

Лекция 6

Делегаты. Лямбда выражения. События

Делегаты

- ▶ Делегат представляет собой объект, который может ссылаться на метод. Синтаксис:
`delegate` возвращаемый_тип `имя`(параметры);
- ▶ Один и тот же делегат может быть использован для вызова разных методов во время выполнения программы. Метод, вызываемый делегатом, определяется во время выполнения, а не в процессе компиляции.





```
delegate string StrMod(string str);
```

```
.....
```

```
public class DelegateTest{
```

```
    public static string ReplaceSpaces(string s){
```

```
        return s.Replace(' ', '-');
```

```
    }
```

```
    public string RemoveSpaces(string s){
```

```
        return s.Replace(" ", "");
```

```
    }
```

```
    public static int GetLength(string s){
```

```
        return s.Length;
```

```
    }
```

```
}
```

```
...
```

```
var strOp = new StrMod(DelegateTest.ReplaceSpaces);
```

```
var str = strOp("Это простой тест.");
```

Групповое преобразование делегируемых методов

- Групповое преобразование методов, позволяет присвоить имя метода делегату, не прибегая к оператору **new** или явному вызову конструктора делегата.

```
StrMod strOp = DelegateTest.ReplaceSpaces;  
var str = strOp("Это простой тест.");  
var dt = new DelegateTest();  
strOp = dt.RemoveSpaces;  
str = strOp("Это простой тест.");
```

Групповая адресация

- Групповая адресация — это возможность создать список, или цепочку вызовов, для методов, которые вызываются автоматически при обращении к делегату.
- Групповую адресацию возможно применять для методов с типом возвращаемого объекта void (для возврата значения можно использовать параметр типа **ref**)
- Для добавления ссылки на делегат используется оператор +=, для удаления -=

```
StrMod replaceSp = ReplaceSpaces;
```

```
StrMod removeSp = RemoveSpaces;
```

```
var str = "Это простой тест.";
```

```
var strOp = replaceSp;
```

```
strOp += removeSp;
```

```
strOp(ref str);
```

Ковариантность и контравариантность

- **Ковариантность** позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата. А **контравариантность** позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

```
class X { public int Val; }
class Y : X { }
delegate X ChangeIt(Y obj);
class CoContraVariance{
    static X IncrA(X obj){
        var temp = new X(); temp.Val = obj.Val + 1; return temp;
    }
    static Y IncrB(Y obj){...}
}
...
var Yob = new Y();
ChangeIt change = IncrA; //пример контравариантности
var Xob = change(Yob);
change = IncrB; // пример ковариантности
Yob = (Y)change(Yob);
```


Анонимные функции

- **Анонимные функции** — один из способов создания безымянного блока кода, связанного с конкретным экземпляром делегата. Для создания анонимного метода достаточно указать кодовый блок после ключевого слова *delegate*.

```
delegate int CountIt(int end);  
  
...  
CountIt count = delegate (int end) {  
    int sum = 0;  
    for (int i = 0; i <= end; i++){  
        Console.WriteLine(i);  
        sum += i;  
    }  
    return sum;  
};  
result = count(3);  
Console.WriteLine("Сумма 3 равна " + result);
```

Лямбда-выражения

- Лямбда-выражение — это способ создания анонимной функции. Лямбда-выражение может быть присвоено делегату.
- В лямбда-выражениях применяется лямбда-оператор `=>`, который разделяет лямбда-выражение на две части. В левой его части указывается входной параметр (или несколько параметров), а в правой части — тело лямбда-выражения.
- Примеры:
 - `count => count + 2;`
 - `(low, high, val) => val >= low && val <= high;`



Блочные лямбда-выражения

- *Блочное лямбда-выражение* характеризуется расширенными возможностями выполнения различных операций, поскольку в его теле допускается указывать несколько операторов.

```
delegate int IntOp(int end);  
IntOp fact = n => {  
    int r = 1;  
    for (int i = 1; i <= n; i++){  
        r = i * r;  
    }  
    return r;  
};
```



Обобщенные делегаты

```
public delegate void SomeDelegate<T>(T item);

public static void Show(string msg){
    Console.WriteLine(msg);
    Console.ReadLine();
}

SomeDelegate<int> d1 = new SomeDelegate<int>(Show);
d1(5);
```

Встроенные делегаты Action и Func

- Делегат Action является обобщенным, принимает параметры и возвращает значение void

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

- Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,...in T16>. Таким образом можно передать до 16 значений в метод. Как правило, этот делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия.
- Func возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров.

```
TResult Func<out TResult>()
```

```
TResult Func<in T, out TResult>(T arg)
```

Встроенный делегат Predicate

- Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено

```
Predicate<int> isPositive = delegate (int x) { return x > 0; };  
Console.WriteLine(isPositive(20));
```



Использования встроенных делегатов в List<T>

- `Exists(Predicate<T>)` - определяет, содержит ли List<T> элементы, удовлетворяющие условиям указанного предиката.
- `Find(Predicate<T>)` - выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает первое найденное вхождение в пределах всего списка List<T>.
- `FindAll(Predicate<T>)` - извлекает все элементы, удовлетворяющие условиям указанного предиката.
- `ForEach(Action<T>)` - выполняет указанное действие с каждым элементом списка List<T>
- `RemoveAll(Predicate<T>)` - удаляет все элементы, удовлетворяющие условиям указанного предиката.

Замыкания

- Замыкание (англ. *closure*) — это процедура, которая ссылается на свободные переменные в своём лексическом контексте. Замыкание, так же как и экземпляр объекта, есть способ представления функциональности и данных, связанных и упакованных вместе.

```
var funcs = new List<Func<int>>();  
for (int i = 0; i < 3; ++i)  
{  
    funcs.Add(() => i);  
}  
foreach (var f in funcs)  
    Console.WriteLine(f());
```

IT'S NOT A BUG




IT'S A FEATURE!



События

- Событие представляет собой автоматическое уведомление о том, что произошло некоторое действие. События — это особый тип делегата, это члены класса, которые нельзя вызывать вне класса независимо от спецификатора доступа.
- События являются членами класса и объявляются с помощью ключевого слова **event**. Чаще всего для этой цели используется следующая форма:

`event делегат_события имя_события;`
- где `делегат_события` обозначает имя делегата, используемого для поддержки события, а `имя_события` — конкретный объект объявляемого события.



```
public class Pub{  
    public event Action OnChange = delegate { };  
    public void Raise(){  
        OnChange();  
    }  
}  
...  
var p = new Pub();  
p.OnChange += () => Console.WriteLine("Subscriber 1!");  
...  
p.OnChange += () => Console.WriteLine("Subscriber 2!");  
...  
p.Raise();
```

Общее соглашение по обработке событий в .Net

- У обработчиков событий должны быть два параметра. Первый из них — ссылка на объект, формирующий событие, второй — параметр типа **EventArgs**, содержащий любую дополнительную информацию о событии, которая требуется обработчику. .NET-совместимые обработчики событий должны иметь следующую общую форму:

```
void handler(object sender, EventArgs e){  
    ...  
}
```

- отправитель — это параметр, передаваемый вызывающим кодом с помощью ключевого слова `this`.
- параметр **e** типа `EventArgs` содержит дополнительную информацию о событии (например, какая кнопка была нажата) и может быть проигнорирован, если он не нужен.

Встроенные делегаты EventHandler<EventArgs> и EventHandler

- В среде .NET Framework предоставляется встроенный обобщенный делегат под названием EventHandler<EventArgs>. В данном случае тип EventArgs обозначает тип аргумента, передаваемого параметру EventArgs события. Например:

```
public event EventHandler<MyEventArgs> SomeEvent;
```

- Необобщенный делегат типа EventHandler может быть использован для объявления обработчиков событий, которым не требуется дополнительная информация о событиях.



Пример

```
public class Pub{
    public event EventHandler<EventArgs> OnChange;
    protected virtual void RiseEvent(EventArgs e)
    {
        EventHandler<EventArgs> handler = OnChange;
        if (handler != null)
        {
            handler(this, e);
        }
    }
}
...
pub.OnChange += SomeMethodToHandleEvent
...
static void SomeMethodToHandleEvent(object sender, EventArgs e){
    Console.WriteLine("My handler");
}
```