К Л А С С Ы

ДВЕ РОЛИ К Л А С С О В

- ***Класс – это модуль***, **архитектурная единица построения программной системы.**

- ***Класс – это сложный тип данных***, **в котором объединены элементы данных (поля) и методы, обрабатывающие эти данные и выполняющие операции по взаимодействию с окружающей средой.**

СИНТАКСИС ОПИСАНИЯ КЛАССА

[спецификатор доступа] [модификатор класса]

```
class имя_класса [:базовый класс]
{
тело_класса
}
```

Спецификаторы:

public

Internal (по умолчанию)

В теле класса могут быть:

- – ***константы*** – хранят неизменяемые значения, связанные с классом;
- – ***поля*** – содержат данные класса;
- – ***методы*** – функции; реализуют вычисления или другие действия, выполняемые классом или экземпляром;
- – ***свойства*** определяют характеристики класса в совокупности со способами их задания и получения, то есть методами записи и чтения

В теле класса могут быть:

- – ***конструкторы и деструкторы*** – специальные методы, реализующие действия по инициализации объекта или класса и действия по их уничтожению;
- – ***индексаторы*** – обеспечивают возможность доступа к элементам класса по их порядковому номеру;
- – ***операции*** – задают действия с объектами с помощью знаков операций;
- – ***события*** – определяют уведомления, которые может генерировать класс;
- – ***типы*** – это типы данных, внутренние по отношению к классу.

```
class MyClass
        {
            public int x = 1;                       //поле данных
            public const double c = 7.18;           //константа
           public static string s = "Демонстрация";  // статическое поле класса
            double y;                               //закрытое поле данных
        }
    class Program
    {
        static void Main(string[ ] args)
        {
 MyClass a = new MyClass();// создание экземпляра класса MyClass
// a.x – обращение к полю класса  через имя экземпляра
            Console.WriteLine(a.x);
 // MyClass.c – обращение к константе всегда выполняется через имя класса
            Console.WriteLine(MyClass.c);
 // MyClass.s –  обращение к статическому полю через имя класса
            Console.WriteLine(MyClass.s);
        }
    }
```

МЕТОДЫ КЛАССОВ

***Метод* класса – это функция, объявленная внутри класса.**

Описание методов

```
заголовок_метода
 {
тело_метода
 }
```

Синтаксис заголовка метода:

```
[атрибуты][спецификаторы] тип_результата_метода
имя_метода    ([список_формальных_аргументов])
```

Имя метода и список формальных аргументов составляют *сигнатуру метода*.

Методу класса доступны все поля класса, их передавать в метод не нужно:

```
Class MyClass {
    int a;   // поле класса
public void MyProc()   // метод класса
{
   a=5;
 }
}
```

Вызов метода

Не *статические* методы вызываются для объекта класса:

имя объекта. имя метода (список параметров)

***Статические* методы (объявленные с модификатором static), можно вызывать, не создавая объекта класса:**

имя класса. имя метода (список параметров)

```
class Vector2d   {
        double x, y;
 //метод set_Vector для установки значений полей
        public void set_Vector(double x1, double y1)
        {
         x = x1;
         y = y1;
        }
// метод Summa для сложения векторов объявлен как статический
        public  static Vector2d Summa (Vector2d a, Vector2d b)
        {
           Vector2d z = new Vector2d();
           z.x = a.x + b.x;
           z.y = a.y + b.y;
           return z;
        }
// метод вывода полей класса
        public void Print()
        {
           Console.WriteLine("{0}  {1}", x, y);
        }
    }
```

```
static void Main(string[] args)
      {
// создание объектов a и b класса
          Vector2d a = new Vector2d();
          Vector2d b = new Vector2d();

// метод set_Vector вызывается для объектов класса
          a.set_Vector(1,2);
          b.set_Vector(2,3);

// статический метод Summa вызывается по имени класса
          Vector2d c = Vector2d.Summa(a, b);
          c.Print();
          }
```

Служебное слово this

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.

```
class TMan   {
// поля класса –  это переменные, обозначающие имя, фамилия, возраст человека
// все поля класса – закрытые, для работы  с ними следует использовать только методы класса
      string firstName;
       string lastName;
       int age;

// методы для доступа к данным
// метод setPeople, имена параметров функции совпадают с именами полей класса
      public void setPeople(string firstName, string lastName, int age)
      {
          this.firstName = firstName;
          this.lastName = lastName;
          this.age = age;
       }
// получение значения поля firstName
      public string GetFirstName()     { return firstName;  }
// получение значения поля lastName
      public string GetLastName()          { return lastName ; }
// получение значения поля age
      public int GetAge()                        {       return age;        }
   }
```

```
 static void Main(string[ ] args)
      {
// создание объекта peop  класса TMan
       TMan peop = new TMan();
Console.WriteLine("Значения полей объекта по умолчанию");
Console.WriteLine("{0}  {1}, {2}", peop.GetFirstName(),
    peop.GetLastName(), peop.GetAge());
// Передача полям значений через параметры функции setPeople, которая
// вызвана  для объекта peop
peop.setPeople("Иванов", "Иван", 18);
Console.WriteLine("Значения полей объекта после их
    установки");
Console.WriteLine("{0}  {1}, {2}",peop.GetFirstName(),
    peop.GetLastName(), peop.GetAge());
Console.ReadLine();
      }
```

Результаты работы этой программы имеют вид:

Значения полей объекта по умолчанию

, 0

Значения полей объекта после их установки

Иванов Иван, 18

Конструкторы классов

Конструкторы

Функция создания и инициализации объектов класса называется *конструктором*

Свойства конструкторов:

- **имеют то же имя, что и класс;**
- **не имеют объявлений возврата (даже void);**
- **не могут быть унаследованы, хотя производный класс может вызывать конструкторы базового класса;**
- **вызываются неявно, при создании или копировании объекта данного класса**

Общий синтаксис конструктора

```
спецификатор доступа имя класса
   ( список параметров)
    {
//   код конструктора
      }
```

Конструкторы без параметров

Если список параметров в определении конструктора отсутствует, то такой конструктор называется *конструктором без параметров*.

Если конструктор без параметров в классе явно не определен, то будет использоваться конструктор по умолчанию, созданный компилятором. Конструктор по умолчанию присваивает нулевые значения всем переменным объекта (для переменных обычных типов) и значения **null** (для переменных ссылочного типа), а строковые поля инициализирует пустой строкой. Если конструктор явно определен в классе, то конструктор по умолчанию использоваться не будет.

```
using System;
namespace demo_konstructor
{
   class MyClass  {
      public int x;
public MyClass()                    // конструктор класса без параметров
      {          x = 10;  }
   }
   class Program
   {
      static void Main(string[ ] args)
      {
// вызов конструктора без параметров при создании объектов t1 и t2
         MyClass tl = new MyClass();
         MyClass t2 = new MyClass();
         Console.WriteLine(tl.x + " " + t2.x);
      }
   }
}
```

Программа выведет значения:

10 10

Конструкторы с параметрами

Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации. Наличие в классе нескольких конструкторов называется *перегрузкой* конструкторов. Перегрузка конструкторов обеспечивает различную инициализацию полей при создании объектов.

```
using System.Text;
namespace demo_konstructor
{
    class MyClass     {
        public int x;
    // конструктор  без параметров
         public MyClass()                  { x = 10; }
    // конструктор с параметром
        public MyClass   ( int x)        {       this.x = i;     }
         }
    class Program
    {
        static void Main(string[] args)
        {
            MyClass tl = new MyClass();   // вызов конструктора без параметров
            MyClass t2 = new MyClass(68);    // вызов конструктора с параметрами
            Console.WriteLine(tl.x + " " + t2.x);
        }
    }
}
```

Программа выведет значения:

10 68

МЕТОДЫ-СВОЙСТВА КЛАССОВ

Методы-свойства

Методы, называемые свойствами (Properties), представляют специальную синтаксическую конструкцию, предназначенную для обеспечения эффективной работы с полями объекта.

Методы-свойства реализуют такой принцип ООП, как инкапсуляция.

Методы, выполняющие функции оболочки для доступа к полю обычно называют аксессорами (accessor).

Стратегии доступа к полям класса:

- чтение, запись (**Read, Write**);
- чтение, запись при первом обращении (**Read, Write-once**);
- только чтение (**Read-only**);
- только запись (**Write-only**);
- ни чтения, ни записи (**Not Read, Not Write**).

```
class TMan    {
        string firstName="“ ,   lastName="";
        int age=0;
public string FirstName   // стратегия: Read, Write-once (чтение, запись при первом обращении)
        {
    // служебное слово value идентифицирует новое значение, присваиваемое полю
            set    { if(firstName=="")    firstName = value;  }
          get { return (firstName); }
        }
public  string LastName  {// стратегия :Read, Write-once (чтение, запись при первом обращении)
    set {    if(lastName=="") lastName = value   }
            get {    return (lastName); }
          }
    public int Age    {   //стратегия: Read, Write (чтение, запись)
    set {   try          {
            if (value < 0 || value > 100)   // проверка значения возраста человека
    throw new ArgumentOutOfRangeException("value", value.ToString(), "Ошибка!\n Возраст должен
    быть больше 0 и меньше 100 лет");
                    age = value;
                }
                catch (Exception e)     {   Console.WriteLine(e.Message);  }
            }
            get { return (age); }
        }  }
```

```
static void Main(string[] args)
        {
TMan peop = new TMan(); // создание объекта класса
            peop.FirstName="Иванов";   // задать фамилию человека
            peop.LastName ="Иван";  // задать  имя человека
             peop.Age=18;  //задать  возраст человека
// вывод значений данных объекта
Console.WriteLine(" Фамилия ={0}, Имя=  {1},возраст= {2}", peop.FirstName, peop.LastName, peop.Age);
// попытка задать неверное значение возраста
// генерируется исключение ArgumentOutOfRangeException
             peop.Age = -3;
// неверное значение возраста установить не удалось, возраст не изменился!
Console.WriteLine(" Фамилия ={0}, Имя=  {1},возраст= {2}", peop.FirstName, peop.LastName,peop.Age );
            Console.ReadLine();
        }
```

```
Эта программа выводит следующее:
Фамилия: = Иванов,  Имя = Иван, возраст = 18
Ошибка!
Возраст должен быть больше 0 и меньше 100 лет
Parametr nane: value
Actual value was -3
Фамилия: = Иванов,  Имя = Иван, возраст = 18
```

СИНТАКСИС МЕТОДОВ-СВОЙСТВ

- **имя метода близко к имени поля** (например, имя поля - **firstName** , а имя поля – **FirstName)** ;
- **тело свойства** содержит два метода − **get** и **set**, один из которых может быть опущен.

Методы-свойства классов являются одним из средств реализации в языке C# такого принципа ООП, как инкапсуляция данных.

Параметры методов классов

Параметры методов классов

Передаваемое методу значение называется ***аргументом***.

Переменная внутри метода, которая принимает аргумент, называется ***параметром***.

Существуют два способа передачи параметров: ***по значению***

и по ссылке

ПЕРЕДАЧА ПАРАМЕТРОВ ПО ЗНАЧЕНИЮ

Метод получает **копии значений** аргументов и операторы метода работают с этими копиями.

Параметр-значение описывается в заголовке метода так:

тип имя

Передача по ссылке (по адресу)

Метод получает **копии адресов** аргументов

Параметр-ссылка описывается в заголовке метода с использованием служебного слова **ref**:

ref тип имя

```
• class Class1 {
// передача параметров по ссылке
    public static void Swap (ref int a, ref int b) {   int t;     t = a;   a = b;    b = t;   }

// передача параметров по значению
     public static void Swap1  (int a, int b)      {   int t;    t = a;   a = b;    b = t;   }
}
   class Program
   {
      static void Main(string[] args)
      {
         int a = 5, b = 10;
        Class1.Swap1(a, b);
Console.WriteLine("После вызова метода, получающего переменные по значению:");
         Console.WriteLine("a={0}  b={1}", a, b);
         Class1.Swap(ref a, ref b);
Console.WriteLine("После вызова метода, получающего переменные по ссылке:");
         Console.WriteLine("a={0}  b={1}", a, b);
             }
   }
```

Эта программа выводит следующее:
После вызова метода, получающего переменные по значению:
a=5 b=10
После вызова метода, получающего переменные по ссылке:
a=10 b=5

ВЫХОДНЫЕ ПАРАМЕТРЫ

Вернуть несколько значений из метода можно через его параметры, указав перед именем типа параметра служебное слово **out.**

Отличие **между модификаторами ref и out заключается в том, что параметр с модификатором out может использоваться только для передачи значения *из метода*.**

Пример использования модификатора out в методе класса

```
namespace demo_out
  {
 class Rectangle
 {
    int len1, len2;
    public Rectangle(int i, int j) // конструктор
    {
       len1 = i;    len2 = j;
    }
   // метод возвращает значение площади прямоугольника  и
  //определяет,  является ли он  квадратом
    public int rectInfo(out bool isSquare)
    {
       if (len1 == len2)   isSquare = true;
       else                isSquare = false;
       return len1 * len2;
    }
 }
```

Пример использования модификатора out в методе класса

```
class Program
    {
        static void Main(string[] args)
        {
            Rectangle rect = new Rectangle(25, 10);
            int area;
            bool isSqr;
            area = rect.rectInfo(out isSqr);
            if (isSqr)
Console.WriteLine("Данный прямоугольник является квадратом");
            else
Console.WriteLine("Данный прямоугольник не является квадратом");
            Console.WriteLine("Площадь прямоугольника="+ area+".");
            Console.ReadLine();
        }
    }
}
```

ПЕРЕГРУЗКА МЕТОДОВ

ПЕРЕГРУЗКА МЕТОДОВ

Перегрузка метода – это создание нового метода с именем уже существующего метода. Заголовок нового метода отличается от заголовка существующего метода только типом параметров и, возможно типом возвращаемого значения.

При перегрузке методов реализуется *статический способ связывания* метода с объектом. Статическое связывание выполняется на этапе компиляции программы, поиск вызываемого метода происходит по типу передаваемых в него аргументов.

Перегрузка методов является проявлением *полиморфизма.*

Пример перегрузки методов

```
namespace metod_Overload  {
// Класс C1 содержит объявление четырёх одноименных методов Fx
// с различными списками параметров.
   class C1   {
public float Fx(float key1)    // метод Fx с одним параметром типа float
        {   return key1;        }

public int Fx(int key1)      // метод Fx с одним параметром типа int
        {   return key1;        }

public int Fx(int key1, int key2)   // метод Fx с двумя параметром типа int
        {   return key1;        }

// метод Fx с двумя параметрами: типа byte и типа int
        public int Fx(byte key1, int key2)
        {
           return  (int) key1;
        }
    }
```

Пример перегрузки методов (продолжение)

```
class Program
  {
     static void Main(string[] args)
     {
        C1 c = new C1();
// вызов метода, получающего аргумент типа float
        Console.WriteLine(c.Fx(3.14F));
// вызов метода, получающего аргумент типа int
        Console.WriteLine( c.Fx(1));
// вызов метода, получающего два аргумента типа int
 Console.WriteLine(c.Fx(1, 2));
// вызов метода, получающего два аргумента: типа byte  и типа int
        Console.WriteLine( c.Fx((byte)10, 125));
     }
  }
}
```

Методы с переменным числом аргументов

Методы с переменным числом аргументов

Параметр метода может быть помечен служебным словом ***params***.

Параметр, помеченный этим словом, размещается в списке параметров последним и обозначает массив заданного типа неопределенной длины, например:

```
public int Calculate ( int a, out int c, params
int[ ] d)  {   }
```

В этот метод можно передать три и более аргументов. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество элементов массива получают с помощью свойства **Length.**

Методы с переменным числом аргументов (пример программы)

```
namespace demo_params  {
    class Class1  {
// объявление метода с переменным числом аргументов
        public void DisplayArrayOfInts(string msg, params int[ ] list)
{
Console.WriteLine(msg);
for (int i=0; i < list.Length; i++)
Console.WriteLine( list[i]);
}  }
    class Program  {
        static void Main(string[ ] args)
        {
            Class1 m = new Class1();   // создание объекта класса
            int[ ] IntArray = new int[3] {10,11,12};
// вызов метода с передачей ему массива значений из трех элементов
m.DisplayArrayOfInts ("Выведен массив целых", IntArray);
// вызов метода с передачей ему значений двух целых чисел
m.DisplayArrayOfInts("Выведено 2 целых числа", 1, 2);
//вызов метода с передачей ему значений пяти целых чисел
m.DisplayArrayOfInts("Выведено 5 целых чисел", 55, 4, 983, 10432,
98, 33);
        }
    }  }
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ

ПЕРЕГРУЗКА ОПЕРАЦИЙ

Выполняется с помощью ***методов-операторов*** класса. Правила:

– оператор должен быть описан как открытый статический метод класса (**public static**);

– параметры в метод-оператор должны передаваться по значению (то есть не должны предваряться ключевыми словами **ref** и **out**);

– сигнатуры всех операторов класса должны различаться;

– типы, используемые в операторе, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

ПЕРЕГРУЗКА ОПЕРАЦИЙ

Можно перегружать следующие операции:

– унарные **+, -, !, ~, ++, --, true, false**

– бинарные **+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=**.

ПЕРЕГРУЗКА ОПЕРАЦИЙ

При перегрузке унарных операций используется синтаксис

public static operator символ оператора (тип имя параметра)

{тело метода}

При перегрузке бинарных операций используется синтаксис

public static operator символ оператора (тип имя параметра1, тип имя параметра2)

{тело метода}

ПЕРЕГРУЗКА ОПЕРАЦИЙ (пример программы)

```
class Vector2d    {
      private double x, y;

public Vector2d() // конструктор по умолчанию
      {   x = 0;  y = 0; ; }

public Vector2d(double x, double y) // конструктор с параметрами
      {  this.x = x;  this.y = y; }

     // метод-оператор для перегрузки операции +
     public static Vector2d operator +(Vector2d a, Vector2d b)
      {
          Vector2d z = new Vector2d();
          return new Vector2d(a.x + b.x, a.y + b.y);
       }
     // метод вывода полей класса
     public void Print()
     {
        Console.WriteLine("{0}  {1}", x, y);
     }
  }
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ (пример программы, продолжение)

```
static void Main(string[] args)
        {
            Vector2d a = new Vector2d(1, 2);
            Vector2d b = new Vector2d(2, 3);
// вызов метода-оператора +
            Vector2d c = a + b;
            c.Print();
        }
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ. Рекомендации

1. Перегружать оператор следует только тогда, когда применение этого оператора будет иметь вполне определенный смысл в некотором классе.

2. Не все языки платформы **.NET** поддерживают саму концепцию перегрузки операторов. Поэтому при работе над реальным приложением всегда следует придерживаться следующего правила — помимо перегруженного оператора в нашем классе обязательно должен быть «обычный» метод, выполняющий те же самые функции.

3. Не все операции **C#** можно перегрузить, например нельзя перегрузить операцию []. Однако можно применять эту операцию к конкретному классу, если в нем реализован метод индексатора.

ИНДЕКСАТОРЫ

ИНДЕКСАТОРЫ

Индексаторы – члены класса, которые позволяют получить доступ к полям объекта способом доступа к массивам, то есть по индексу.

Синтаксис индексатора аналогичен синтаксису свойства:

```
[атрибуты] [спецификаторы]  тип this [список параметров]
  {
     get  { возврат значения элемента с указанными индексами}
     set  { присваивание значения элементу с указанными индексами }
   }
```

ИНДЕКСАТОРЫ (пример программы 1)

```
class ArrInt    {
      int[ ] arr;  // ссылка на массив
     int Length;      // количество элементов массива
public   bool flag ;   // flag=true, если индекс элемента массива выходит за допустимые пределы
public ArrInt(int size)    // в конструкторе создается массив заданного размера
      {  arr = new int[size] ;  Length = size;       }

public int this[int index]    {  // индексатор тип элемента - int
          get  {
             if (Ok(index)==true)         {    flag = false;   return arr[index];   }
             else                {    flag = true;  return 0;  }
           }
       set   {
             if (Ok(index) == true)         {    arr[index] = value;     flag = false; }
         else         flag =true;
           }
      }

private bool Ok(int index)   // возвращает true, если индекс находится в заданных границах
      {
        if (index >= 0 && index < Length)     return true;
        else   return false;
      }
       }
```

ИНДЕКСАТОРЫ (пример программы 1, продолжение)

```
class Program  {
        static void Main(string[] args)
        {
           ArrInt myarr = new ArrInt(5);
          for (int i = 0; i < 10; i++)
           {
              myarr[i] = i; // вызов из индексатора метода доступа set
// если ошибка доступа к элементу массива не возникла
              if (myarr.flag == false)
              {
// при обращении myarr[i] происходит вызов из индексатора метода доступа get
  Console.WriteLine("Элемент массива с индексом {0} равен {1}", i, myarr[i]) ;
              }
// индекс элемента массива выходит за допустимые пределы
              else
  Console.WriteLine("При обращении к элементу myarr[{0}] превышено
значение индекса массива", i);
           }
        }
     }
```

ИНДЕКСАТОРЫ (пример программы 2)

```
class Book
  {
      internal string Title, Autor;
    // конструктор
    public Book(string Title, string Autor)
    {
       this.Title = Title;
       this.Autor = Autor;
    }
  }
```

ИНДЕКСАТОРЫ (пример программы 2, продолжение)

```
class Books      {
// создание массива объектов типа Book
            Book[ ] bookArray = new Book[10];
public Book this[int pos]    //индексатор, тип элемента − Book
            {
          get
            {
            if (pos >= 0 || pos < 10) return bookArray[pos];
else throw new IndexOutOfRangeException("Вне диапазона");
            }
         set {    bookArray[pos] = value; }
            }
        }
```

ИНДЕКСАТОРЫ (пример программы 2, продолжение)

```
class Program
  {
     static void Main(string[] args)
     {
        Books BS = new Books();
// при установке значений объектов BS используется индексатор set
        BS[0] = new Book("Биллиг", "Основы C#");
         BS[1] = new Book("Троелсен", "C# и платформа .NET");
//  при обращении к объектам BS  используется индексатор get
        for (int i = 0; i < 2; i++)
  Console.WriteLine("Книга  {0}: {1}  {2}",  i,  BS[i].Autor,
    BS[i].Title);
        Console.ReadLine();
     }
  }
```

ВЛОЖЕННЫЕ КЛАССЫ

Вложенные классы

Класс, описанный в теле другого класса, называется вложенным (nested) классом.

```
// внешний класс
public class MyClass
 {
  // Члены внешнего класса
  ……….……
   //внутренний класс
   public class MyNestedClass
    {
      // Члены внутреннего класса
       ……..……..
    }
 }
```

Вложенные классы (пример программы)

```
public class ClassA  {    // внешний класс
    private class ClassB   //Вложенный класс
      {    public int z; }
       private ClassB   w; //Переменная типа вложенного класса
       public ClassA()   { //Конструктор
       w=new ClassB (); // создание экземпляра вложенного класса
       w.z=35; // обращение к полю вложенного класса через имя его экземпляра
    }
   public int SomeMethod()    //  доступ к экземпляру вложенного класса
     {  return w.z;    }
     }

  class Program    {
      static void Main(string[ ] args)      {
// в методе Main воможно создание экземпляра  только внешнего класса ClassA
ClassA v=new ClassA();
int k=v.SomeMethod() ;    // вызов метода внешнего класса
Console.WriteLine(k) ;
      }
  }
```

Вложенные классы (Рекомендации)

1. Вложенные классы могут объявляться и как **private**, и как **public**. Однако классы, которые объявлены в пространстве имен напрямую (то есть те классы, которые не вложены ни в какой другой класс), не могут быть объявлены как **private.**

2. Вложенный класс имеет смысл использовать тогда, когда его экземпляр используется только в определенном классе.

3. Вложенные классы улучшают читаемость кода — если нас не интересует устройство основного класса, то разбирать работу вложенного класса нет необходимости.