# Интерфейсы

# Общие сведения об интерфейсе

- *Интерфейс* является «крайним случаем» абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализованы в производных классах.
- Интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами.
- Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом.
- Каждый класс может определять элементы интерфейса по-своему. Так достигается полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода.
- Синтаксис аналогичен синтаксису класса:

**[ атрибуты ] [ спецификаторы ] interface имя [ : предки ]**

**тело_интерфейса [ ; ]**

©Павловская Т.А.
(СПбГУ ИТМО)

# Общие сведения об интерфейсе

- Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае *предки* перечисляются через запятую.
- *Тело интерфейса* составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.
- Интерфейс **не может содержать** константы, поля, операции, конструкторы, деструкторы, типы и любые статические элементы.

```
interface IAction
{
    void Draw();
    int Attack(int a);
    void Die();
    int Power { get; }
}
```

# Интерфейсы или наследование классов?

- Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии.
- То, что работает в пределах иерархии одинаково, предпочтительно полностью определить в базовом классе.
- Интерфейсы чаще используются для задания общих свойств классов, относящихся к **различным** иерархиям.

# Отличия интерфейса от абстрактного класса

- элементы интерфейса по умолчанию имеют спецификатор доступа public и не могут иметь спецификаторов, заданных явным образом;
- интерфейс не может содержать полей и обычных методов — все элементы интерфейса должны быть абстрактными;
- класс, в списке предков которого задается интерфейс, должен определять *все* его элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным);
- класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

# Реализация интерфейса

- В C# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов, реализуя их по своему усмотрению.
- Сигнатуры методов в интерфейсе и реализации должны полностью совпадать.
- Для реализуемых элементов интерфейса в классе следует указывать спецификатор public.
- К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса.

©Павловская Т.А.
(СПбГУ ИТМО)

# Пример

```
interface Iaction
  {  void Draw();  int Attack( int a );  void Die();  int Power { get; }  }
class Monster : IAction
   {  public void Draw() { Console.WriteLine( "Здесь был " + name ); }
      public int Attack( int ammo_ )  {
          ammo -= ammo_;
          if ( ammo > 0 ) Console.WriteLine( "Ба-бах!" ); else  ammo = 0;
          return ammo;
      }
      public void Die()
         { Console.WriteLine( "Monster " + name + " RIP" ); health = 0; }
      public int Power { get  { return ammo * health; }
   }
```

```
Monster Vasia = new Monster( 50, 50, "Вася" );  // объект класса Monster
    Vasia.Draw();                                  // результат: Здесь был Вася
IAction Actor = new Monster( 10, 10, "Маша" );  // объект типа интерфейса
    Actor.Draw();                                  // результат: Здесь был Маша
```

# Обращение к реализованному методу через объект типа интерфейса

- Удобство этого способа проявляется при присваивании объектам типа IAction ссылок на объекты различных классов, поддерживающих этот интерфейс.
- Например, есть метод с параметром типа интерфейса. На место этого параметра можно передавать любой объект, реализующий интерфейс:

```
static void Act( IAction A )
{
     A.Draw();
}
static void Main()
{
    Monster Vasia = new Monster( 50, 50, "Вася" );
    Act( Vasia );
    ...
}
```

©Павловская Т.А.
(СПбГУ ИТМО)

# Второй способ реализации интерфейса

*Явное указание имени интерфейса* перед реализуемым элементом.

Спецификаторы доступа не указываются. К таким элементам можно обращаться в программе *только через объект типа интерфейса*:

```
class Monster : IAction {
    int IAction.Power { get{ return ammo * health;}}
    void IAction.Draw() {
        Console.WriteLine( "Здесь был " + name );    }            }
...
IAction Actor = new Monster( 10, 10, "Маша" );
Actor.Draw();                    // обращение через объект типа интерфейса
// Monster Vasia = new Monster( 50, 50, "Вася" );
// Vasia.Draw();                                          ошибка!
```

При этом соответствующий метод *не входит в интерфейс класса*. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользователю класса.

Кроме того, этот способ позволяет избежать конфликтов при множественном наследовании

©Павловская Т.А.
(СПбГУ ИТМО)

# Пример

Пусть класс Monster поддерживает два интерфейса: один для управления объектами, а другой для тестирования:

```
interface Itest { void Draw(); }
interface Iaction { void Draw(); int Attack( int a ); … }
class Monster : IAction, Itest {
    void ITest.Draw()  {
        Console.WriteLine( "Testing " + name );    }
    void IAction.Draw() {
        Console.WriteLine( "Здесь был " + name );    }
    ... }
```

Оба интерфейса содержат метод Draw с одной и той же сигнатурой. Различать их помогает явное указание имени интерфейса.

Обращаются к этим методам, используя **операцию приведения типа**:

```
Monster Vasia = new Monster( 50, 50, "Вася" );
((ITest)Vasia).Draw();                  // результат: Здесь был Вася
((IAction)Vasia).Draw();                // результат:   Testing Вася
```

# Операция is

- При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс.
- Проверка выполняется с помощью бинарной операции is. Она определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова is, с типом, заданным справа.
- Результат операции равен true, если объект можно преобразовать к заданному типу, и false в противном случае. Операция обычно используется в следующем контексте:

```
if ( объект is тип )
{
   // выполнить преобразование "объекта" к "типу"
   // выполнить действия с преобразованным объектом
}
```

# Операция as

- Операция as выполняет преобразование к заданному типу, а если это невозможно, формирует результат null:

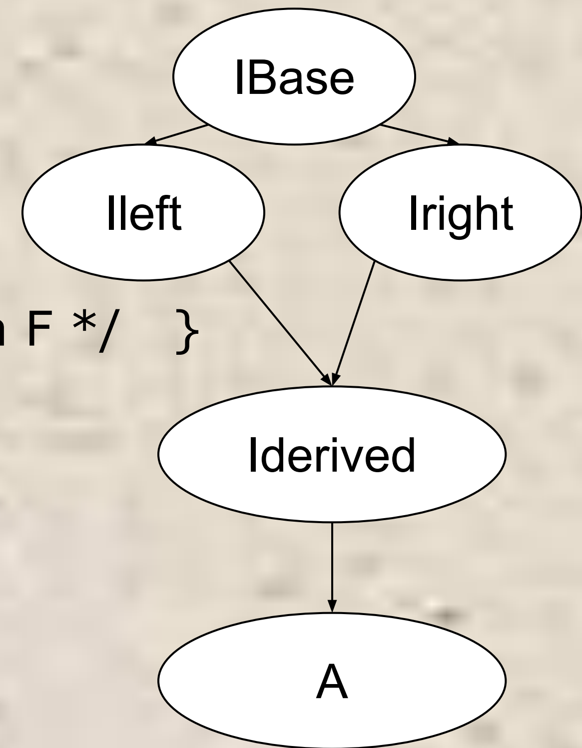
```
static void Act( object A )
{
    IAction Actor = A as IAction;
    if ( Actor != null ) Actor.Draw();
}
```

- Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

# Интерфейсы и наследование

- Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня.
- Базовые интерфейсы должны быть доступны в не меньшей степени, чем их потомки.
- Как и в обычной иерархии классов, базовые интерфейсы определяют общее поведение, а их потомки конкретизируют и дополняют его.
- В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово new, как и в аналогичной ситуации в классах. С помощью этого слова соответствующий элемент базового интерфейса скрывается.

# Пример переопределения




```
interface IBase  {  void F( int i );  }
interface Ileft : IBase {
   new void F( int i );  /* переопределение метода F */   }
interface Iright : IBase  {   void G();  }
interface Iderived : ILeft, IRight {}
class A  {
   void Test( IDerived d ) {
      d.F( 1 );                  // Вызывается ILeft.F
      ((IBase)d).F( 1 );     // Вызывается IBase.F
      ((ILeft)d).F( 1 );      // Вызывается ILeft.F
      ((IRight)d).F( 1 );    // Вызывается IBase.F
   }
}
```

Метод F из интерфейса IBase скрыт интерфейсом ILeft, несмотря на то, что в цепочке IDerived — IRight — IBase он не переопределялся.

# Особенности реализации интерфейсов

- Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные. Если при этом явно указывается имя интерфейса, оно должно ссылаться на тот интерфейс, в котором был описан соответствующий элемент.
- Интерфейс, на собственные или унаследованные элементы которого имеется явная ссылка, должен быть указан в списке предков класса.
- Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса.

# Стандартные интерфейсы .NET

- В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс **IComparable** задает метод сравнения объектов на «больше-меньше», что позволяет выполнять их сортировку.
- Реализация интерфейсов **IEnumerable** и **IEnumerator** дает возможность просматривать содержимое объекта с помощью foreach, а реализация интерфейса **ICloneable** — клонировать объекты.
- Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью foreach возможна оттого что тип Array реализует интерфейсы IEnumerable и IEnumerator.
- Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

# Сравнение объектов

- Интерфейс IComparable определен в пространстве имен System. Он содержит всего один метод CompareTo, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

```
interface IComparable
{
    int CompareTo( object obj )
}
```

- Метод должен возвращать:
  - 0, если текущий объект и параметр равны;
  - отрицательное число, если текущий объект меньше параметра;
  - положительное число, если текущий объект больше параметра.

©Павловская Т.А.
(СПбГУ ИТМО)

# Пример реализации интерфейса

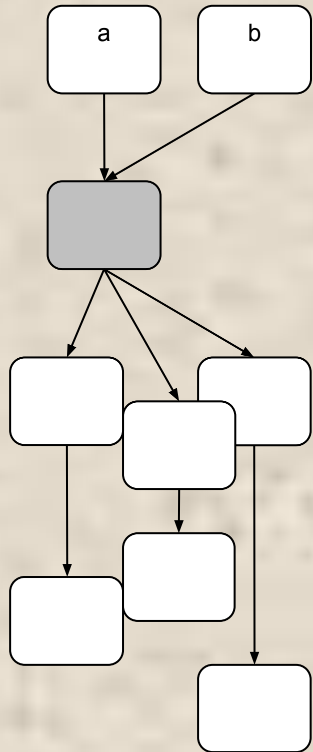
```
class Monster : IComparable
{ public int CompareTo( object obj )          // реализация интерфейса
      {  Monster temp = (Monster) obj;
         if ( this.health > temp.health ) return  1;
         if ( this.health < temp.health ) return -1;
         return 0;  }
   ... }
   class Class1
   {   static void Main()
      {  const int n = 3;
         Monster[] stado = new Monster[n];
         stado[0] = new Monster( 50, 50, "Вася" );
         stado[1] = new Monster( 80, 80, "Петя" );
         stado[2] = new Monster( 40, 10, "Маша" );
         Array.Sort( stado );            // сортировка стала возможной
}}
```

# Параметризованные интерфейсы

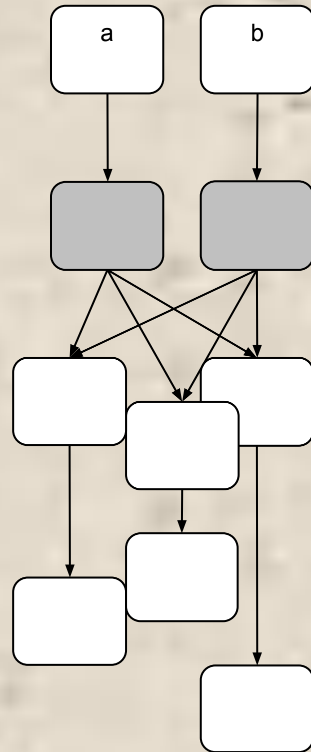
```
class Program {
        class Elem : IComparable<Elem>
        {  string data;
            int key;
            ...
            public int CompareTo( Elem obj )
            {   return key - obj.key;  }
        }
        static void Main(string[] args)
        {
            List<Elem> list = new List<Elem>();
            for ( int i = 0; i < 10; ++i ) list.Add( new Elem() ); ...
            list.Sort();
            ...
        }
    }
```

# Клонирование объектов

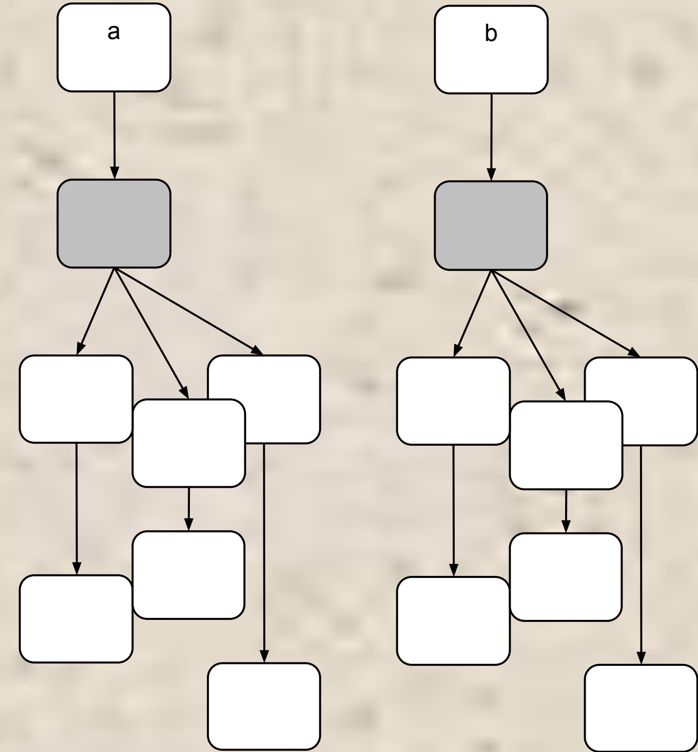
- *Клонирование* — создание копии объекта. Копия объекта называется клоном.




а) присваивание b = a




б) поверхностное клонирование




в) глубокое клонирование

©Павловская Т.А. (СПбГУ ИТМО)

# Виды клонирования

- При присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект (рис. *а*).
- Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом MemberwiseClone, который объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются (рис. *б*). Это называется *поверхностным клонированием*.
- Для создания полностью независимых объектов необходимо *глубокое клонирование*, когда в памяти создается дубликат всего дерева объектов (рис. *в*).
- Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.
- Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса **ICloneable** и переопределять его единственный метод **Clone**.

©Павловская Т.А.
(СПбГУ ИТМО)

# Структуры

# Определение структуры

*Структура* — тип данных, аналогичный классу, отличия:

- является *значимым*, а не ссылочным типом данных;
- не может участвовать в иерархиях наследования, может только реализовывать интерфейсы;
- в структуре запрещено определять конструктор по умолчанию, поскольку он определен неявно и присваивает всем ее элементам нули соответствующего типа;
- в структуре запрещено определять деструкторы, поскольку это бессмысленно.

*Область применения структур*: типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками (снижаются накладные расходы на динамическое выделение памяти)

©Павловская Т.А. (СПбГУ ИТМО)

# Синтаксис структуры

[ атрибуты ] [ спецификаторы ] **struct** имя_структуры [ : интерфейсы ]

тело_структуры [ ; ]

- *Спецификаторы* доступа - public, internal и private (последний — только для вложенных структур).
- *Интерфейсы*, реализуемые структурой, перечисляются через запятую.
- *Тело структуры* может состоять из констант, полей, методов, свойств, событий, индексаторов, операций, конструкторов и вложенных типов.

# Пример структуры

```
struct Complex
  {  public double re, im;
     public Complex( double re_, double im_ )
     {     re = re_; im = im_;  }
     public static Complex operator + ( Complex a, Complex b )
     {    return new Complex( a.re + b.re, a.im + b.im );         }
     public override string ToString()
     { return ( string.Format( "({0,2:0.##}; {1,2:0.##})", re, im ) );
     }
  }
class Class1
  {   static void Main()
     { Complex a = new Complex( 1.2345, 5.6 );
        Console.WriteLine( "a = " + a );
        Complex [] mas = new Complex[4]; …
 }}
```

Результат работы программы:
a = (1,23; 5,6)

# Описание элементов структур

- поскольку структуры не могут участвовать в иерархиях, для их элементов не могут использоваться спецификаторы protected и protected internal;
- структуры не могут быть абстрактными (abstract), к тому же по умолчанию они бесплодны (sealed);
- методы структур не могут быть абстрактными и виртуальными;
- переопределяться (то есть описываться со спецификатором override) могут только методы, унаследованные от базового класса object;
- параметр this интерпретируется как значение, поэтому его можно использовать для ссылок, но не для присваивания;
- при описании структуры нельзя задавать значения полей по умолчанию.

©Павловская Т.А.
(СПбГУ ИТМО)

# Перечисления

# Определение перечисления

Перечисление – набор связанных констант:

```
enum Menu { Read, Write, Append, Exit };

enum Радуга { Красный, Оранжевый, Желтый, Зеленый, Синий,
    Фиолетовый };

enum Nums { two = 2, three, four, ten = 10, eleven, fifty = ten + 40 };

enum Flags : byte

{    b0, b1, b2, b3 = 0x04, b4 = 0x08, b5 = 0x10, b6 = 0x20, b7 = 0x40    };
```

- Имена перечисляемых констант внутри каждого перечисления должны быть уникальными, а значения могут совпадать.
- Все перечисления являются потомками базового класса System.Enum

Преимущества перечислений перед описанием именованных констант:

- связанные константы нагляднее;
- компилятор выполняет проверку типов;
- интегрированная среда разработки подсказывает возможные значения констант.

С переменными перечисляемого типа можно выполнять:

- арифметические операции (+, –, ++, ––),
- логические поразрядные операции (^, &, |, ~),
- сравнения (<, <=, >, >=, ==, !=)
- получать размер в байтах (sizeof).

©Павловская Т.А.
(СПбГУ ИТМО)

- При использовании переменных перечисляемого типа в целочисленных выражениях и операциях присваивания требуется явное *преобразование типа*.
- Переменной перечисляемого типа можно присвоить любое значение, представимое с помощью базового типа.

```
enum Flags : byte
{  b0, b1, b2, b3 = 0x04, b4 = 0x08, b5 = 0x10, b6 = 0x20, b7 = 0x40    };
Flags a = Flags.b2 | Flags.b4;
++a;
int x = (int) a;
Flags b = (Flags) 65;
```

©Павловская Т.А.
(СПбГУ ИТМО)

# Делегаты

# Определение делегата

- *Делегат* — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод.
- Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка.
- Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

**[ атрибуты ] [ спецификаторы ] delegate тип имя([ параметры ])**

Пример описания делегата:

```
public delegate void D ( int i );
```

- Базовым классом делегата является класс System.Delegate

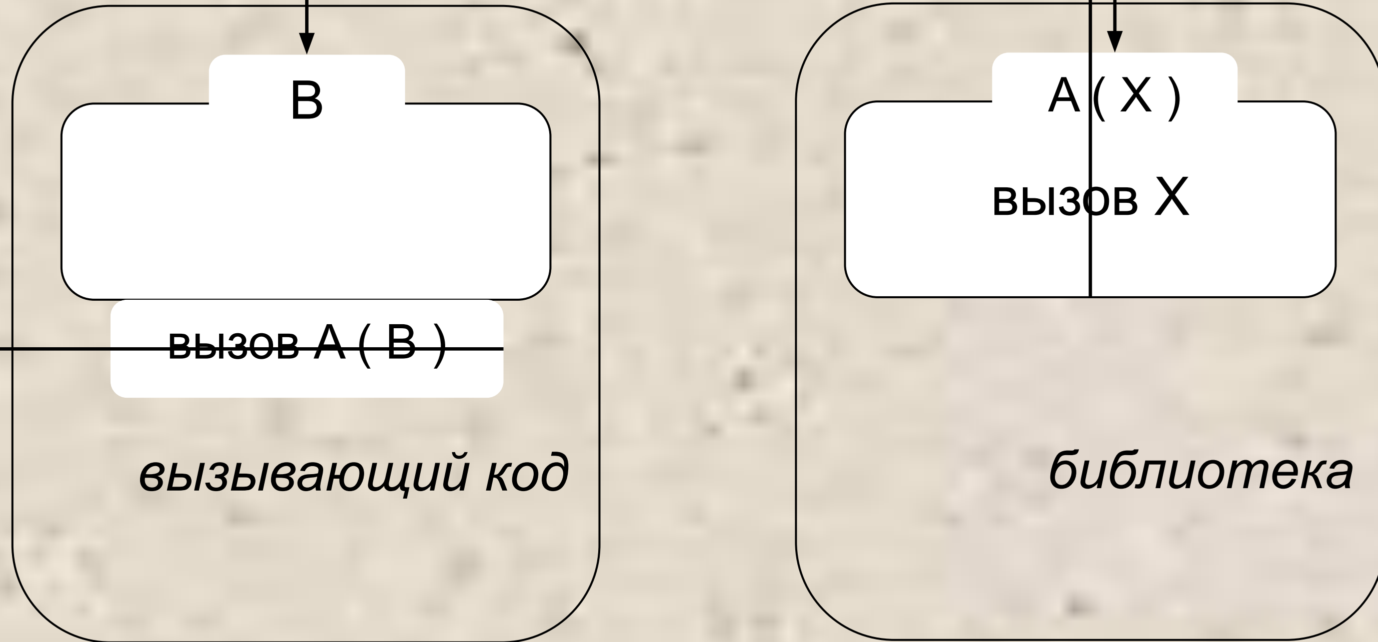
©Павловская Т.А.
(СПбГУ ИТМО)

# Использование делегатов

- Делегаты применяются в основном для следующих целей:
  - получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
  - обеспечения связи между объектами по типу «источник — наблюдатель»;
  - создания универсальных методов, в которые можно передавать другие методы (поддержки механизма обратных вызовов).

©Павловская Т.А. (СПбГУ ИТМО)

# Обратный вызов (callback)



# Передача делегата через список параметров

```
namespace ConsoleApplication1 {
    public delegate double Fun( double x );      // объявление делегата
    class Class1 {
        public static void Table( Fun F, double x, double b )
        {   Console.WriteLine( " ----- X ----- Y -----" );
            while (x <= b)
            {  Console.WriteLine( "| {0,8} | {1,8} |", x, F(x));
               x += 1; }
        }
        public static double Simple( double x ) { return 1; }

        static void Main()
        { Table( Simple, 0, 3 );
           Table( Math.Sin, -2, 2 );                    // new Fun(Math.Sin)
           Table( delegate (double x ){ return 1; }, 0, 3 );
}}}
```

©Павловская Т.А.
(СПбГУ ИТМО)

# Операции

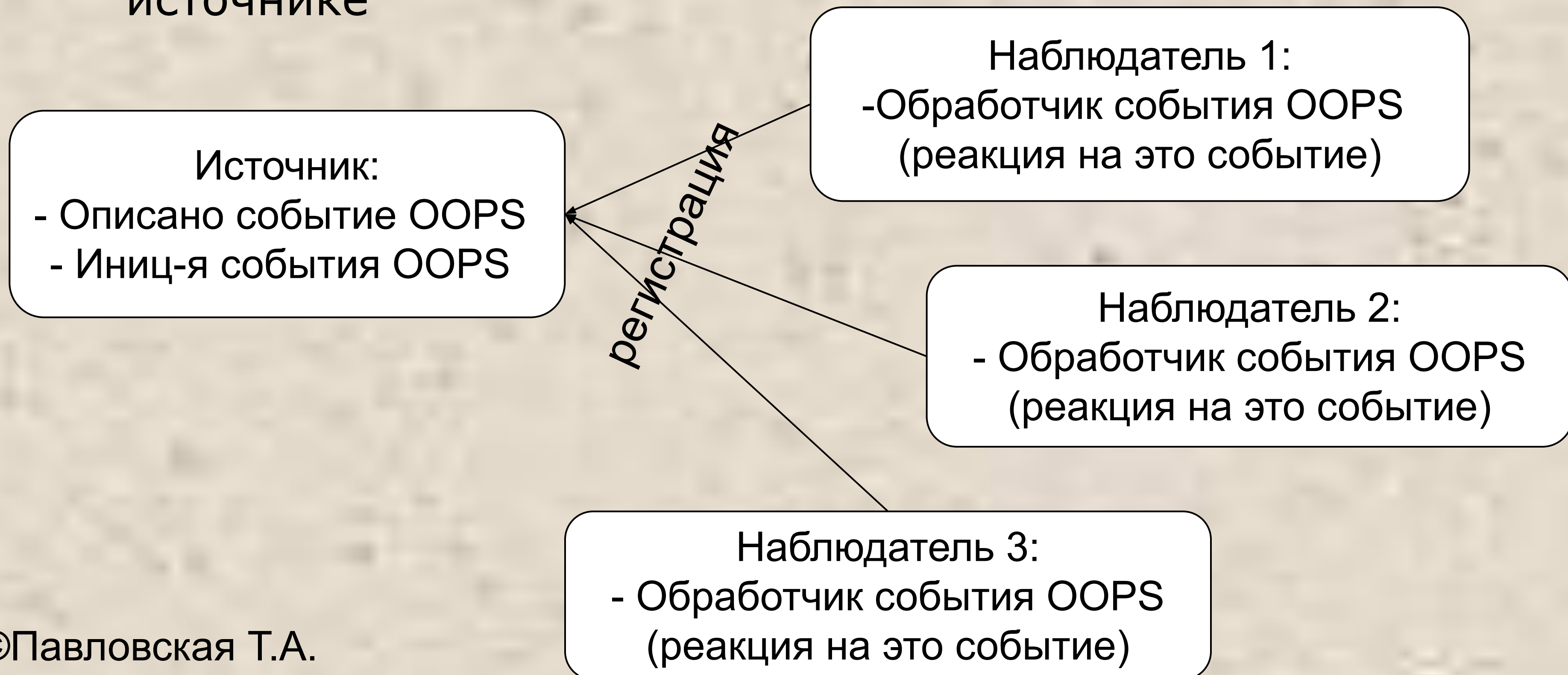
- Делегаты можно *сравнивать на равенство и неравенство*. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке.
- С делегатами одного типа можно *выполнять операции* простого и сложного *присваивания*.
- Делегат, как и строка string, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.
- Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

©Павловская Т.А. (СПбГУ ИТМО)

# События

# Определение события

- *Событие* — элемент класса, позволяющий ему посылать другим объектам (наблюдателям) уведомления об изменении своего состояния.
- Чтобы стать наблюдателем, объект должен иметь обработчик события и зарегистрировать его в объекте-источнике



# Пример

```
class Subj  {          // -------------- Класс-источник события ---------------------
      public event EventHandler Oops;     // Описание события станд. типа
      public void CryOops() {                  // Метод, инициирующий событие
            Console.WriteLine( "OOPS!" );  if ( Oops != null ) Oops( this, null ); }
   }
class Obs     {        // --------------- Класс-наблюдатель --------------------------
      public void OnOops( object sender, EventArgs e ) {   // Обработчик соб-я
            Console.WriteLine( «Оййй!" );
      }
   }
class Class1 {
      static void Main()        {
         Subj s  = new Subj();
         Obs o1 = new Obs();  Obs o2 = new Obs();
         s.Oops += o1.OnOops;                          // регистрация обработчика
         s.Oops += o2.OnOops;                          // регистрация обработчика
         s.CryOops();
      }
   }
```

```
OOPS!
Оййй!
Оййй!
```

# Механизм событий

- События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:
  - описание делегата, задающего сигнатуру обработчиков событий;
  - описание события;
  - описание метода (методов), инициирующих событие.
- Синтаксис события:

**[ атрибуты ] [ спецификаторы ] event тип имя**

# Пример

```
public delegate void Del( object o );     // объявление делегата
class A
{
    public event Del Oops;                 // объявление события
    ...
}
```

- *Обработка событий* выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.
- Событие — это удобная абстракция для программиста. На самом деле оно состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.
- Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции += и -=. Тип результата этих операций — void, в отличие от операций сложного присваивания для арифметических типов. Иного способа доступа к списку обработчиков нет.

# Пример

```
public delegate void Del();                    // объявление делегата
    class Subj                                 // класс-источник
    {  public event Del Oops;                  // объявление события
        public void CryOops()                  // метод, инициирующий событие
        {   Console.WriteLine( "OOPS!" ); if ( Oops != null ) Oops();      }
    }
    class ObsA                                 // класс-наблюдатель
    {
        public void Do();                      // реакция на событие источника
        {  Console.WriteLine( "Вижу, что OOPS!" );  }
    }
    class ObsB                                 // класс-наблюдатель
    {
        public static void See()               // реакция на событие источника
        {  Console.WriteLine( "Я тоже вижу, что OOPS!" );   }
    }
```

©Павловская Т.А.
(СПбГУ ИТМО)

```
class Class1
    {
        static void Main()
        {
            Subj s  = new Subj();                //    объект класса-источника
            ObsA o1 = new ObsA();                //  объекты
            ObsA o2 = new ObsA();                //  класса-наблюдателя
            s.Oops += new Del( o1.Do );          //              добавление
            s.Oops += new Del( o2.Do );          //              обработчиков
            s.Oops += new Del( ObsB.See );       //              к событию
            s.CryOops();                         //      инициирование события
        }
    }
```

©Павловская Т.А.
(СПбГУ ИТМО)

# Еще немного про делегаты и события

- Делегат можно вызвать асинхронно (в отдельном потоке), при этом в исходном потоке можно продолжать вычисления.
- Анонимный делегат (без создания класса-наблюдателя):

```
s.Oops += delegate ( object sender, EventArgs e )
            { Console.WriteLine( "Я с вами!" ); };
```

- Делегаты и события обеспечивают гибкое взаимодействие взаимосвязанных объектов, позволяющее поддерживать их согласованное состояние.
- События включены во многие стандартные классы .NET, например, в классы пространства имен Windows.Forms, используемые для разработки Windows-приложений.