

Делегаты и события



Делегаты

- ▶ это объект, предназначенный для хранения ссылок на методы(указатель на функцию C++)
- ▶ функции обратного вызова + без. типов

- 1) используются для поддержки событий
- 2) как самостоятельная конструкция языка

[атрибуты] [спецификаторы]

System.Delegate

`delegate` тип `имя_делегата`([параметры])

System.MulticastDelegate

`new`, `public`, `protected`, `internal` и `private`.

- Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать

Делегат может служить для вызова любого метода с соответствующей сигнатурой и возвращаемым типом.

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

Определяется класс компилятором, содержит четыре метода: конструктор, методы Invoke, BeginInvoke и EndInvoke

← прототип

← асинхронный обратный вызов

+

1. **_target** - ссылается на объект.
2. **_methodptr** - определение метода, который надлежит вызвать обратно.
3. **_invocationlist** может ссылаться на массив делегатов при строительстве цепочки делегатов.



... .class nested private auto ansi sealed

... extends [mscorlib]System.MulticastDelegate

... .ctor : void(object,native int)

... BeginInvoke : class [mscorlib]System.IAsyncResult(int32,class [mscorlib]System.AsyncCallback,ob

... EndInvoke : void(class [mscorlib]System.IAsyncResult)

... Invoke : void(int32)

► Использование делегатов

```
public delegate void D(int i);  
class Class1  
{  
    private static void HelloI(int i) { }  
    static void Main()  
    {  
        D del;  
        del = HelloI;  
        del.Invoke(4);  
    }  
}
```

Объявляем делегат

Или так

Создаем переменную делегата

Присваиваем этой переменной адрес метода

Вызываем метод

`del = new D(HelloI);`
`del(4);`

Чтобы использовать делегат, нам надо создать его объект с помощью конструктора, в который мы передаем адрес метода, вызываемого делегатом

Свойства

- ▶ Тип данных
- ▶ Наследовать от делегата нельзя
- ▶ Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса (в любом месте, где может быть определен класс)

```
public delegate void D(int i);  
  
class GGG  
{  
    public delegate void GGGD(int i);  
}
```

- ▶ может вызывать только такие методы, у которых тип возвращаемого значения и список параметров совпадают
- ▶ Может быть статический метод класса
- ▶ Имеет тот же синтаксис, что и вызов метод
- ▶ Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат (цепочки (chaining))

- ▶ делегаты могут выполняться в асинхронном режиме, при этом не надо создавать второй поток, надо вместо метода Invoke использовать пару методов **BeginInvoke/EndInvoke**
- ▶ делегат можно вызывать как обычный метод
- ▶ делегаты могут быть параметрами методов

```
public delegate void D(int i);  
    class Class1  
    {  
        public void ReadDelegat (D _del) { _del(1); }  
    }
```

Передача делегатов в методы

```
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("| {0,8:0.000} | {1,8:0.000} |", x, F(x));
            x += 1;
        }
        Console.WriteLine(" ");
    }
}

static void Main()
{
    Table(new Fun(Math.Sin), -2, 2);
}

}
```

создает экземпляр делегата

Назначение делегатов

- ▶ 1) возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- ▶ 2) обеспечения связи между объектами по типу «источник — наблюдатель»;
- ▶ 3) создания универсальных методов, в которые можно передавать другие методы;
- ▶ 4) поддержки механизма обратных вызовов.

Пример

```
delegate string strMod(string stx);

class DelegateTest
{
    static string replaceSpaces(string a) { Console.WriteLine("Замена"); return a; }
    static string removeSpaces(string a) { Console.WriteLine("Удаление"); return a; }
    static string reverse(string a) { Console.WriteLine("Реверс"); return a; }

    public static void Main()
    {
        // ...
        public static void Main()
        {
            strMod strOp = replaceSpaces;
            string str; str = strOp("ЭТО простой тест.");
            strOp = DelegateTest.removeSpaces;
            str = strOp("Это простой тест.");
        }
    }
}
```

C:\windows\system32\cmd.exe

Замена
Удаление
Для продолжения нажмите любую клавишу .

Операции над делегатами

- ▶ можно *сравнивать на равенство и неравенство* (не содержат ссылок или если ссылки на одни и те же методы в одном и том же порядке)
- ▶ *выполнять операции простого и составного присваивания* (один тип д.и.)

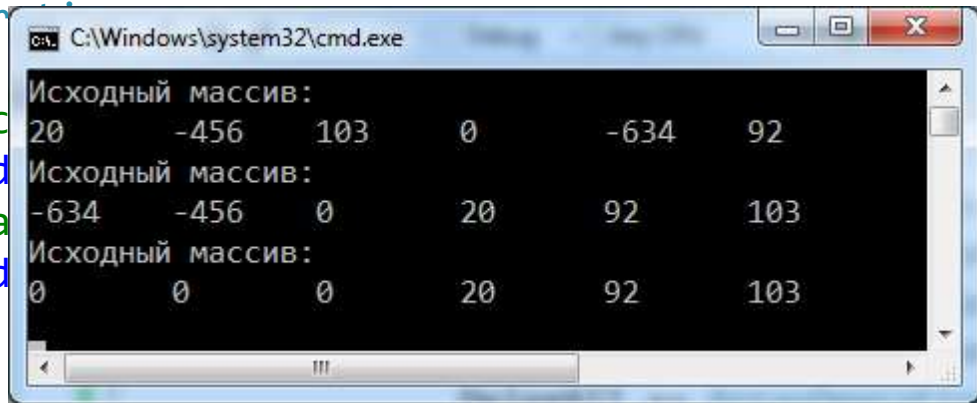
```
del += HelloI; // добавляем делегат  
del -= HelloI; // удаляем делегат
```

- ▶ является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

Групповая адресация

Создание списка, или цепочки вызовов, для методов, которые вызываются автоматически при обращении к делегату

```
delegate void OperationWithArray(ref int[] arr);
public class ArrayOperation
{
    public static void
        // Сортировка массива
        public static void
        // Заменяем отрицательные значения на положительные
        public static void
    }
class Program
{
    static void Main()
    {
        int[] someArr = new int[] { 20, -456, 103, 0, -634, 92 };
        // Структурируем делегаты
        OperationWithArray DelegAll; // Групповая адресация
        DelegAll = ArrayOperation.WriteArray;
        DelegAll += ArrayOperation.IncSort;
        DelegAll += ArrayOperation.WriteArray;
        DelegAll += ArrayOperation.NegatArr;
        DelegAll += ArrayOperation.WriteArray;
        // Выполняем делегат
        DelegAll(ref someArr);
        Console.ReadLine();
    }
}
```



Анонимные функции

- ▶ представляет собой безымянный кодовый блок, передаваемый конструктору делегата
 - Анонимные методы
 - Лямбда - выражения

```
delegate int Summator(int b);
```

```
static int result = 0;  
Summator someDelegat = delegate (int number)  
{  
    for (int i = 0; i <= number; i++)  
        result += i; //захват переменной  
    return result;  
};
```



метод, встроенный в код

- ▶ Параметры должны соответствовать параметрам делегата
- ▶ может не содержать никаких параметров
- ▶ метод имеет доступ ко всем переменным, определенным во внешнем коде



Лямбда-выражения

- ▶ упрощенная запись анонимных методов

параметр => выражение
(список_параметров) => выражение

```
(x, y) => x + y;
```

```
i => i * i;
```

не надо указывать тип параметров, не
надо использовать оператор return

Лямбда-выражения

//объявление

```
        delegate int  FindMax(int s1, int s2);  
    private delegate bool isNegate(int a);  
static void Main()  
{
```

//определение

```
    FindMax maxint = (s1, s2) => s1 > s2 ? s1 : s2;  
    isNegate nega = s3 => s3 < 0;
```

//вызов

```
        int a = 10;  
        int b = -8;  
        maxint(a, b);  
        nega(b);
```

упрощенная запись
анонимных методов

Блочные лямбда-выражения

```
//объявление
private delegate bool isOdd(int a);

static void Main()
{
    //определение
    isOdd testIsOdd = s4 =>
    {
        if (s4%2 == 0) return true;
        else return false;
    };
    //вызов
    int a = 10;
    testIsOdd(a);
}
```

► Список параметров может быть пустым

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

```
class Class1
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        D del = () => { Console.WriteLine("Hello"); };  
        del();
```

```
    }
```

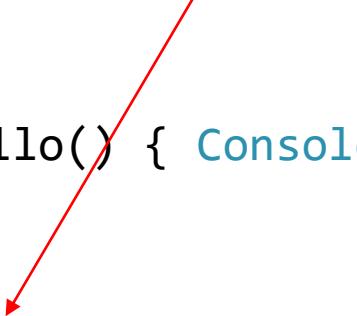
```
}
```

- ▶ можно передавать в качестве аргументов методу
- ▶ может принимать ссылку на метод

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

```
class Class1
{
    private static void SayHello() { Console.WriteLine("Hello"); }

    static void Main()
    {
        D del = () => SayHello();
        del();
    }
}
```



лямбда-выражения можно передавать в качестве параметров методу

Обобщённые делегаты .NET

Action<T> и Func<T>, Predicate<T>

Определённые в пространстве имен System

- ▶ void Action<in T1, in T2, in T3 ...in T16 >
- ▶ TResult Func<out TResult> До 16 параметров
- ▶ TResult Func<in T1,.....T16, out TResult>

Вместо определения собственных типов делегатов рекомендуется по мере возможности использовать обобщённые делегаты

```
static public void Sort<T>  
    (IList<T> sortArray,  
     Func<T, T, bool> res) { ....}
```

Action

```
public delegate void Action<T>(T obj)
```

```
class Test
{
    static void Operation(int x1, int x2, Action<int, int> op)
        => op(x1, x2);

    static void Main(string[] args)
    {
        Action<int, int> op;
        op = (int a, int b) => {int c = a + b; };
        Operation(10, 6, op);
    }
}
```

предусматривает вызов определенных действий

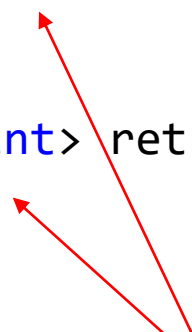
Два параметра

Func

Он возвращает результат действия
и может принимать параметры

```
class Test
{
    static void Main(string[] args)
    {
        Console.WriteLine(Operation(6, x => x * x)); // 36
        Console.WriteLine(Operation(10, x => x / 2)); // 5
    }

    static int Operation(int x1, Func<int, int> retF)
    => x1 < 0 ? 0: retF(x1);
}
```



принимает число int и возвращает int

События

- ▶ это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния

модель «публикация — подписка» или паттерн «наблюдатель»,

класс Button - событие Click

класс, являющийся отправителем (sender) сообщения, публикует события, которые он может инициировать,

а другие классы, являющиеся получателями (receivers) сообщения, подписываются на получение этих событий.

5)Обработка

Почтовый клиент

1) объект регистрируется
в качестве получателя
уведомлений о событии Mail

4)Уведомляем объект
о событии

Почтовый
Менеджер

Событие
- Mail

2) «знает», что
объект следует
уведомить о
сообщении

3)Пришло сообщение
Наступило событие



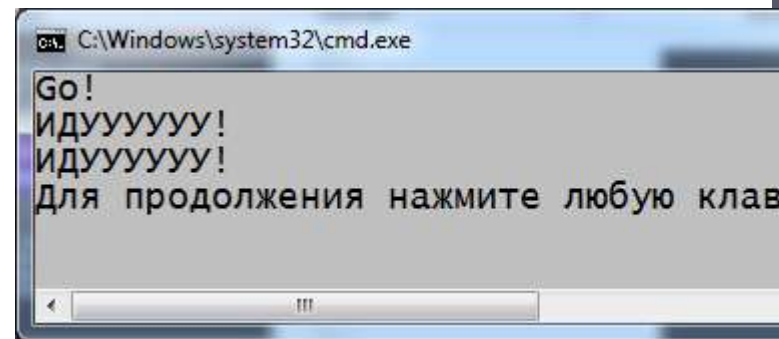

```

class Mind          // ---- Класс-источник события -----
{
    public event EventHandler Go; // Описание соб. станд. типа
    public void CommandGo() // Метод, инициирующий событие
    {
        Console.WriteLine("Go!");
        if (Go != null)
            Go(this, null);
    }
}

class Leg          // ----- Класс-наблюдатель -----
{
    public void OnGo(object sender, EventArgs e)
    {
        Console.WriteLine("ИДУУУУУУ!"); // Обработчик соб-я
    }
}

class Class1
{
    static void Main()
    {
        Mind s = new Mind();
        Leg o1 = new Leg();
        Leg o2 = new Leg();
        Leg o3 = new Leg();
        s.Go += o1.OnGo;
        s.Go += o2.OnGo;
        s.CommandGo();
    }
}

```



```

// регистрация обработчика
// регистрация обработчика

```

```

class Mind                // ---- Класс-источник события -----
{
    public event EventHandler Go; // Описание соб. станд. типа
    public void CommandGo() // Метод, инициирующий событие
    {
        Console.WriteLine("Go!");
        if (Go != null)
            Go(this, null);
    }
}

class Leg                // ----- Класс-наблюдатель -----
{
    public void OnGo(object sender, EventArgs e)
    {
        Console.WriteLine("ИДУУУУУУ!"); // Обработчик соб-я
    }
}

class Class1
{
    static void Main()
    {
        Mind s = new Mind();
        Leg o1 = new Leg();
        Leg o2 = new Leg();
        Leg o3 = new Leg();
        s.Go += o1.OnGo; // регистрация обработчика
        s.Go += o2.OnGo; // регистрация обработчика
        s.CommandGo();
    }
}

```

Значение события по умолчанию — null

тип результата void

источник события - имеет тип object
аргументы события и имеет тип EventArgs
или производный от него

В библиотеке .NET описано много стандартных делегатов,
предназначенных для реализации механизма обработки событий

закрытый статический класс, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата

[атрибуты] [спецификаторы]
event имяделегата имясобытия

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры и возвращать тот же тип, что и делегат.

- 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.
- new, public, protected, internal, private, static, virtual, sealed, override, abstract и extern

```
public delegate void Doing(object o); // объявление делегата

class A
{
    public event Doing Oppa; // объявление события
}
```

- 1) Обработка событий выполняется в классах-получателях
- 2) сигнатура методов-обработчиков событий, == типу делегата
- 3) Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод :+= и -=
методы add_....., remove_
- 4) Поддерживается групповая адресация

```
public delegate void SomeDelegat();
```

```
class Mind
```

```
{    public event SomeDelegat Go;
    public void ComandGo()
    {    Console.WriteLine("Go!");
        if (Go != null)
            Go();
    }
}
```

```
class Leg
```

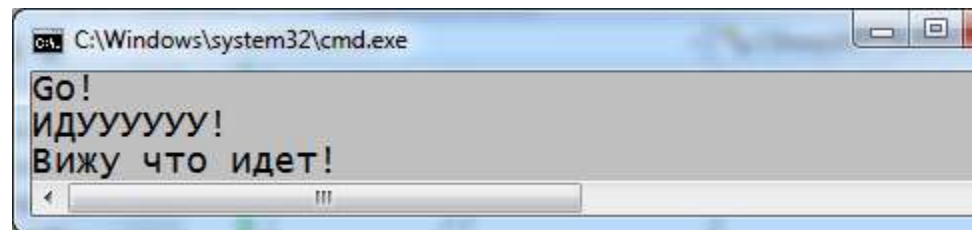
```
{    public void OnGo() {    Console.WriteLine( "ИДУУУУУУ!" );    }
}
```

```
class Eye
```

```
{    public void OnGo() { Console.WriteLine("Вижу что идет!");}
```

```
class Class1
```

```
{ static void Main()
    {    Mind s = new Mind();
        Leg o1 = new Leg();
        Eye o2 = new Eye();
        s.Go += new SomeDelegat(o1.OnGo);
        s.Go += new SomeDelegat(o2.OnGo);
        s.ComandGo();
    }
}
```



Стандартные .NET делегаты

правила:

- имя делегата заканчивается суффиксом EventHandler;
- делегат получает два параметра:
 - 1) первый параметр задает источник события и имеет тип object;
 - 2) второй параметр задает аргументы события и имеет тип EventArgs или производный от него.

- ▶ Имя обработчика события принято составлять из префикса On и имени события

void обработчик(object отправитель, EventArgs e) { //... }

- ▶ Классы содержащие информацию о событиях должны наследовать от типа System.EventArgs, а имя типа должно заканчиваться словом EventArgs

+делегаты

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

```
s.Go += new SomeDelegat(()=>  
    { Console.WriteLine("Я слышу что идете!"); });
```

- ▶ Делегаты и события обеспечивают взаимодействие взаимосвязанных объектов.
- ▶ События включены во многие стандартные классы .NET, используемые для разработки Windows-приложений.