КЛАССЫ

Классы *(как и структуры)* предназначены для объявления **типов**.
Отличие классов от структур состоит в том, что классы между собой поддерживают отношения наследования.

Разновидностью класса являются **специализированные** типы

- интерфейсы;
- делегаты.

Ориентированы на расширение только **функциональности** разрабатываемого типа (*методы и свойства*), но при этом позволяют реализовать ***множественное*** наследование

На основе делегатов программируется **реакция** систем на разнообразные внешние воздействия или события

Объявление класса и создание его экземпляров

```
class A{}

A o1 = new A();
A o2 = new A();
A o3 = o2;
```

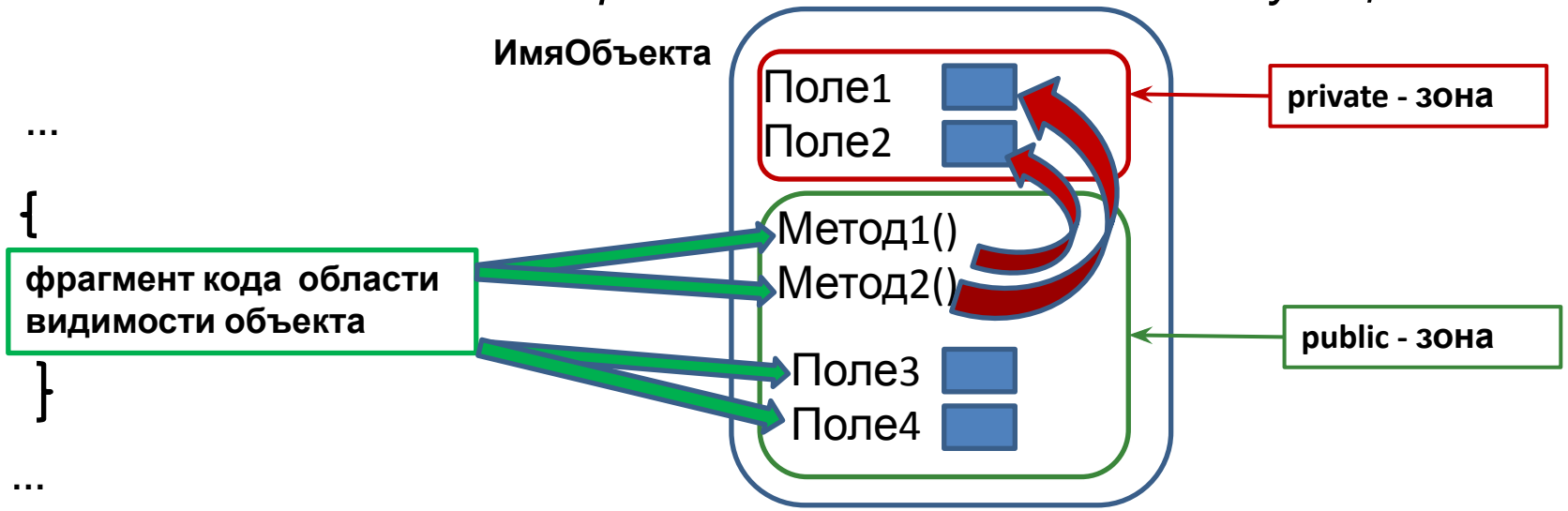
Элементы класса

Элементами класса могут быть ***поля, методы и события***.

Поля – пассивная, но при этом, ***основная*** часть класса *(предназначены* для ***размещения данных)***.

Методы – активная, но при этом ***вспомогательная*** часть класса: задача методов – ***обслуживание полей*** *(инициализация, модификация, представление и другие действия, связанные с обработкой информации)*.

Событие – это объект *типа* ***делегат***, предназначенный для размещения ссылок ***на метод*** *или* ***методы*** (*с помощью событий обеспечивается т.н. обратный вызов соответствующих методов*)



Методы *(как элементы класса)*

делятся на:

- ***особые*** (*Конструкт**оры*** *и* *Деструкт**ор***);
- ***неособые***.

Конструкторы предназначены для создания экземпляров класса *(могут быть перегружены)*. **Деструктор** предназначен для уничтожения экземпляра класса *(всегда один, не может быть перегружен)*.

Особые методы могут быть в режиме ***по умолчанию:***

Конструктор класса ***по умолчанию*** выполняет инициализацию полей объекта предустановленными значениями *по умолчанию*. Но уже **первый** явный конструктор фактом своего появления в классе **отменяет** конструктор ***по умолчанию***. Явный деструктор соответственно отменяет деструктор по умолчанию.

Объявление конструктора: модификатор **Имя** (сигнатура){тело}

- ***конструктор*** не может, *в принципе*, иметь **возвращаемого** значения, а, значит и не имеет **типа, ИМЯ** конструктора – это всегда **ИМЯ** класса,
- ***деструктор также*** *не имеет типа, но также деструктор не имеет и входных аргументов. Имя* ***деструктора*** *– это* ***имя*** класса с точностью до «тильда» (~).

И класс, и структура, могут иметь неограниченное количество перегруженных **конструкторов**.

Режимы защиты элементов класса

устанавливаются для каждого элемента класса модификаторами доступа:

- **public** – общедоступный элемент класса (нет защиты):
  доступен в любом месте области видимости объекта класса;

- **protected** – защищенный элемент класса:
  доступен только элементам данного классса и производного класса.
  *Примечание: спецификатора **protected** не используется в структурах, так как структурные типы **не поддерживают** наследования;*

- **private** – частный элемент: доступен только элементам данного класса.

При отсутствии у элемента класса (или структуры) явного модификатора
действует защита **private** *(режим «по умолчанию»)*

Объявление и реализация структуры и класса, конструкторы по умолчанию, поля

```
using System;
   class Primer
   {
      struct A
      {
         public   string      s;
         public          bool        a;
         public          int         b;
      }
      class B
      {
         public   string      s;
         public   bool        a;
         public   int    b;
      }
      static void Main()
      {    A  obj ;                    //Конструктор по умолчанию создаёт экземпляр
       obj.a = true;                  //структурного типа:  поля конструктором не   obj.b =
10;                  //инициализируются!
                obj.s = "Объект типа A";
                Console.WriteLine("{0} a={1}   b={2}", obj.s, obj.a, obj.b);
          B   obj1 = new B(); //Выделение памяти и вызов конструктора по
умолчанию   obj1.s = "Объект типа B";
```

В консольном окне:

```
Объект типа A a=True b=10
Объект типа B a=False b=0
```

Пример *(объявление класса и создание трёх его экземпляров)*

```
class A
   {   int a;
       public A( int ia)
       {
          a = ia;
       }
       public override string ToString()  // переопределение   метода ToString
       {
          return String.Format ( "a={0} ", a++ ) ;
       }
      static void Main()
       {
          A    o1 = new A ( 1 ),
               o2 = new A ( 1 ),
               o3 = o2;
          Console.WriteLine ( o1 );      //Использование переопределения
          Console.WriteLine ( o2 );
          Console.WriteLine ( o3 );
       }
   }
```

```
a=1
a=1
a=2
```

Краткая справка по **object**

object – это псевдоним для *супербазового* класса ***System.Object***.
Класс ***object*** имеет один конструктор и семь методов.

Статические методы:

Equals – проверка равенства двух экземпляров ***object***,
ReferenceEquals – проверка на совпадение двух экземпляров ***object,***

Нестатические методы:

GetType – возвращает тип экземпляра,
MemberwiseClone – выполняет поверхностное копирование текущего объекта,
Finalize – освобождение ресурсов или иная зачистка перед утилизацией объекта (***virtual*** *- виртуальный*),
GetHashCode – создание числа (хэш-кода), соответствующего значению объекта (***virtual***),
ToString – возвращение значения объекта в виде строки (***virtual***).

Как следует из модификаторов, методы ***Finalize, GetHashCode и ToString*** в пользовательских типах целесообразно переопределять

Конструкторы, конструкторы копии, неособые методы

Конструктором **копии** называется конструктор,
типом входного аргумента которого является **тот же самый** тип

```
class A1
 {  int A;                          // Частное поле
    public A1( int iA )                    //Конструктор
    { A = iA; }
    public void Type ( )                        //Неособый метод
                { Console.WriteLine(" A= {0}", A); }
    public void ClearA( )                  //Тоже неособый метод
                { A = 0; }
    public A1( A1 iA )                     //Конструктор копии
                { A = iA.A + 10;
                  iA.A = 100;              //оригинал тоже можно изменить!
                 }
    static void Main()
    {  A1 obj1 = new A1( 1 ),  obj2 = new A1( 2 ),   obj3 = obj2;
       obj1.Type( );
       obj2.Type( );
       obj3.Type( );
       obj3.ClearA( );
       obj2.Type( );
       A1 obj4 = new  A1( obj1 );              //Вызов конструктора копии
       obj4.Type( );
       obj1.Type( );
    }
 }
```

```
A= 1
A= 2
A= 2
A= 0
A= 11
A= 100
```

Ключевое слово **this**

- скрытая ***ссылка*** на объект в списке входных аргументов ***нестатических*** методов;
- вызов перегруженного конструктора (*до начала выполнения* ***данного*** *конструктора*);
- объявление и определение ***индексатора*** *(далее, в теме по свойствам);*

```
class A
 {  int  a;
    double  b;
    string   s;
    public   A ( int a, double b, string s)
        {   this.a = a;        //Первое применение this
            this.b = b;
            this.s = s;
            Console.WriteLine("Конструктор 1");
        }
    public  A ( string s) : this( 10, 20.5, s)          //Второе применение this
            {  Console.WriteLine("Конструктор 2"); }
    public override string ToString()
            { return String.Format("a={ 0 }, b= { 1 }, s ={ 2 }", a, b, s); }

    static void Main()
            {           A  o1 = new A (1, 2.5, " Первый объект" );
            A  o2 = new A ( " Второй объект" );
            Console.WriteLine ( o1 );
            Console.WriteLine ( o2 );
```

```
Конструктор 1
Конструктор 1
Конструктор 2
a=1, b= 2,5, s = Первый объект
a=10, b= 20,5, s = Второй объект
```

Деструктор

- имя ***совпадает*** с именем класса с точностью до **~**;
- ***не имеет типа*** и всегда ***пустой*** список входных аргументов;
- всегда **public**, момент вызова определяет ***Garbage Collector*** *(сборщик мусора)*

```
class A
    {
        string   S;
     public  A ( string s )
        {
          S = s;
          Console.WriteLine ("Создаю = " + s);    //s -  входной аргумент
         }

    ~A ( )  { Console.WriteLine ("Уничтожаю = " + S); }    // S  -  поле

    static void Main()
          {
            A object1 = new A ("первый"),  object2 = new A ("второй");
          }
     }
```

```
Создаю = первый
Создаю = второй
Уничтожаю = второй
Уничтожаю = первый
```

Свойство

- средство доступа к полю*(ям)* класса, по сути - ***упрощенный метод***;
- по режиму защиты могут быть ***public*** *(как правило)* или ***protected*** *(для возможности использования в производных классах);*
- не могут иметь тип ***void*** ;
- по применению похожи на ***объекты****;*
- могут иметь блоки **set -** для установки значения поля и **get -** для получения значения поля. Эти блоки называются ***акссессоры*** ;
- блок **get** всегда возвращает ***одно*** значение;
- блок **set** предназначен для установки значения(ий) поля(ей) имеет ***один*** скрытый аргумент с системным именем ***value****;*
- синтаксис объявления ***свойства:***

```
public    Тип    ИмяСвойства
     {
       get  // акссессор на чтение (закрытых) полей
     {// должен содержать хотя бы один   return   со значением Тип
     }
        set  // аксесссор для установки значений закрытого поля (полей)
     {// имеет одну    скрытую входную  переменную  value
     }
}
```

- имеется также ***автосвойство ,*** основную функциональность которого определяет система***;***

Свойства (пример)

```
class Complex
  { double  Real,   Image;                  //поля
    public double real                      //свойство
        {
          get { return Real; }
          set { Real = value; }
         }
    public double image                     //свойство
        {
          get { return Image; }
          set  { Image = 2 * value; }
         }
    public override string ToString()   { return String.Format("{0} + j {1}", real, image ); }
    public double modul                          // тоже  свойство
          {      get { return Math.Sqrt(Image*Image+Real*Real); }
          }
    static void Main()
          { Complex a = new Complex( ),  b = new Complex( );    //объявление экземпляров
            a.real = 1;                          // вызов  аксессора   set
            b.real = a.real + 2;                       // вызов  аксессора   get
            b.image = 4;
            a.image = b.image + 3;
            Console.WriteLine("Комплекс a:{ 0 }",a );
            Console.WriteLine("Комплекс b:{ 0 }",b );
            Console.WriteLine("Модуль a  = { 0:f4 }", a.modul );
            Console.WriteLine("Модуль b  = { 0:f4 }", b.modul );
          }
}
```

В консольном окне:

```
Комплекс a:1 + j 22
Комплекс b:3 + j 8
Модуль a  = 22,0227
Модуль b  = 8,5440
```

Автосвойство

```
class Complex
   {
     public double real  { get; set; }
     public double image   { get; set; }

     public override string ToString()  { return String.Format("{0} + j {1}", real, image ); }

     public double modul
                    {
            get {  return    Math.Sqrt(image*image+real*real );  }
                    }
    static void Main()
     {
       Complex a = new Complex() { real = 5, image = 6 };     //именованная инициализация
       Console.WriteLine("Комплекс  a:{ 0 }",  a);
       Console.WriteLine("Модуль a  = { 0:f4 }", a.modul );
     }
}
```

В консольном окне:

```
Комплекс a:5 + j 6
Модуль a  = 7,8102
```

Автосвойство:

- позволяет упростить код и использовать ***именованную*** инициализацию;
- его возможности сводятся только к ***установке*** и ***получению*** значений полей.

Индексатор

- разновидность ***свойства***, с помощью которого для ***классного*** типа можно перегрузить операцию ***«квадратные скобки»***;
- используется для типа, который содержит набор элементов, доступ к каждому из которых, удобно организовать по индексу;
- синтаксис объявления :

модификатор **Тип this [тип1 имя1, тип2 имя2] { аксессоры }**

```
class IndArray
{  int [ ]  arr;
   int  Len;
   public IndArray (  int   len )
      {     arr = new   int [  Len = len ];    }
   public int this [ int  ind ]    //одномерный индексатор
      {   set {  if ( ind  <  Len )  arr [  ind  ] = value; }
          get {  if ( ind  <  Len )  return arr [  ind  ] ; else return 0; }
       }
   static   void   Main( )
   {   IndArray   mass  =  new   IndArray( 2 );
      for ( int  i  = 0;  i <  4; i++)
          {   mass[  i  ] = i * 2 + 3;
            Console.WriteLine ( "mass[ { 0} ]  = { 1 } ", i, mass[ i ] );
          }
   }
}
```

```
mass[0] = 3
mass[1] = 5
mass[2] = 0
mass[3] = 0
```

Операторные методы *(перегрузка операций)*

Операторными методами можно перегрузить достаточно большое количество операций:

- все **унарные** (+, -, !, ~, ++, --);
- ***бинарные*** *(кроме логических «&&» (И) и «||» (ИЛИ))* ;
- операции ***true*** и ***false*** для использования в выражениях проверки *(условная операция, операторы* ***if-else****);*
- операции ***приведения*** типов.

Особенности операторных методов:

- операторный метод должен быть обязательно **public static**;
- операторный метод должен сохранять ***арность*** операции;
- один из входных аргументов операторного метода должен быть ***целевого*** типа *(того, для которого осуществляется перегрузка операции);*
- имя операторного метода образуется из слова ***operator*** и ***знака*** перегружаемой операции;
- операторный метод, как правило, не изменяет значение ***входного аргумента***;
- операторный метод обычно возвращает ***целевой*** тип

Операторные методы *(пример)*

```
class Complex
{ public double real  { get; set;}
  public double image   { get; set;}
  public override string ToString()
     { return String.Format("{0} + j {1}", real, image); }

public static Complex operator++ (Complex argin)
       {   Complex argout = new Complex();
           argout.real = argin.real + 1;
           argout.image = argin.image + 1;
           return argout;
       }

public static Complex operator+ (Complex arg1, Complex arg2)
         {   Complex argout= new Complex();
             argout.real = arg1.real + arg2.real;
             argout.image = arg1.image + arg2.image;
             return argout;
         }
 static void Main()
       {   Complex a = new Complex() { real = 5, image = 6 };
           Complex b = new Complex() { real = 7, image = 8 };
           a++;
           Complex c = a + b;
           Console.WriteLine("Комплекс a:{0}",a);
           Console.WriteLine("Комплекс b:{0}", b);
           Console.WriteLine("Комплекс c:{0}", c);
       }
}
```

```
Комплекс a:6 + j 7
Комплекс b:7 + j 8
Комплекс c:13 + j 15
```