**Программирование C#**

# 12. Делегаты, события и лямбда-выражения

Карбаев Д.С., 2015

- **Делегат** предоставляет возможность инкапсулировать метод, а событие уведомляет о том, что произошло некоторое действие.
- **Лямбда-выражение** представляет собой новое синтаксическое средство, обеспечивающее упрощенный, но в то же время эффективный способ определения того, что по сути является единицей исполняемого кода. Лямбда-выражения обычно служат для работы с делегатами и событиями, поскольку делегат может ссылаться на лямбда-выражение.

# Делегаты

- Делегат представляет собой объект, который может ссылаться на метод. Следовательно, когда создается делегат, то в итоге получается объект, содержащий ссылку на метод. Метод можно вызывать по этой ссылке.
- Т.е., делегат позволяет вызывать метод, на который он ссылается.
- Один и тот же делегат может быть использован для вызова разных методов во время выполнения программы, для чего достаточно изменить метод, на который ссылается делегат.
- Общая форма объявления делегата:

```
delegate возвращаемый_тип имя(список_параметров);
```

- Делегат может служить для вызова любого метода с соответствующей сигнатурой и возвращаемым типом. Вызываемый метод может быть методом экземпляра, связанным с отдельным объектом, или же статическим методом, связанным с конкретным классом.

```
delegate string StrMod(string str); // Объявить тип делегата
class DelegateTest{
    static string ReplaceSpaces(string s) { // Заменить пробелы дефисами
        Console.WriteLine("Замена пробелов дефисами.");
        return s.Replace(' ', '-');
    }
    static string RemoveSpaces(string s) { // Удалить пробелы
        string temp = ""; int i;
        Console.WriteLine("Удаление пробелов.");
        for (i = 0; i < s.Length; i++)
            if (s[i] != ' ') temp += s[i];
        return temp;
    }
    static string Reverse(string s) {// Обратить строку
        string temp = "";  int i, j;
        Console.WriteLine("Обращение строки.");
        for (j = 0, i = s.Length - 1; i >= 0; i--, j++)
            temp += s[i];
        return temp;
    }
    static void Main() {
        StrMod strOp = new StrMod(ReplaceSpaces); // Сконструировать делегат
        string str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = new StrMod(RemoveSpaces); str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = new StrMod(Reverse); str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
    }}
```

**Результат:**

```
Замена пробелов дефисами
Результирующая строка: Это
простой-тест.
Удаление пробелов.
Результирующая строка:
Этопростойтест.
Обращение строки.
Результирующая строка:
.тсет йотсорп отЭ
```

# Групповое преобразование делегируемых методов

```
static void Main() {
    // Сконструировать делегат, используя групповое преобразование методов
    // (без оператора new и явного вызова конструктора делегата)
    StrMod strOp = ReplaceSpaces; // использовать групповое преобразование методов
    string str;
    // Вызвать методы с помощью делегата
    str = strOp("Это простой тест");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();
    strOp = RemoveSpaces; // использовать групповое преобразование методов
    str = strOp("Это простой тест");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();
    strOp = Reverse; // использовать групповое преобразование методов
    str = strOp("Это простой тест");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();
}
```

```
// Делегаты могут ссылаться и на методы экземпляра
delegate string StrMod(string str);
class StringOps {
public string ReplaceSpaces(string s) {// Заменить пробелы дефисами
 //...
    }
public string RemoveSpaces(string s) {// Удалить пробелы
 //...
    }
public string Reverse(string s) {// Обратить строку
 //...
}}
class DelegateTest {
    static void Main(){
        StringOps so = new StringOps(); // создать экземпляр
        // объекта класса StringOps
        StrMod strOp = so.ReplaceSpaces;// Инициализировать делегат
        string str;
        // Вызвать методы с помощью делегатов
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = so.RemoveSpaces;
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
        strOp = so.Reverse;
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
    }}
```

# Групповая адресация

- Одним из самых примечательных свойств делегата является поддержка групповой адресации.
- **Групповая адресация** — это возможность создать список, или ***цепочку вызовов***, для методов, которые вызываются автоматически при обращении к делегату.
- Создать такую цепочку нетрудно. Для этого достаточно получить  экземпляр делегата, а затем добавить методы в цепочку с помощью оператора **+** или **+=**.
- Для удаления метода из цепочки служит оператор **-** или **-=**. Если делегат возвращает значение, то им становится значение, возвращаемое последним методом в списке вызовов. Поэтому делегат, в котором используется групповая адресация, обычно имеет возвращаемый тип void.

```
// Продемонстрировать групповую адресацию
delegate void StrMod(ref string str);
class MultiCastDemo {
    static void ReplaceSpaces(ref string s) {
      // Заменить пробелы дефисами
    }
    static void RemoveSpaces(ref string s) {
      // Удалить пробелы
    }
    static void Reverse(ref string s) {
      // Обратить строку
    }
    static void Main() {
        // Сконструировать делегаты
        StrMod strOp;
        StrMod replaceSp = ReplaceSpaces;
        StrMod removeSp = RemoveSpaces; StrMod reverseStr = Reverse;
        string str = "Это простой тест.";
        // Организовать групповую адресацию
        strOp = replaceSp; strOp += reverseStr;
        // Обратиться к делегату с групповой адресацией
        strOp(ref str);
        Console.WriteLine("Результирующая строка: " + str); Console.WriteLine();
        // Удалить метод замены пробелов и добавить метод удаления пробелов
        strOp -= replaceSp; strOp += removeSp;
        str = "Это простой тест."; // восстановить исходную строку
        // Обратиться к делегату с групповой адресацией
        strOp(ref str);
        Console.WriteLine("Результирующая строка: " + str); Console.WriteLine();
  }}
```

```
Замена пробелов дефисами.
Обращение строки.
Результирующая строка: .тсет-йотсорп-отЭ
Обращение строки.
Удаление пробелов.
Результирующая строка: .тсетйотсорпотЭ
```

# Ковариантность и контравариантность

- Делегаты становятся еще более гибкими средствами программирования благодаря двум свойствам: ковариантности и контравариантности.
- Как правило, метод,  передаваемый делегату, должен иметь такой же возвращаемый тип и сигнатуру, как и делегат.
- **Ковариантность** позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата.
- **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 { }
// Делегат возвращает объект класса X и принимает объект класса Y в качестве аргумента
delegate X ChangeIt(Y obj);
class CoContraVariance {
    //Метод возвращает объект класса X и имеет объект класса X в качестве параметра
    static X IncrA(X obj) {
        X temp = new X(); temp.Val = obj.Val + 1;
        return temp;
    }
    //Метод возвращает объект класса Y и имеет объект класса Y в качестве параметра
    static Y IncrB(Y obj) {
        Y temp = new Y(); temp.Val = obj.Val + 1;
        return temp;
    }
    static void Main() {
        Y Yo = new Y();
        // В данном случае параметром метода IncrA является объект класса X,
        // а параметром делегата ChangeIt – объект класса Y. Но благодаря
        // контравариантности следующая строка кода вполне допустима
        ChangeIt change = IncrA;
        X Xo = change(Yo);
        Console.WriteLine("Xo: " + Xo.Val);
        // В этом случае возвращаемым типом метода IncrB служит объект класса Y,
        // а возвращаемым типом делегата ChangeIt – объект класса X. Но благодаря
        // ковариантности следующая строка кода оказывается вполне допустимой
        change = IncrB;
        Yo = (Y)change(Yo);
        Console.WriteLine("Yo: " + Yo.Val);
    }}
```

**Результат:**

Xo: 1

Yo: 1

# Делегаты

- Все делегаты и классы оказываются производными неявным образом от класса System. Delegate.
- Делегаты применяются по двум причинам:
  - Во-первых, делегаты поддерживают события.
  - Во-вторых, делегаты позволяют вызывать методы во время выполнения программы, не зная о них ничего определенного в ходе компиляции.
- Рассмотрим в качестве примера графическую программу, аналогичную стандартной сервисной программе Windows Paint.
- С помощью делегата можно предоставить пользователю возможность подключать специальные цветные фильтры или анализаторы изображений. Кроме того, пользователь может составлять из этих фильтров или анализаторов целые последовательности.

# Анонимные функции и методы

- Метод, на который ссылается делегат, нередко используется только для этой цели. Единственным основанием для существования метода служит то обстоятельство, что он может быть вызван посредством делегата, но сам он не вызывается вообще. В подобных случаях можно воспользоваться анонимной функцией, чтобы не создавать отдельный метод.
- **Анонимная функция** представляет собой безымянный кодовый блок, передаваемый конструктору делегата. Благодаря ей отпадает необходимость объявлять отдельный метод, единственное назначение которого состоит в том, что он передается делегату.
- В C# предусмотрены две разновидности анонимных функций: анонимные методы и лямбда-выражения.
- В целом лямбда-выражение совершенствует принцип действия анонимного метода. Анонимные методы могут быть использованы в целом ряде случаев, где применение лямбда-выражений оказывается невозможным.
- Анонимный метод — один из способов создания безымянного блока кода, связанного с конкретным экземпляром делегата.

# Анонимные методы

```
delegate void CountIt();
class AnonMetDemo {
    static void Main()
    {
     // Далее следует код для подсчета чисел, передаваемый делегату
     // в качестве анонимного метода
        CountIt count = delegate {
            // Этот кодовый блок передается делегату
            for (int i = 0; i <= 5; i++)
                Console.WriteLine(i);
        }; // обратите внимание на точку с запятой
        count();
    }
}
```

```
0
1
2
3
4
5
```

# Передача аргументов анонимному методу

```
//Обратите внимание на то, что теперь у делегата CountIt имеется параметр
delegate void CountIt(int end);
class AnonMetDemo2 {
    static void Main()
    {
      // Здесь конечное значение для подсчета передается анонимному методу
        CountIt count = delegate(int end) {
            for (int i = 0; i <= end; i++)
                Console.WriteLine(i);
        };
        count(3);
        Console.WriteLine();
        count(5);
    }
}
```

```
0
1
2
3

0
1
2
3
4
5
```

# Возврат значения из анонимного метода

```
delegate int CountIt(int end); // Этот делегат возвращает значение
class AnonMethDemo3{
    static void Main(){
        int result;
        // Здесь конечное значение для подсчета передается анонимному методу,
        // а возвращается сумма подсчитанных чисел
        CountIt count = delegate(int end){
            int sum = 0;
            for (int i = 0; i <= end; i++){
                Console.WriteLine(i);
                sum += i;
            }
            return sum; //вернуть значение из анонимного метода
        };
        result = count(3);
        Console.WriteLine("Сумма 3 равна " + result);
        Console.WriteLine();
        result = count(5);
        Console.WriteLine("Сумма 5 равна " + result);
    }
}
```

```
0
1
2
3
Сумма 3 равна 6
0
1
2
3
4
5
Сумма 5 равна 15
```

# Захват переменной

```
// Этот делегат возвращает значение типа int и принимает аргумент типа int
delegate int CountIt(int end);
class VarCapture {
    static CountIt Counter() {
       int sum = 0;
       // Здесь подсчитанная сумма сохраняется в переменной sum
       CountIt ctObj = delegate(int end) {
            for (int i = 0; i <= end; i++){
                Console.WriteLine(i);
                sum += i;
            }
            return sum;
        };
        return ctObj;
    }
    static void Main() {
       // Получить результат подсчета
       CountIt count = Counter();
        int result;
        result = count(3);
        Console.WriteLine("Сумма 3 равна " + result);
        Console.WriteLine();
        result = count(5);
        Console.WriteLine("Сумма 5 равна " + result);
    }
}
```

```
0
1
2
3
Сумма 3 равна 6
0
1
2
3
4
5
Сумма 5 равна 21
```

# Лямбда-выражения

- **Лямбда-выражение** — это другой собой создания анонимной функции. Следовательно, лямбда-выражение может быть присвоено делегату.
- Во всех лямбда-выражениях применяется новый 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. Оператор **=>** иногда описывается такими словами, как "переходит" или "становится".
- В C# поддерживаются две разновидности лямбда-выражений в зависимости от тела самого лямбда-выражения. Так, если тело лямбда-выражения состоит из одного выражения, то образуется **одиночное лямбда-выражение**. В этом случае тело выражения не заключается в фигурные скобки.
- Если же тело лямбда-выражения состоит из блока операторов, заключенных в фигурные скобки, то образуется **блочное лямбда-выражение**. При этом блочное лямбда-выражение может содержать целый ряд операторов, в том числе циклы, вызовы методов и условные операторы if.

# Одиночные лямбда-выражения

- В **одиночном лямбда-выражении** часть, находящаяся справа от оператора =>, воздействует на параметр (или ряд параметров), указываемый слева. Возвращаемым результатом вычисления такого выражения является результат выполнения лямбда-оператора.
- Общая форма одиночного лямбда-выражения, принимающего единственный параметр.

```
параметр => выражение
```

- Если же требуется указать несколько параметров:

```
(список_параметров) => выражение
```

- Пример одиночного лямбда-выражения:

```
count => count + 2;
```

- В этом выражении count служит параметром, на который воздействует выражение count + 2. В итоге значение параметра count увеличивается на 2.
- Еще один пример одиночного лямбда-выражения.

```
n => n % 2 == 0;
```

- В данном случае выражение возвращает логическое значение true, если числовое значение параметра n оказывается четным, а иначе — логическое значение false.

```
delegate int Incr(int v);
delegate bool IsEven(int v);
class SimpleLambdaDemo {
    static void Main() {
        // Создать делегат Incr, ссылающийся
        // на лямбда-выражение,
        // увеличивающее свой параметр на 2
        Incr incr = count => count + 2;
        // А теперь использовать лямбда-выражение incr
        Console.WriteLine("Использование лямбда-выражения incr: ");
        int x = -10;
        while (x <= 0) {
            Console.Write(x + " ");
            x = incr(x); // увеличить значение x на 2
        }
        Console.WriteLine("\n");
        // Создать экземпляр делегата IsEven, ссылающийся на лямбда-выражение,
        // возвращающее логическое значение true, если его параметр имеет четное
        // значение, а иначе – логическое значение false
        IsEven isEven = n => n % 2 == 0;
        //А теперь использовать лямбда-выражение isEven
        Console.WriteLine("Использование лямбда-выражения isEven: ");
        for (int i = 1; i <= 10; i++)
            if (isEven(i)) Console.WriteLine(i + " четное.");
    }}
```

```
Использование лямбда-выражения incr:
-10 -8 -6 -4 -2 0
Использование лямбда-выражения isEven:
2 четное.
4 четное.
6 четное.
8 четное.
10 четное.
```

# Одиночные лямбда-выражения

- В целом у лямбда-выражений может быть любое количество параметров, в том числе и нулевое. Если в лямбда-выражении используется несколько параметров, их необходимо заключить в скобки.
- Пример использования лямбда-выражения с целью определить, находится ли значение в заданных пределах:

```
(low, high, val) => val >= low && val <= high;
```

- Тип делегата, совместимого с этим лямбда-выражением:

```
delegate bool InRange(int lower, int upper, int v);
```

- Следовательно, можно объеявить экземпляр делегата InRange:

```
InRange rangeOK = (low, high, val) => val >= low && val <= high;
```

- После этого одиночное лямбда-выражение может быть выполнено так:

```
if(rangeOK(1, 5, 3)) Console.WriteLine(
"Число 3 находится в пределах от 1 до 5.");
```

- Внешние переменные могут использоваться и захватываться в лямбда-выражениях таким же образом, как и в анонимных методах.

# Блочные лямбда-выражения

- Второй разновидностью является **блочное лямбда-выражение**. Для такого лямбда-выражения характерны расширенные возможности выполнения различных операций, поскольку в его теле  допускается указывать несколько операторов.
- Например, в блочном лямбда-выражении можно использовать циклы и условные операторы if, объявлять переменные и т.д.
- Для создания блочного лямбда-выражения достаточно заключить тело выражения в фигурные скобки. Помимо возможности использовать несколько операторов, в остальном блочное лямбда-выражение, практически ничем не отличается от только что рассмотренного одиночного лямбда-выражения.

# Блочные лямбда-выражения

```
// Делегат IntOp принимает один аргумент типа int
// и возвращает результат типа int
delegate int IntOp(int end);
class StatementLambdaDemo {
    static void Main(){
        // Блочное лямбда-выражение возвращает факториал
        // передаваемого ему значения
        IntOp fact = n => {
            int r = 1;
            for (int i = 1; i <= n; i++)
                r = i * r;
            return r;
        };
        Console.WriteLine("Факториал 3 равен " + fact(3));
        Console.WriteLirie("Факториал 5 равен " + fact(5));
    }
}
```

Факториал 3 равен 6
Факториал 5 равен 120

```
// Первый пример применения делегатов, переделанный с
// целью использовать блочные лямбда-выражения
delegate string StrMod(string s);
class UseStatementLambdas {
    static void Main() {
        // Создать делегаты, ссылающиеся на лямбда-выражения,
        // выполняющие различные операции с символьными строками
        StrMod ReplaceSpaces = s => {
            // Заменить пробелы дефисами
            return s.Replace(' ', '-');
        };
        StrMod RemoveSpaces = s => {
            // Удалить пробелы
            return temp;
        };
        StrMod Reverse = s => {
            // Обратить строку
            return temp;
        };
        string str;
        // Обратиться к лямбда-выражениям с помощью делегатов
        StrMod strOp = ReplaceSpaces;
        str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        strOp = RemoveSpaces; str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
        strOp = Reverse; str = strOp("Это простой тест.");
        Console.WriteLine("Результирующая строка: " + str);
    }}
```

```
Замена пробелов дефисами.
Результирующая строка:
Это-простой-тест.
Удаление пробелов.
Результирующая строка:
Этопростойтест.
Обращение строки.
Результирующая строка:
.тсет йотсорп отЭ
```

# События

- **Событие**, по существу, представляет собой автоматическое уведомление о том, что произошло некоторое действие. События действуют по следующему принципу: объект, проявляющий интерес к событию, регистрирует обработчик этого события.
- Когда же событие происходит, вызываются все зарегистрированные обработчики этого события. Обработчики событий обычно представлены делегатами.
- 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.

# События

```
// Объявить тип делегата для события
delegate void MyEventHandler();
class MyEvent {
    public event MyEventHandler SomeEvent;
    // Этот метод вызывается для запуска события
    public void OnSomeEvent() {
        if (SomeEvent != null)
            SomeEvent();
    }
}
class EventDemo {
    // Обработчик события
    static void Handler() {
        Console.WriteLine("Произошло событие");
    }
    static void Main() {
        MyEvent evt = new MyEvent();
        // Добавить метод Handler() в список событий
        evt.SomeEvent += Handler;
        // Запустить событие
        evt.OnSomeEvent();
    }
}
```

```
// Групповая адресация событий
delegate void MyEventHandler();
class MyEvent {
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent(){//метод для запуска события
        if (SomeEvent != null) SomeEvent();
}   }
class X {
    public void Xhandler() {
      Console.WriteLine("Событие получено объектом класса X");
}   }
class Y {
    public void Yhandler() {
      Console.WriteLine("Событие получено объектом класса Y");
}   }
class EventDemo2 {
    static void Handler() {
      Console.WriteLine("Событие получено объектом класса EventDemo");
    }
    static void Main(){
        MyEvent evt = new MyEvent();
        X xObj = new X(); Y yObj = new Y();
        // Добавить обработчики в список событий
        evt.SomeEvent += Handler;
        evt.SomeEvent += xObj.Xhandler;
        evt.SomeEvent += yObj.Yhandler;
        // Запустить событие
        evt.OnSomeEvent();
        // Удалить обработчик
        evt.SomeEvent -= xObj.Xhandler;
        evt.OnSomeEvent();
    }}
```

Событие получено объектом класса EventDemo
Событие получено объектом класса X
Событие получено объектом класса Y
Событие получено объектом класса EventDemo
Событие получено объектом класса Y

```
// Уведомления о событиях получают отдельные объекты, когда метод экземпляра
// используется в качестве обработчика событий
delegate void MyEventHandler() ;
class MyEvent {
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent() {// Этот метод вызывается для запуска события
        if (SomeEvent != null)
            SomeEvent();
}   }
class X {
    int id;
    public X(int x) { id = x; }
    // Этот метод экземпляра предназначен в качестве обработчика событий
    public void Xhandler(){
        Console.WriteLine("Событие получено объектом " + id);
    }
}
class EventDemo3 {
    static void Main() {
        MyEvent evt = new MyEvent();
        X o1 = new X(1);
        X o2 = new X(2);
        X o3 = new X(3);
        evt.SomeEvent += o1.Xhandler;
        evt.SomeEvent += o2.Xhandler;
        evt.SomeEvent += o3.Xhandler;
        // Запустить событие
        evt.OnSomeEvent();
    }}
```

Событие получено объектом 1
Событие получено объектом 2
Событие получено объектом 3

```
// Уведомления о событии получает класс, когда статический
// метод используется в качестве обработчика событий
delegate void MyEventHandler();
class MyEvent {
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent(){// Этот метод вызывается для запуска события
        if (SomeEvent != null)
            SomeEvent();
    }
}
class X {
    // Этот статический метод предназначен в качестве обработчика событий
    public static void Xhandler(){
        Console.WriteLine("Событие получено классом.");
    }
}
class EventDemo4 {
    static void Main() {
        MyEvent evt = new MyEvent();
        evt.SomeEvent += X.Xhandler;
        // Запустить событие
        evt.OnSomeEvent();
}   }
```

Событие получено классом.

# Применение аксессоров событий

- Организовать управление списком вызовов обработчиков событий можно и вручную, чтобы, например, реализовать специальный механизм сохранения событий.
- Для управления списком обработчиков событий служит расширенная форма оператора event, позволяющая использовать **аксессоры событий**. Эти аксессоры предоставляют средства для управления реализацией подобного списка в приведенной ниже форме.

```
event делегат_события имя_события {
  add {
   // Код добавления события в цепочку событий.
  }
  remove {
   // Код удаления события из цепочки событий.
  }
}
```

- Аксессор **add** вызывается, когда обработчик событий добавляется в цепочку событий с помощью оператора **+=**. Аксессор **remove** вызывается, когда обработчик событий удаляется из цепочки событий с помощью оператора **-=**.
- Обработчики событий можно хранить в массиве, стеке или очереди.

```
// Создать специальные средства для управления списками вызова обработчиков событий
delegate void MyEventHandler();
class MyEvent {
    MyEventHandler[] evnt = new MyEventHandler[3];
    public event MyEventHandler SomeEvent {
        add { // Добавить событие в список
            int i;
            for (i = 0; i < 3; i++)
                if (evnt[i] == null){ evnt[i] = value; break; }
            if (i == 3) Console.WriteLine("Список событий заполнен.");
        }
        remove { // Удалить событие из списка
            int i;
            for (i = 0; i < 3; i++)
                if (evnt[i] == value){ evnt[i] = null; break; }
            if (i == 3) Console.WriteLine("Обработчик событий не найден.");
    }   }
    // Этот метод вызывается для запуска событий
    public void OnSomeEvent() {
        for (int i = 0; i < 3; i++)
            if (evnt[i] != null) evnt[i]();
    }
}
// Создать ряд классов, использующих делегат MyEventHandler
class W {
    public void Whandler() {
        Console.WriteLine("Событие получено объектом W");
    }
}
```

```
class X {
    public void Xhandler(){ Console.WriteLine("Событие получено объектом X"); }
}
class Y {
    public void Yhandler(){ Console.WriteLine("Событие получено объектом Y"); }
}
class Z {
    public void Zhandler(){ Console.WriteLine("Событие получено объектом Z"); }
}
class EventDemo5 {
    static void Main() {
        MyEvent evt = new MyEvent();
        W wObj = new W(); X xObj = new X();
        Y yObj = new Y(); Z zObj = new Z();
        // Добавить обработчики в цепочку событий.
        Console.WriteLine("Добавление событий. ");
        evt.SomeEvent += wObj.Whandler; evt.SomeEvent += xObj.Xhandler;
        evt.SomeEvent += yObj.Yhandler;
        evt.SomeEvent += zOb.Zhandler;// Сохранить нельзя - список заполнен
        evt.OnSomeEvent(); // Запустить события
        // Удалить обработчик
        Console.WriteLine("Удаление обработчика xObj.Xhandler.");
        evt.SomeEvent -= xObj.Xhandler; evt.OnSomeEvent();
        // Попробовать удалить обработчик еще раз
        Console.WriteLine("Попытка удалить обработчик xObj.Xhandler еще раз.");
        evt.SomeEvent -= xObj.Xhandler; evt.OnSomeEvent();
        //А теперь добавить обработчик Zhandler
        Console.WriteLine("Добавление обработчика zObj.Zhandler.");
        evt.SomeEvent += zObj.Zhandler; evt.OnSomeEvent();
}  }
```

**Результат**

Добавление событий.
Список событий заполнен.
Событие получено объектом W
Событие получено объектом X
Событие получено объектом Y
Удаление обработчика xOb.Xhandler.
Событие получено объектом W
Событие получено объектом Y
Попытка удалить обработчик
xOb.Xhandler еще раз.
Обработчик событий не найден.
Событие получено объектом W
Событие получено объектом Y
Добавление обработчика zOb.Zhandler.
Событие получено объектом W
Событие получено объектом X
Событие получено объектом Y

# Возможности событий

- События могут быть определены и в интерфейсах. При этом события должны предоставляться классами, реализующими интерфейсы.
- События могут быть также определены как абстрактные (abstract). В этом случае конкретное событие должно быть реализовано в производном классе. Но аксессорные формы событий не могут быть абстрактными.
- Событие может быть определено как герметичное (sealed).
- Событие может быть виртуальным, т.е. его можно переопределить в производном классе.

```
// Использовать лямбда-выражение в качестве обработчика событий
delegate void MyEventHandler(int n);
class MyEvent {
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent(int n) {
        if (SomeEvent != null)
            SomeEvent(n);
    }
}
class LambdaEventDemo {
  static void Main() {
    MyEvent evt = new MyEvent();
    // Использовать лямбда-выражение в качестве обработчика событий
    evt.SomeEvent += (n) =>
      Console.WriteLine("Событие получено. Значение равно " + n);
    // Запустить событие
    evt.OnSomeEvent(1);
    evt.OnSomeEvent(2);
  }
}
```

Событие получено. Значение равно 1
Событие получено. Значение равно 2

# Рекомендации по обработке событий в среде .NET Framework

- В C# разрешается формировать какие угодно разновидности событий.
- Рекомендации сводятся к следующему требованию: 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 обработчик(object отправитель, EventArgs e) {
// ...
}
```

- Как правило, отправитель — это параметр, передаваемый вызывающим кодом с помощью ключевого слова **this**.
  А параметр **e** типа **EventArgs** содержит дополнительную информацию о событии и может быть проигнорирован, если он не нужен.

```
// Пример формирования .NET-совместимого события
class MyEventArgs : EventArgs { public int EventNum; }
delegate void MyEventHandler(object source, MyEventArgs arg);
class MyEvent {
    static int count = 0;
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent(){
        MyEventArgs arg = new MyEventArgs();
        if (SomeEvent != null){ afg.EventNum = count++;
            SomeEvent(this, arg);
}   }   }
class X {
    public void Handler(object source, MyEventArgs arg){
        Console.WriteLine("Событие " + arg.EventNum +
        " получено объектом класса X.");
        Console.WriteLine("Источник: " + source);
}   }
class Y{
    public void Handler(object source, MyEventArgs arg){
        Console.WriteLine("Событие " + arg.EventNum +
        " получено объектом класса Y.");
        Console.WriteLine("Источник: " + source);
}}
class EventDemo6{
    static void Main(){
        X ob1 = new X(); Y ob2 = new Y();
        MyEvent evt = new MyEvent();
        evt.SomeEvent += ob1.Handler;
        evt.SomeEvent += ob2.Handler;
        evt.OnSomeEvent(); evt.OnSomeEvent();
    }}
```

```
Событие 0 получено
объектом класса X
Источник: MyEvent
Событие 0 получено
объектом класса Y
Источник: MyEvent
Событие 1 получено
объектом класса X
Источник: MyEvent
Событие 1 получено
объектом класса Y
Источник: MyEvent
```

# Применение делегатов EventHandler<TEventArgs> и EventHandler

- В среде .NET Framework предоставляется встроенный обобщенный делегат под названием **EventHandler<TEventArgs>**. В данном случае тип **TEventArgs** обозначает тип аргумента, передаваемого параметру **EventArgs** события.
- Например, в приведенной выше программе событие **SomeEvent** может быть объявлено в классе **MyEvent** следующим образом.

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

- В общем, рекомендуется пользоваться именно таким способом, а не определять собственный делегат.
- Для обработки многих событий параметр типа **EventArgs** оказывается ненужным. Поэтому с целью упростить создание кода в подобных ситуациях в среду .NET Framework внедрен необобщенный делегат типа **EventHandler**. Он может быть использован для объявления обработчиков событий, которым не требуется дополнительная информация о событиях.

```
// Использовать встроенный делегат EventHandler
class MyEvent {
        public event EventHandler SomeEvent;
        public void OnSomeEvent() {
            if (SomeEvent != null)
                SomeEvent(this, EventArgs.Empty);
        }
    }
class EventDemo7 {
        static void handler(object source, EventArgs arg) {
            Console.WriteLine("Произошло событие");
            Console.WriteLine("Источник: " + source);
        }
        static void Main() {
            MyEvent evt = new MyEvent();
            // Добавить обработчик Handler() в цепочку событий
            evt.SomeEvent += Handler;
            // Запустить событие
            evt.OnSomeEvent();
        }
}
```

Произошло событие
Источник: MyEvent

```
// Пример обработки событий, связанных с нажатием клавиш на клавиатуре.
// Создать класс, производный от класса EventArgs и хранящий символ нажатой клавиши
    class KeyEventArgs : EventArgs {
        public char ch;
    }
// Объявить класс события, связанного с нажатием
// клавиш на клавиатуре
class KeyEvent {
    public event EventHandler<KeyEventArgs> KeyPress;
    // Этот метод вызывается при нажатии клавиши
    public void OnKeyPress(char key){
        KeyEventArgs k = new KeyEventArgs();
        if (KeyPress != null){
            k.ch = key; KeyPress(this, k);
}   }   }
// Продемонстрировать обработку события типа KeyEvent.
class KeyEventDemo {
 static void Main(){
  KeyEvent kevt = new KeyEvent();
  ConsoleKeyInfo key; int count = 0;
  // Использовать лямбда-выражение для отображения факта нажатия клавиши
  kevt.KeyPress += (sender, e) =>
    Console.WriteLine(" Получено сообщение о нажатии клавиши: " + e.ch);
  // Использовать лямбда-выражение для подсчета нажатых клавиш
  kevt.KeyPress += (sender, e) => count++; // count – это внешняя переменная
  Console.WriteLine("Введите несколько символов. "+"По завершении введите точку.");
  do{ key = Console.ReadKey();
      kevt.OnKeyPress(key.KeyChar);
  }while (key.KeyChar != '.');
  Console.WriteLine("Было нажато " + count + " клавиш.");
}}
```

```
Введите несколько символов.
По завершении введите точку.
t Получено сообщение о
нажатии клавиши: t
e Получено сообщение о
нажатии клавиши: e
s Получено сообщение о
нажатии клавиши: s
t Получено сообщение о
нажатии клавиши: t
. Получено сообщение о
нажатии клавиши: .
Было нажато 5 клавиш.
```