name;   }
  public int GetAmmo()        // метод
     {  return ammo;}
```

```
public int Health {                  // свойство
   get { return health;  }
   set { if (value > 0) health = value;
           else                health = 0;
         }
 }

 public void Passport()                  // метод
     {   Console.WriteLine(
  "Monster {0} \t health = {1} \
    ammo = {2}", name, health, ammo );
      }

public override string ToString(){
     string buf = string.Format(
       "Monster {0} \t health = {1} \
     ammo = {2}", name, health, ammo);
    return buf;  }

  string name;
  int health, ammo;
}
```

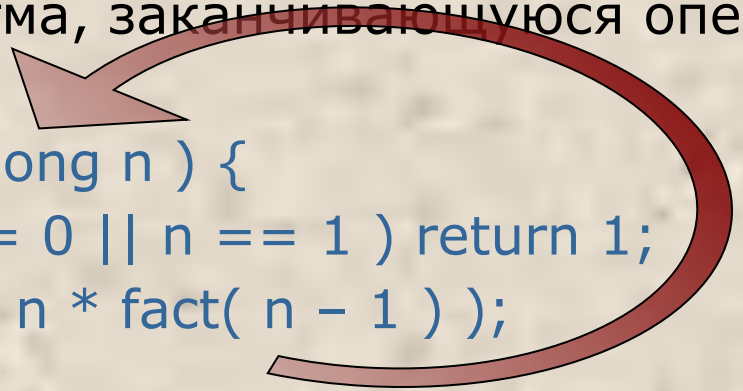
Методы с переменным количеством аргументов

```
class Class1  {
    public static double Average( params int[] a ) {
        if ( a.Length == 0 )
                    throw new Exception( "Недостаточно аргументов");
          double sum = 0;
          foreach ( int elem in a ) sum += elem;
          return sum / a.Length;
       }
       static void Main()    {   try   {
              int[] a = { 10, 20, 30 };
              Console.WriteLine( Average( a ) );          // 1
              int[] b = { -11, -4, 12, 14, 32, -1, 28 };
              Console.WriteLine( Average( b ) );          // 2
              short z = 1, e = 13;
              byte v = 100;
              Console.WriteLine( Average( z, e, v ) );     // 3
              Console.WriteLine( Average() );             // 4
          }
          catch( Exception e ) {Console.WriteLine( e.Message ); return; }
   }}
```

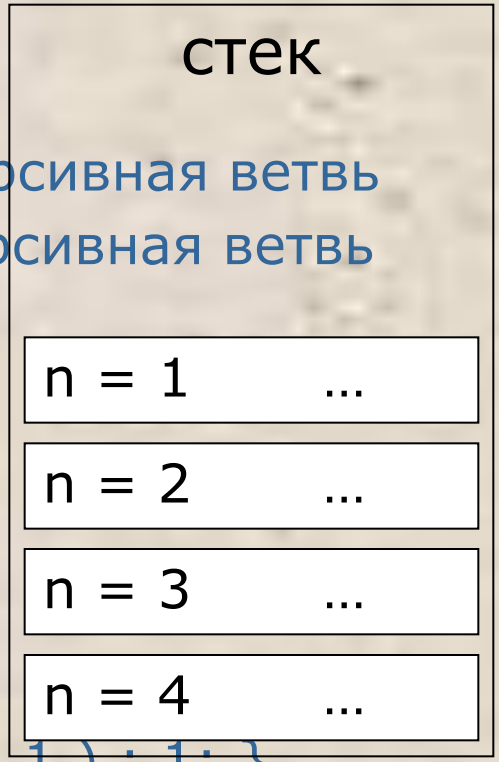
Результат:
20
10
38
Недостаточно аргументов

Рекурсивные методы

- *Рекурсивным* называется метод, который вызывает сам себя (*прямая рекурсия*). *Косвенная* рекурсия - когда два или более метода вызывают друг друга.
- Для завершения вычислений каждый рекурсивный метод должен содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата.



```
long fact( long n ) {
    if ( n == 0 || n == 1 ) return 1;      // нерекурсивная ветвь
    return ( n * fact( n – 1 ) );          // рекурсивная ветвь
}
… long m=fact(4);
```




```
// или:
long fact( long n ) { return ( n > 1 ) ? n * fact( n – 1 ) : 1; }
```

Характеристики рекурсии

Достоинство рекурсии: компактность записи.

Недостатки: опасность переполнения стека;
расход времени и памяти на повторные вызовы метода и передачу ему копий параметров.

Перегрузка методов

- Использование нескольких методов с одним и тем же именем, но различными типами параметров называется *перегрузкой методов*.
- Компилятор определяет, какой именно метод требуется вызвать, по типу фактических параметров. Это называется *разрешением* (resolution) перегрузки.

```
// Возвращает наибольшее из двух целых:
int max( int a, int b )
// Возвращает наибольшее из трех целых:
int max( int a, int b, int c )
// Возвращает наибольшее из первого параметра и длины второго:
int max ( int a, string b )
...
Console.WriteLine( max( 1, 2 ) );
Console.WriteLine( max( 1, 2, 3 ) );
Console.WriteLine( max( 1, "2" ) );
```

- Перегрузка методов является проявлением *полиморфизма*

Операции класса

- В C# можно переопределить для своих классов действие большинства операций. Это позволяет применять экземпляры объектов в составе выражений аналогично переменным стандартных типов:

```
MyObject a, b, c;  ...
c = a + b;                    // операция сложения класса MyObject
```

- Определение собственных операций класса называют *перегрузкой операций*.
- Операции класса описываются с помощью методов специального вида (*функций-операций*):

public static объявитель_операции тело

Например: public static MyObject operator --(MyObject m) { … }

В C# три вида операций класса: унарные, бинарные и операции преобразования типа.

Общие правила описания операций класса

- операция должна быть описана как открытый статический метод класса (спецификаторы **public static**);
- параметры в операцию должны передаваться **по значению** (то есть не должны предваряться ключевыми словами ref или out);
- сигнатуры всех операций класса должны различаться;
- типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

Унарные операции

Можно определять: **+ - ! ~ ++ -- true false**

Примеры заголовков:

```
public static int operator +( MyObject m )     // унарный плюс
public static MyObject operator --( MyObject m ) // декремент
```

Параметр должен иметь тип этого класса.

Операция должна возвращать:

- для операций +, -, ! и ~ величину любого типа;
- для операций ++ и -- величину типа класса, для которого она определяется.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

Префиксный и постфиксный инкремент/декремент не различаются

Пример унарной операции класса

```
class Monster {
  public static Monster operator ++(Monster m)
  {
     Monster temp = new Monster();
     temp.health = m.health + 1;
     return temp;
  }
  …
}
```

```
 …
Monster vasia = new Monster();
++vasia; vasia++;
…
```

Бинарные операции

Можно определять:

+ - * / % & | ^ << >> == != > < >= <=

Примеры заголовков бинарных операций:

```
public static MyObject operator +  ( MyObject m1, MyObject m2 )
public static bool     operator == ( MyObject m1, MyObject m2 )
```

Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа.

Операции == и !=, > и <, >= и <= определяются только парами и обычно возвращают логическое значение.

Пример бинарных операций класса

```
class Monster {
  public static Monster operator +( Monster m, int k )
  { Monster temp = new Monster();
    temp.ammo = m.ammo + k;
    return temp;
  }
 public static Monster operator +( int k, Monster m )
  { Monster temp = new Monster();
    temp.ammo = m.ammo + k;
    return temp;
  }
  …
}
 …
Monster vasia = new Monster();
Monster masha = vasia + 10;
Monster petya = 5 + masha;
…
```

Операции преобразования типа

Обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных.

implicit operator тип (параметр) // неявное преобразование

explicit operator тип (параметр) // явное преобразование

Выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразование либо типа класса к другому типу, либо наоборот.

```
public static implicit operator int( Monster m )
{
    return m.health;
}
public static explicit operator Monster( int h )
{
    return new Monster( h, 100, "FromInt" );
} ...
Monster Masha = new Monster( 200, 200, "Masha" );
int i = Masha;                  // неявное преобразование
Masha = (Monster) 500;          // явное преобразование
```

Применение операций преобразования

- *Неявное преобразование* выполняется автоматически:
  - при присваивании объекта переменной целевого типа;
  - при использовании объекта в выражении, содержащем переменные целевого типа;
  - при передаче объекта в метод на место параметра целевого типа;
  - при явном приведении типа.
- *Явное преобразование* выполняется при использовании операции приведения типа.

Индексаторы

Индексатор представляет собой разновидность свойства. Если у класса есть скрытое *поле*, представляющее собой *массив*, то с помощью индексатора можно обратиться к элементу этого массива, используя *имя объекта* и номер элемента массива в квадратных скобках.

Синтаксис индексатора аналогичен синтаксису свойства:

```
спецификаторы тип this [ список_параметров ]
{
 get код_доступа
 set код_доступа
 }
```

Код доступа представляет собой блоки операторов, которые выполняются при получении (`get`) или установке значения (`set`) элемента массива. Может отсутствовать либо часть `get`, либо `set`, но не обе одновременно. Если отсутствует часть `set`, *индексатор* доступен только для чтения (*read-only*), если отсутствует часть `get`, *индексатор* доступен только для записи (write-only).

Список параметров содержит одно или несколько описаний индексов, по которым выполняется *доступ* к элементу. Чаще всего используется один *индекс* целого типа.

Пример

```
class Book
{    public Book(string name)
    {
        this.Name=name;
    }
    public string Name { get; set; }
}

class Library
{
    Book[] books;

    public Library()
    {  books = new Book[] { new Book("Отцы и дети"), new
Book("Война и мир"), new Book("Евгений Онегин") }; }

     public int Length
     { get { return books.Length; }
     }
    public Book this[int index]
    {  get  {  return books[index]; }
        set  { books[index] = value;} }
    }
```

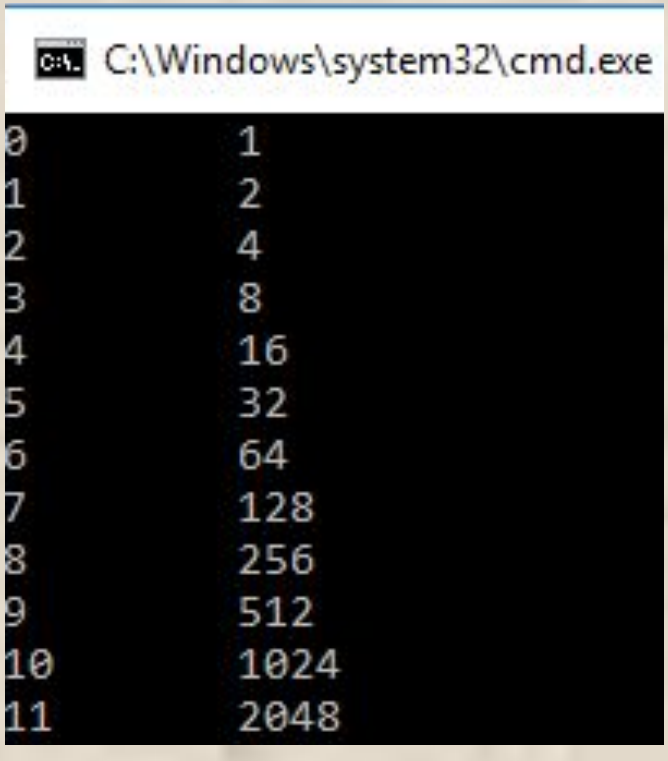
```
class Program
{
    static void Main(string[] args)
    {
        Library library = new Library();
        Console.WriteLine(library[0].Name);
        library[0] = new Book("Преступление и
наказание");
        Console.WriteLine(library[0].Name);

        Console.ReadLine();
    }
}
```

Многомерные индексаторы

Допускает использование *многомерных индексаторов*. Они описываются аналогично обычным и применяются в основном для контроля за занесением данных в многомерные массивы и выборке данных из многомерных массивов, оформленных в виде классов.

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

Индексаторы без массива




```
namespace ConsoleApp4
{
   class Program
   {
      static void Main(string[] args)
      {
         int n = 13;
         Pow2 pow2 = new Pow2();
         for (int i = 0; i < n; i++)
            Console.WriteLine("{0} \t {1}", i, pow2[i]);
      }   }

   class Pow2
   {
      public ulong this [int i]
      {
         get
         {
            if (i >= 0)
            {
               ulong res = 1;
               for (int k = 0; k < i; k++)
                  res *= 2;
               return res;
            }
            else
               return 0;
          } } }}
```

Метод Main

Метод, которому передается управление после запуска программы, должен иметь имя `Main` и быть статическим. Он может *принимать параметры* из внешнего окружения и *возвращать значение* в вызвавшую среду. Предусматривается два варианта метода — с параметрами и без параметров:

```
// без параметров:
static тип Main() { ... }
static void Main() { ... }
 // с параметрами:
 static тип Main( string[] args ) { /* ... */ }
 static void Main( string[] args ) { /* ... */ }
```