СОЦИАЛЬНО-ГУМАНИТАРНЫЙ КОЛЛЕДЖ
УО «МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ А.А. КУЛЕШОВА»

Дисциплина
«Конструирование программ и языки программирования»

Разработка делегатов, событий, лямюда-выражений и анонимных методов и использование их в программах

(4 часа)

Методические рекомендации к лабораторной работе №12-13

Могилев 2019

Понятия «делегаты», «события», «анонимные методы», «лямбда-выражения». Методические указания по лабораторной работе №12-13 «Конструирование программ и языки программирования». Для учащихся очной формы обучения специальности 1–40 01 01 «Программное обеспечение информационных технологий.

Оглавление

1 Цель работы

Цель:

получить практические навыки по созданию приложений, использующие события, делегаты, лямбда-вражения и анониманые методы.

2 Ход работы

1. Изучение теоретического материала.
2. Выполнение практических индивидуальных заданий по вариантам (вариант уточняйте у преподавателя).
3. Оформление отчета.
   3.1. Отчет оформляется индивидуально каждым студентом. Отчет должен содержать задание, алгоритм и листинг программы.
   3.2. Отчет по лабораторной работе выполняется на листах формата А4. В состав отчета входят:
      1) титульный лист;
      2) цель работы;
      3) текст индивидуального задания;
      4) выполнение индивидуального задания.
4. Контрольные вопросы.

3 Краткие теоретические сведения

Делегат предоставляет возможность инкапсулировать метод, а событие — это своего рода уведомление о том, что имело место некоторое действие. Делегаты и события связанны между собой, поскольку событие создается на основе делегата. Эти средства расширяют диапазон задач программирования, к которым можно применить язык C#.

3.1 Делегаты

Делегат (delegate) — это объект, который может ссылаться на метод, т.е. создавая делегат, получаем объект, который содержит ссылку на метод. Кроме того, этот метод можно вызвать посредством соответствующей ссылки.

Делегаты используются по двум основным причинам. Во-первых, делегаты обеспечивают поддержку функционирования событий. Во-вторых, во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, которые точно не известны в период компиляции, просто заменив метод, на который ссылается этот делегат.

Все делегаты представляют собой классы, которые неявным образом выводятся из класса System.Delegate.

Делегат объявляется с помощью ключевого слова **delegate**. Общая форма объявления делегата имеет следующий вид:

```
delegate <тип_возврата> <имя>{<список_параметров>);
```

где элемент <**тип_возврата**> представляет собой тип значений, возвращаемых методами, которые этот делегат будет вызывать. Имя делегата указывается элементом <**имя**>. Параметры, принимаемые методами, которые вызываются посредством делегата, задаются с помощью элемента <**список_параметров**>. Делегат может вызывать только такие методы, у которых тип возвращаемого значения и список параметров (т.е. его сигнатура) совпадают с соответствующими элементами объявления делегата.

Делегат может вызывать либо метод экземпляра класса, связанный с объектом, или статический метод, связанный с классом.

3.1.1 Вызов статических методов с помощью делегата

Пример 1. Рассмотрим простой пример, демонстрирующий использование делегата для вызова статических методов. Результат выполнения примера представлен на рис. 1.

```
using System;
namespace ConsoleApplication1
{
    delegate string strMod(string stx); //объявляем делегата с именем strMod,который
                                        // принимает один параметр типа string и
                                        // возвращает string-значение.
    class DelegateTest// В классе DelegateTest объявляются три статических метода,
                    //сигнатура которых совпадает с сигнатурой, заданной делегатом.
    {
        // Метод заменяет пробелы дефисами
        public static string replaceSpaces(string a)
        {
            Console.WriteLine(" Замена пробелов дефисами.");
            return a.Replace(' ', '-');
        }
```

```
        // Метод удаляет пробелы
        public static string removeSpaces(string a)
        {
            string temp = "";
            Console.WriteLine(" Удаление пробелов.");
            for (int i = 0; i < a.Length; i++)
                if (a[i] != ' ')
                    temp += a[i];
            return temp;
        }
        // Метод реверсирует строку
        public static string reverse(string a)
        {
            string temp = "";
            Console.WriteLine(" Реверсирование строки.");
            for (int j = 0, i = a.Length - 1; i >= 0; i--, j++)
                temp += a[i];
            return temp;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = "Пример №1";
            Console.BackgroundColor = ConsoleColor.White; Console.Clear();
            Console.ForegroundColor = ConsoleColor.Black;
            string str;
            // Создание делегата и вызов метода посредством делегата
            // В качестве параметра делегату передается имя метода replaceSpaces()
            strMod strOp = new strMod(DelegateTest.replaceSpaces);
            // метод replaceSpaces() вызывается посредством экземпляра делегата
            // с именем strOp, т.к. экземпляр strOp ссылается на метод
            // replaceSpaces(), то вызывается метод replaceSpaces().
            str = strOp(" Это простой тест.");
            Console.WriteLine(" Результирующая строка: " + str);
            Console.WriteLine();
            // Создание делегата и вызов метода посредством делегата
            strOp = new strMod(DelegateTest.removeSpaces);
            str = strOp(" Это простой тест.");
            Console.WriteLine(" Результирующая строка: " + str);
            Console.WriteLine();
            // Создание делегата и вызов метода посредством делегата
            strOp = new strMod(DelegateTest.reverse);
            str = strOp(" Это простой тест.");
            Console.WriteLine(" Результирующая строка: " + str);
            Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
            Console.Read();
        }
    }
}
```

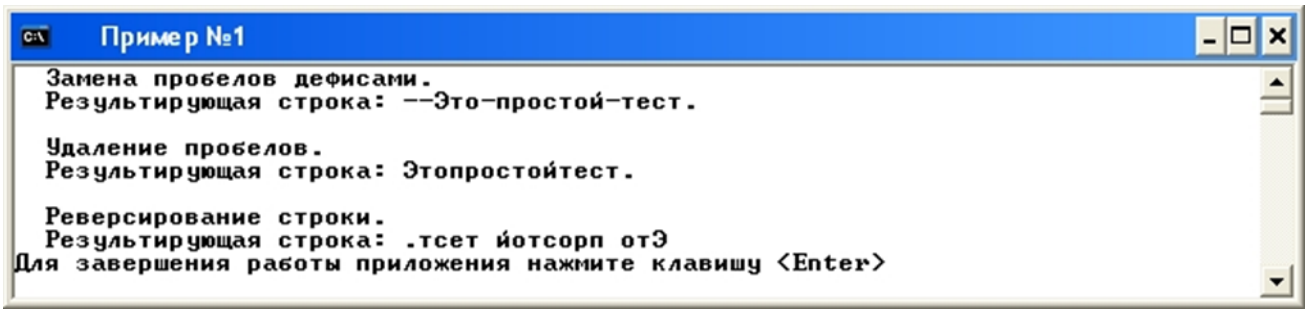


Рис. 1. Результат выполнения примера №1.

Таким образом, решение о вызываемом методе принимается во время выполнения программы, а не в период компиляции.

3.1.2 Вызов методов экземпляра класса с помощью делегата

Несмотря на то, что в предыдущем примере используются статические методы, делегат может также ссылаться на методы экземпляров класса, при этом он должен использовать объектную ссылку.

Пример 2. Вот как выглядит предыдущая программа, переписанная с целью инкапсуляции операций над строками внутри класса StringOps.

```
using System;
namespace ConsoleApplication1
{
    delegate string strMod(string stx); // Объявляем делегата
    class StringOps
    {
        // Метод заменяет пробелы дефисами
        public string replaceSpaces(string a)
        {
            Console.WriteLine(" Замена пробелов дефисами.");
            return a.Replace(' ', '-');
        }
        // Метод удаляет пробелы
        public string removeSpaces(string a)
        {
            string temp = "";
            Console.WriteLine(" Удаление пробелов.");
            for (int i = 0; i < a.Length; i++)
                if (a[i] != ' ') temp += a[i];
            return temp;
        }
        // Метод реверсирует строку
        public string reverse(string a)
        {
            string temp = "";
            Console.WriteLine(" Реверсирование строки.");
            for (int j = 0, i = a.Length - 1; i >= 0; i--, j++)
                temp += a[i];
            return temp;
        }
        class Program
        {
```

```
            static void Main(string[] args)
            {
                Console.Title = "Пример №2";
                Console.BackgroundColor = ConsoleColor.White; Console.Clear();
                Console.ForegroundColor = ConsoleColor.Black;
                string str;
                StringOps so = new StringOps(); //Создаем экземпляр класса StringOps
                                                // Создание делегата и вызов метода
                                                //посредством делегата
                strMod strOp = new strMod(so.replaceSpaces);
                str = strOp(" Это простой тест.");
                Console.WriteLine(" Результирующая строка: " + str);
                Console.WriteLine();
                // Создание делегата и вызов метода посредством делегата
                strOp = new strMod(so.removeSpaces);
                str = strOp(" Это простой тест.");
                Console.WriteLine(" Результирующая строка: " + str);
                Console.WriteLine();
                // Создание делегата и вызов метода посредством делегата
                strOp = new strMod(so.reverse);
                str = strOp(" Это простой тест.");
                Console.WriteLine(" Результирующая строка: " + str);
                Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
                Console.Read();
            }
        }
    }
}
```

Результаты выполнения этой программы совпадают с результатами предыдущей версии, но в этом случае делегат ссылается на методы экземпляра класса StringOps.

3.1.3 Многоадресная передача

Одна из самых интересных возможностей делегата — поддержка многоадресной передачи (multicasting). Многоадресная передача — это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата.

Для создания цепочки вызовов достаточно создать экземпляр делегата, а затем для добавления методов в эту цепочку использовать оператор "+=". Для удаления метода из цепочки используется оператор " -= ". (можно для добавления и удаления методов использовать в отдельности операторы "+", " - " и "=", но чаще применяются составные операторы "+=" и "-=".)

Делегат с многоадресной передачей имеет одно ограничение: он должен возвращать тип void.

Пример 3. Рассмотрим пример многоадресной передачи. Это — переработанный вариант предыдущих примеров, в котором тип string для значений, возвращаемых методами обработки строк, заменен типом void, а для возврата модифицированных строк используется ref-параметр. Результат выполнения примера представлен на рис. 2.

```
using System;
namespace ConsoleApplication1
{
```

```
delegate void strMod(ref string str); // Объявляем делегата
class StringOps
{
    // Метод заменяет пробелы дефисами
    public static void replaceSpaces(ref string a)
    {
        Console.WriteLine(" Замена пробелов дефисами.");
        a = a.Replace(' ', '-');
    }
    // Метод удаляет пробелы
    public static void removeSpaces(ref string a)
    {
        string temp = "";
        Console.WriteLine(" Удаление пробелов.");
        for (int i = 0; i < a.Length; i++)
            if (a[i] != ' ')
                temp += a[i];
        a = temp;
    }
    // Метод реверсирует строку
    public static void reverse(ref string a)
    {
        string temp = "";
        Console.WriteLine(" Реверсирование строки.");
        for (int j = 0, i = a.Length - 1; i >= 0; i--, j++)
            temp += a[i];
        a = temp;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.Title = "Пример №3";
        Console.BackgroundColor = ConsoleColor.White; Console.Clear();
        Console.ForegroundColor = ConsoleColor.Black;
        // Создаем экземпляры делегатов
        strMod strOp;
        strMod replaceSp = new strMod(StringOps.replaceSpaces);
        strMod removeSp = new strMod(StringOps.removeSpaces);
        strMod reverseStr = new strMod(StringOps.reverse);
        string str = "Это простой тест.";
        // Организация многоадресной передачи
        strOp = replaceSp;
        strOp += reverseStr;
        // Вызов делегата с многоадресной передачей
        strOp(ref str);
        Console.WriteLine(" Результирующая строка: " + str);
        Console.WriteLine();
        // Удаляем метод замены пробелов и добавляем метод их удаления
        strOp -= replaceSp;
        strOp += removeSp;
        str = "Это простой тест."; // Восстановление исходной строки
                                   // Вызов делегата с многоадресной передачей
        strOp(ref str);
```

```
            Console.WriteLine(" Результирующая строка: " + str);
            Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
            Console.Read();
        }
    }
}
```

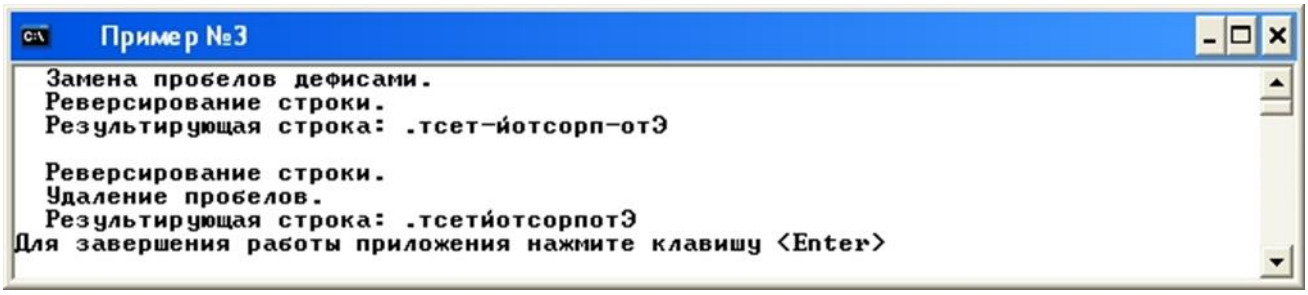


Рис. 2. Результат выполнения примера №3.

В методе Main() создаются четыре экземпляра делегата. Первый, strOp, имеет null-значение. Три других ссылаются на методы модификации строк. Затем организуется делегат для многоадресной передачи, который вызывает методы removeSpaces() и reverse(), с помощью операторов программы:

```
strOp = repiaceSp;
strOp += reverseStr; // в цепочку вызовов добавляется ссылка reverseStr
```

Цепочки вызовов, организованные с помощью делегата, — мощный механизм, который позволяет определять набор методов, выполняемых "единым блоком". Цепочки делегатов имеют особое значение для событий.

3.2 События

На основе делегатов построено еще одно важное средство C#: **событие** (event) – это по сути автоматическое уведомление о выполнении некоторого действия.

События работают следующим образом. Объект, которому необходима информация о некотором событии, регистрирует обработчик для этого события. Когда ожидаемое событие происходит, вызываются все зарегистрированные обработчики . Обработчики событий представляются делегатами.

События — это члены класса, которые объявляются с использованием ключевого слова **event**.

Наиболее распространенная форма объявления события имеет следующий вид:

```
event <событийный_делегат> <объект>;
```

где элемент <**событийный_делегат**> означает имя делегата, используемого для поддержки объявляемого события, а элемент <**объект**> — это имя создаваемого событийного объекта.

3.2.1 Создание событий

Пример 4. Рассмотрим простой пример, демонстрирующий использование простейшего события.

```
using System;
namespace ConsoleApplication1
{
```

```
    delegate void MyEventHandler(); // Объявляем делегата для события
                                    // Объявляем класс события
    class MyEvent
    {
        public event MyEventHandler SomeEvent;
        // Этот метод вызывается для генерирования события
        public void OnSomeEvent()
        {
            SomeEvent?.Invoke(); //можно if (SomeEvent != null) SomeEvent();
        }
    }
    class EventDemo
    {
        // Обработчик события
        public static void handler()
        {
            Console.WriteLine(" Произошло событие.");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = "Пример №4";
            Console.BackgroundColor = ConsoleColor.White;
            Console.Clear();
            Console.ForegroundColor = ConsoleColor.Black;
            MyEvent evt = new MyEvent();
            // Добавляем метод handler() в список события
            evt.SomeEvent += new MyEventHandler(EventDemo.handler);
            // Генерируем событие
            evt.OnSomeEvent();
            Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
            Console.Read();
        }
    }
}
```

Программа начинается с объявления делегата для обработчика события:

```
delegate void MyEventHandler();
```

Так как все события активизируются посредством делегата, то событийный делегат определяет сигнатуру для события. В данном случае параметры отсутствуют, однако событийные параметры разрешены. Поскольку события обычно предназначены для многоадресной передачи, они должны возвращать значение типа **void**.

Затем создается класс события **MyEvent**, в котором в первую очередь объявляется событийный объект **SomeEvent**:

```
public event MyEventHandler SomeEvent;
```

Кроме того, внутри класса **MyEvent** объявляется метод **OnSomeEvent()**, который в этой программе вызывается, чтобы сигнализировать о событии. Телом метода является оператор, который вызывает обработчик события посредством делегата **SomeEvent**:

```
if(SomeEvent != null) SomeEvent();
```

Обработчик события вызывается только в том случае, если делегат **SomeEvent** не равен значению **null**. Поскольку другие части программы, чтобы получить уведомлении о событии, должны зарегистрироваться, можно сделать так, чтобы метод OnSomeEvent() был вызван до регистрации любого обработчика события. Чтобы предотвратить вызов null-объекта, событийный делегат необходимо протестировать и убедиться в том, что он не равен null-значению.

В классе **EventDemo** создается обработчик события **handler()**, который просто отображает сообщение. В методе **Main()** создается объект класса **`MyEvent, MyEvent evt = new MyEvent();`** а метод **handler()** регистрируется в качестве обработчика этого события:

```
evt.SomeEvent += new MyEventHandler(handler);
```

Обработчик добавляется в список с использованием составного оператора "+=". Следует отметить, что события поддерживают только операторы "+=" и " -= ". В нашем примере метод handler() является статическим, но в общем случае обработчики событий могут быть методами экземпляров классов.

При выполнении оператора **`evt.OnSomeEvent();`** "происходит" событие.

При вызове метода **OnSomeEvent()** вызываются все зарегистрированные обработчики событий. В данном примере зарегистрирован только один обработчик. Результат выполнения программы представлен на рис. 3.

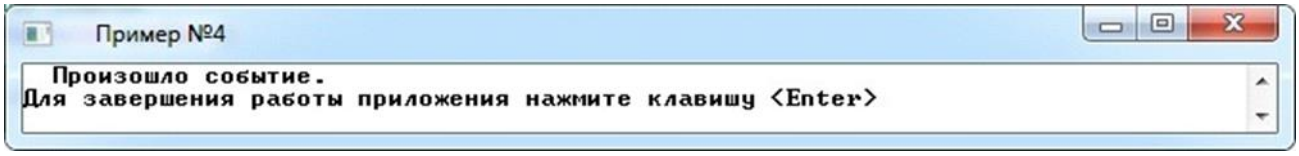


Рис. 3. Результат выполнения примера №4.

Подобно делегатам события могут предназначаться для многоадресной передачи. В этом случае на одно уведомление о событии может отвечать несколько объектов.

Пример 5. Рассмотрим простой пример, демонстрирующий использование простейшего события, предназначенного для многоадресной передачи. В этом примере создаются два дополнительных класса X и Y, в которых также определяются обработчики событий, совместимые с сигнатурой делегата MyEventHandler. Следовательно, эти обработчики могут стать частью цепочки событийных вызовов. Так как обработчики в классах X и Y не являются статическими, то сначала должны быть созданы объекты каждого класса, после чего в цепочку событийных вызовов должен быть добавлен обработчик, связанный с каждым экземпляром класса. Результат выполнения программы представлен на рис. 4.

```
using System;
namespace ConsoleApplication1
{
    delegate void MyEventHandler(); // Объявляем делегата для события
                                    // Объявляем класс события
    class MyEvent
    {
        public event MyEventHandler SomeEvent;
        // Этот метод вызывается для генерирования события
        public void OnSomeEvent()
        {
            SomeEvent?.Invoke(); //можно if (SomeEvent != null) SomeEvent();
        }
```

```
    }
    class X
    {
        public void Xhandler()
        {
            Console.WriteLine(" Событие, полученное объектом X.");
        }
    }
    class Y
    {
        public void Yhandler()
        {
            Console.WriteLine(" Событие, полученное объектом Y.");
        }
    }
    class EventDemo
    {
        public static void handler()
        {
            Console.WriteLine(" Событие, полученное классом EventDemo.");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = " Пример №5";
            Console.BackgroundColor = ConsoleColor.White;
            Console.Clear();
            Console.ForegroundColor = ConsoleColor.Black;
            MyEvent evt = new MyEvent();
            X xOb = new X(); Y yOb = new Y();
            // Добавляем обработчики в список события
            evt.SomeEvent += new MyEventHandler(EventDemo.handler);
            evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
            evt.SomeEvent += new MyEventHandler(yOb.Yhandler);
            // Генерируем событие
            evt.OnSomeEvent();
            Console.WriteLine();
            // Удаляем один обработчик
            evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
            evt.OnSomeEvent();
            Console.Write("Для завершения работы приложения нажмите <Enter>");
            Console.Read();
        }
    }
}
```

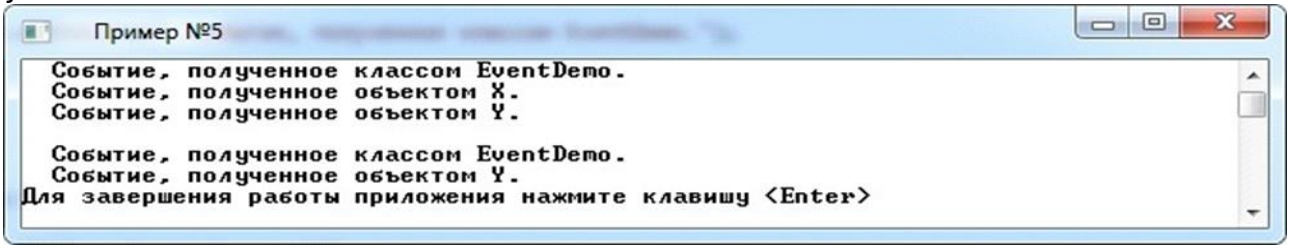


Рис. 4. Результат выполнения примера №5.

3.2.1 Методы, используемые в качестве обработчиков событий

Несмотря на то, что и методы экземпляров классов, и статические методы могут служить обработчиками событий, в их использовании в этом качестве есть существенные различия. Если в качестве обработчика используется статический метод, уведомление о событии применяется к классу (и неявно ко всем объектам этого класса). Если же в качестве обработчика событий используется метод экземпляра класса, события посылаются к конкретным экземплярам этого класса. Следовательно, каждый объект класса, который должен получать уведомление о событии, необходимо регистрировать в отдельности. На практике в большинстве случаев роль обработчиков событий играют методы экземпляров классов.

3.2.2 Метод экземпляра класса в роли обработчика события

Пример 6. В нижеприведенной программе создается класс X, в котором в качестве обработчика событий определен метод экземпляра. Это значит, что для получения информации о событиях каждый объект класса X необходимо регистрировать отдельно. Для демонстрации этого факта программа готовит уведомление о событии для многоадресной передачи трем объектам типа X. Результат выполнения программы представлен на рис. 5.

```
using System;
namespace ConsoleApplication1
{
    delegate void MyEventHandler(); // Объявляем делегата для события
                                    // Объявляем класс события
    class MyEvent
    {
        public event MyEventHandler SomeEvent;
        // Этот метод вызывается для генерирования события
        public void OnSomeEvent()
        {
            SomeEvent?.Invoke(); //можно if (SomeEvent != null) SomeEvent();
        }
    }
    class X
    {
        int id;
        public X(int x) { id = x; }
        // Метод экземпляра, используемый в качестве обработчика событий
        public void Xhandler()
        {
            Console.WriteLine(" Событие принято объектом " + id);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = " Пример №6";
            Console.BackgroundColor = ConsoleColor.White;
            Console.Clear();
            Console.ForegroundColor = ConsoleColor.Black;
```

```
            MyEvent evt = new MyEvent();
            X o1 = new X(1);
            X o2 = new X(2);
            X o3 = new X(3);
            evt.SomeEvent += new MyEventHandler(o1.Xhandler);
            evt.SomeEvent += new MyEventHandler(o2.Xhandler);
            evt.SomeEvent += new MyEventHandler(o3.Xhandler);
            // Генерируем событие
            evt.OnSomeEvent();
            Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
            Console.Read();
        }
    }
}
```

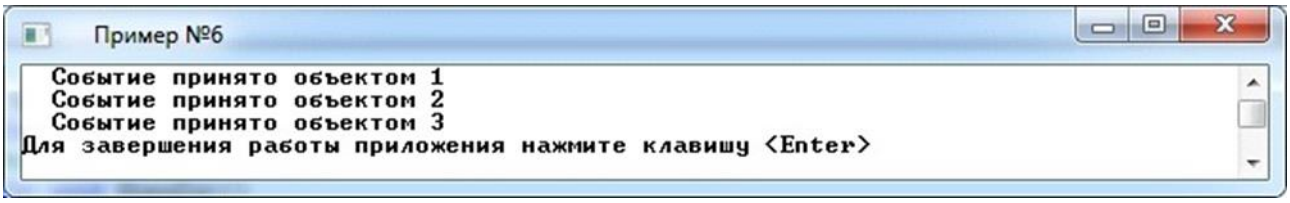


Рис. 5. Результат выполнения примера №6.

Как подтверждают эти результаты, каждый объект заявляет о своей заинтересованности в событии и получает о нем отдельное уведомление.

3.2.3 Статический метод класса в роли обработчика события

Если же в качестве обработчика событий используется статический метод, то, как показано в следующей программе, события обрабатываются независимо от объекта.

Пример 7. При использовании в качестве обработчиков событий статического метода уведомление о событиях получает класс. Результат выполнения программы представлен на рис. 6.

```
using System;
namespace ConsoleApplication1
{
    delegate void MyEventHandler(); // Объявляем делегата для события
                                    // Объявляем класс события
    class MyEvent
    {
        public event MyEventHandler SomeEvent;
        // Этот метод вызывается для генерирования события
        public void OnSomeEvent()
        {
            if (SomeEvent != null) SomeEvent();
        }
    }
    class X
    {
        // Это статический метод, используемый в качестве обработчика события
        public static void Xhandler()
        {
            Console.WriteLine(" Событие получено классом.");
```

```
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = "Пример №7";
            Console.BackgroundColor = ConsoleColor.White; Console.Clear();
            Console.ForegroundColor = ConsoleColor.Black;
            MyEvent evt = new MyEvent();
            evt.SomeEvent += new MyEventHandler(X.Xhandler);
            // Генерируем событиеevt.OnSomeEvent();
            Console.Write("Для завершения работы приложения нажмите <Enter>");
            Console.Read();
        }
    }
}
```

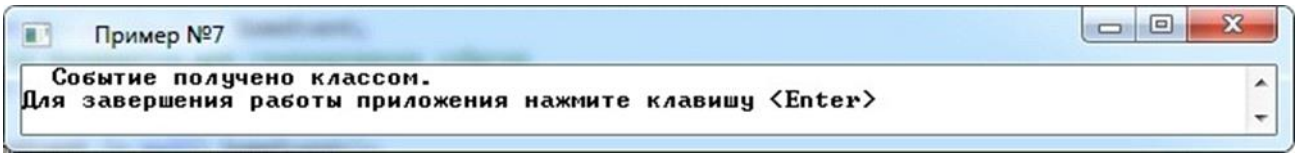


Рис. 6. Результат выполнения примера №7.

В программе не создается ни одного объекта типа X. Но поскольку handler() — статический метод класса X, его можно связать с событием SomeEvent и обеспечить его выполнение при вызове метода OnSomeEvent().

3.2.4 Использование событийных средств доступа

Предусмотрены две формы записи инструкций, связанных с событиями. Форма, используемая в предыдущих примерах, обеспечивала создание событий, которые автоматически управляют списком вызова обработчиков, включая такие операции, как добавление обработчиков в список и удаление их из списка. Таким образом, можно было не беспокоиться о реализации операций по управлению этим списком. Поэтому такие типы событий, безусловно, являются наиболее применимыми. Однако можно и самим организовать ведение списка обработчиков событий, чтобы, например, реализовать специализированный механизм хранения событий.

Чтобы управлять списком обработчиков событий, используйте вторую форму event-инструкции, которая позволяет использовать средства доступа к событиям. Эти средства доступа дают возможность управлять реализацией списка обработчиков событий.

Форма имеет следующий вид:

```
event <событийный_делегат> <имя_события>
{
      add
      {
      // Код добавления события в цепочку событий
      }
      remove
      {
```

```
        // Код удаления события из цепочки событий
        }
}
```

Эта форма включает два средства доступа к событиям: **add** и **remove**. Средство доступа **add** вызывается в случае, когда с помощью оператора "+=" в цепочку событий добавляется новый обработчик, а средство доступа **remove** вызывается, когда с помощью оператора " - = " из цепочки событий удаляется новый обработчик.

Средство доступа **add** или **remove** при вызове получает обработчик, который необходимо добавить или удалить, в качестве параметра. Этот параметр, как и в случае использования других средств доступа, называется **value**. При реализации средств доступа **add** и **remove** можно задать собственную схему хранения обработчиков событий. Например, для этого можно использовать массив, стек или очередь.

Пример 8. Рассмотрим пример использования событийных средств доступа. Здесь для хранения обработчиков событий взят массив. Поскольку этот массив содержит три элемента, в любой момент времени в событийной цепочке может храниться только три обработчика событий. Результат выполнения программы представлен на рис. 7.

```
using System;
namespace ConsoleApplication1
{
    delegate void MyEventHandler(); // Объявляем делегат для события
                                    // Объявляем класс события для хранения трех
                                    // обработчиков событий
    class MyEvent
    {
        MyEventHandler[] evnt = new MyEventHandler[3];
        public event MyEventHandler SomeEvent
        {
            // Добавляем обработчик события в список
            add
            {
                int i;
                for (i = 0; i < 3; i++) if (evnt[i] == null)
                    {
                        evnt[i] = value; break;
                    }
                if (i == 3)
                    Console.WriteLine(" Список обработчиков событий полон.");
            }
            // Удаляем обработчик события из списка
            remove
            {
                int i;
                for (i = 0; i < 3; i++) if (evnt[i] == value)
                    {
                        evnt[i] = null; break;
                    }
                if (i == 3) Console.WriteLine(" Обработчик события не найден.");
            }
        }
        // Этот метод вызывается для генерирования событий
        public void OnSomeEvent()
```

```
        {
            for (int i = 0; i < 3; i++)
                if (evnt[i] != null) evnt[i]();
        }
        // Создаем классы, которые используют делегата 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 Program
        {
            static void Main(string[] args)
            {
                Console.Title = "Пример №8";
                Console.BackgroundColor = ConsoleColor.White;
                Console.Clear();
                Console.ForegroundColor = ConsoleColor.Black; MyEvent evt = new
MyEvent();
                W wOb = new W();
                X xOb = new X();
                Y yOb = new Y();
                Z zOb = new Z();
                // Добавляем обработчики в список
                Console.WriteLine(" Добавление обработчиков событий.");
                evt.SomeEvent += new MyEventHandler(wOb.Whandler);
                evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
                evt.SomeEvent += new MyEventHandler(yOb.Yhandler);
                // Этот обработчик сохранить нельзя – список полон
                evt.SomeEvent += new MyEv-entHandler(zOb.Zhandler);
                Console.WriteLine();
                // Генерируем события
```

```
                evt.OnSomeEvent();
                Console.WriteLine();
                // Удаляем обработчик из списка
                Console.WriteLine(" У даляем обработчик xOb.Xhandler.");
                evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
                evt.OnSomeEvent();
                Console.WriteLine();
                // Пытаемся удалить его еще раз
                Console.WriteLine(" Попытка повторно удалить обработчик
xOb.Xhandler.");
                evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
                evt.OnSomeEvent();
                Console.WriteLine();
                // Теперь добавляем обработчик Zhandler
                Console.WriteLine(" Добавляем обработчик zOb.Zhandler.");
                evt.SomeEvent += new MyEventHandler(zOb.Zhandler);
                evt.OnSomeEvent();
                Console.Write("Для завершения работы приложения нажмите клавишу
<Enter> );
                Console.Read();
            }
        }
    }
}
```

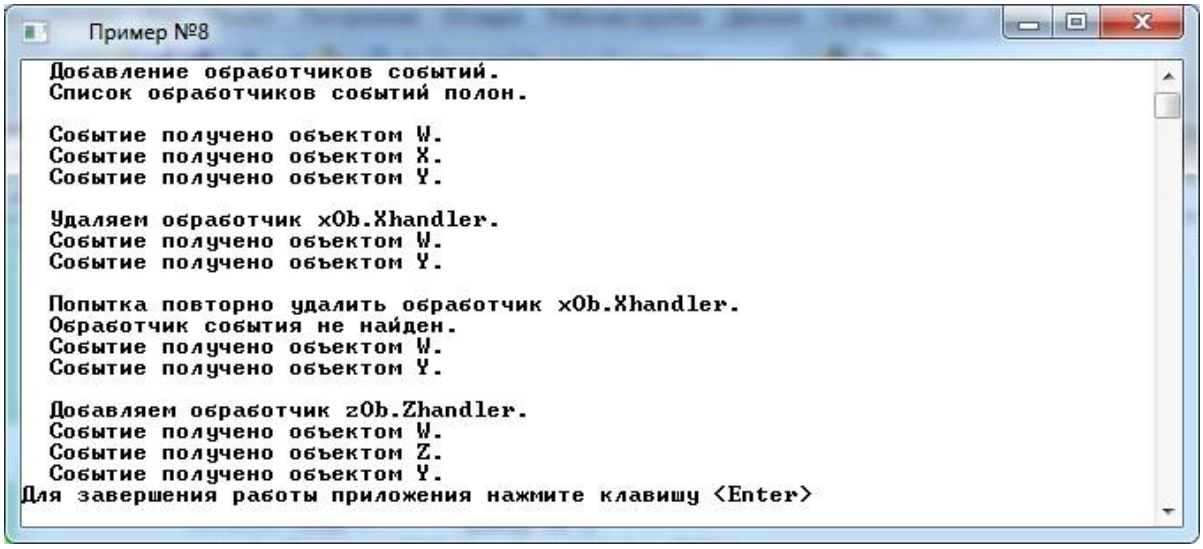


Рис. 7. Результат выполнения примера №8.

В начале программы определяется делегат обработчика события **MyEventHandler**. В классе **MyEvent** объявляется трехэлементный массив обработчиков событий

```
evnt: MyEventHandler[] evnt = new MyEventHandler[3];
```

Этот массив предназначен для хранения обработчиков событий, которые добавлены в цепочку событий. Элементы массива **evnt** инициализируются null-значениями по умолчанию.

При добавлении в список обработчика событий вызывается **add**-средство, и ссылка на этот обработчик (содержащаяся в параметре **value**) помещается в первый встретившийся неиспользуемый элемент массива **evnt**. Если свободных элементов нет, выдается сообщение об ошибке. Поскольку массив **evnt** рассчитан на хранение лишь трех элементов, он может принять только три обработчика событий. При удалении заданного обработчика событий вызывается **remove**-средство, и в массиве

evnt выполняется поиск ссылки на обработчик, переданной в параметре **value**. Если ссылка найдена, в соответствующий элемент массива помещается значение null, что равнозначно удалению обработчика из списка.

При генерировании события вызывается метод **OnSomeEvent()**. Он в цикле просматривает массив **evnt**, вызывая по очереди каждый обработчик событий.

3.2.5 Рекомендации по обработке событий в среде .NET Framework

C# позволяет программисту создавать события любого типа. Однако в целях компонентной совместимости со средой .NET Framework необходимо следовать рекомендациямMicrosoft.

Центральное место занимает требование того, чтобы обработчики событий имели два параметра. Первый должен быть ссылкой на объект, который будет генерировать событие. Второй должен иметь тип **EventArgs** и содержать остальную информацию, необходимую обработчику, то есть .NET-совместимые обработчики событий должны иметь следующую общую форму записи:

```
void handler(object source, EventArgs arg)
{
// . . .
}
```

где параметр **source** передается вызывающим кодом, параметр типа **EventArgs** содержит дополнительную информацию, которую в случае ненадобности можно проигнорировать.

Класс **EventArgs** не содержит полей, которые используются при передаче дополнительных данных обработчику; он используется в качестве базового класса, из которого можно выводить класс, содержащий необходимые поля. Так как многие обработчики обходятся без дополнительных данных, в класс **EventArgs** включено статическое поле **Empty**, которое задает объект, не содержащий никаких данных.

Пример 9. Нижеприведенный пример демонстрирует использование .NET-совместимого события. В производном классе **MyEventArgs** (базовый класс **EventArgs**) добавлено только одно "собственное" поле — **eventnum**. В соответствии с требованиями .NET Framework делегат для обработчика событий **MyEventHandler** принимает два параметра: первый из них представляет собой объектную ссылку на генератор событий, а второй — ссылку на класс **EventArgs** или производный от класса **EventArgs** (используется ссылка на объект типа **MyEventArgs**). Результат выполнения программы представлен на рис. 8.

```
using System;
namespace ConsoleApplication1
{
    class MyEventArgs : EventArgs
    {
        public int eventnum;
    }
    // Объявляем делегат для события
    delegate void MyEventHandler(object source, MyEventArgs arg);
    // Объявляем класс события
    class MyEvent
    {
        static int count = 0;
        public event MyEventHandler SomeEvent;
        // Этот метод генерирует SomeEvent-событие
        public void OnSomeEvent()
        {
```

```
            MyEventArgs arg = new MyEventArgs();
            if (SomeEvent != null)
            {
                arg.eventnum = count++;
                SomeEvent(this, arg);
            }
        }
    }
    class X
    {
        public void handler(object source, MyEventArgs arg)
        {
            Console.WriteLine(" Событие " + arg.eventnum + " получено объектом X.");
            Console.WriteLine(" Источником является класс " + source + ".");
            Console.WriteLine();
        }
    }
    class Y
    {
        public void handler(object source, MyEventArgs arg)
        {
            Console.WriteLine(" Событие " + arg.eventnum + " получено объектом Y.");
            Console.WriteLine(" Источником является класс " + source + ".");
            Console.WriteLine();
        }
    }
    namespace Primer9
    {
        class Program
        {
            static void Main()
            {
                Console.Title = "Пример №9";
                Console.BackgroundColor = ConsoleColor.White;
                Console.Clear();
                Console.ForegroundColor = ConsoleColor.Black;
                X obi = new X();
                Y ob2 = new Y();
                MyEvent evt = new MyEvent();
                // Добавляем обработчик handler() в список событий
                evt.SomeEvent += new MyEventHandler(obi.handler);
                evt.SomeEvent += new MyEventHandler(ob2.handler);
                // Генерируем событие
                evt.OnSomeEvent();
                evt.OnSomeEvent();
                Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
                Console.Read();
            }
        }
    }
}
```

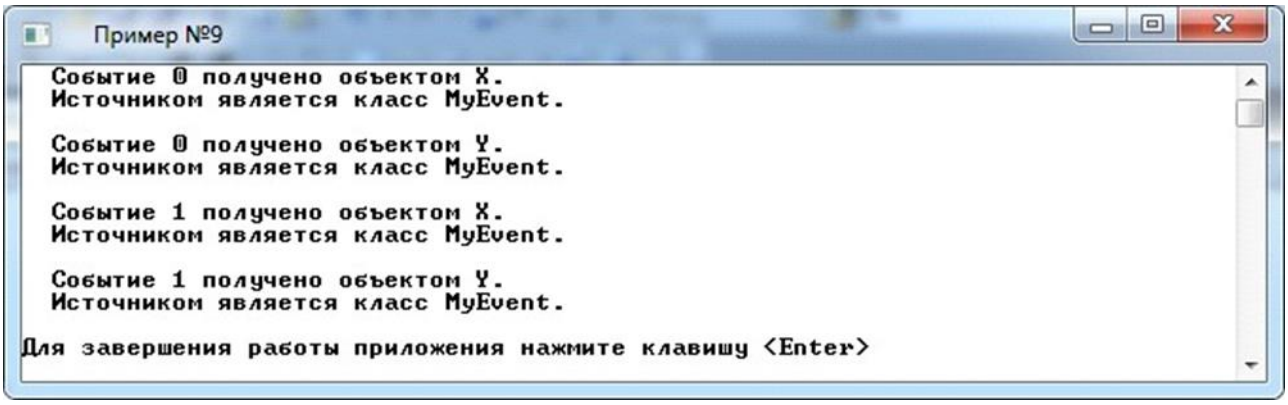


Рис. 8. Результат выполнения примера №9.

3.2.6 Использование встроенного делегата EventHandler

Для многих событий параметр типа EventArgs не используется. Для упрощения процесса создания кода в таких ситуациях среда .NET Framework включает встроенный тип делегата, именуемый EventHandler. Его можно использовать для объявления обработчиков событий, которым не требуется дополнительная информация.

Пример 10. Нижеприведенный пример демонстрирует использование встроенного делегата EventHandler. Результат выполнения программы представлен на рис. 9.

```
using System;
namespace ConsoleApplication1
{
    class MyEvent
    {
        public event EventHandler SomeEvent;
        // Этот метод вызывается для генерирования SomeEvent-событие
        public void OnSomeEvent()
        {
            if (SomeEvent != null)
                SomeEvent(this, EventArgs.Empty);
        }
    }
    namespace Primer10
    {
        class Program
        {
            public static void handler(object source, EventArgs arg)
            {
                Console.WriteLine(" Событие произошло.");
                Console.WriteLine(" Источником является класс " + source + ".");
            }
            static void Main()
            {
                Console.Title = "Пример №10";
                Console.BackgroundColor = ConsoleColor.White;
                Console.Clear();
                Console.ForegroundColor = ConsoleColor.Black;
                MyEvent evt = new MyEvent();
                // Добавляем обработчик handler() в список событий
                evt.SomeEvent += new EventHandler(handler);
```

```
                // Генерируем событие
                evt.OnSomeEvent();
                Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
                Console.Read();
            }
        }
    }
}
```

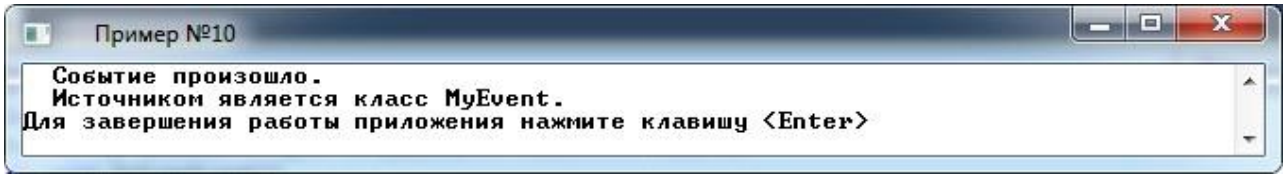


Рис. 9. Результат выполнения примера №10.

3.2.7 Пример использования событий

События часто используются в таких средах с ориентацией на передачу сообщений, как ОС Windows, в которой программа просто ожидает до тех пор, пока не получит сообщение, а затем выполняет соответствующие действия. Такая архитектура прекрасно подходит для обработки событий в стиле языка C#, позволяя создавать обработчики событий для различных сообщений и просто вызывать обработчик при получении определенного сообщения. Например, с некоторым событием можно было бы связать сообщение, получаемое в результате щелчка левой кнопкой мыши. Тогда после щелчка левой кнопкой мыши все зарегистрированные обработчики будут уведомлены о приходе этого сообщения.

Пример 11. Рассмотрим пример обработки события, связанного с нажатием клавиши на клавиатуре. Событие называется **KeyPress**, и при каждом нажатии клавиши оно генерируется посредством вызова метода **OnKeyPress()**.

Программа начинается с объявления класса **KeyEventArgs**, который используется для передачи сообщения о нажатии клавиши обработчику событий. Затем делегат **KeyHandler** определяет обработчик для событий, связанных с нажатием клавиши на клавиатуре. Эти события инкапсулируются в классе **KeyEvent**.

Программа для обработки нажатий клавиш создает два класса: **ProcessKey** и **CountKeys**. Класс **ProcessKey** включает обработчик с именем **keyhandler()**, который отображает сообщение о нажатии клавиши. Класс **CountKeys** предназначен для хранения текущего количества нажатых клавиш.

В методе **Main()** создается объект класса **KeyEvent**. Затем создаются объекты классов **ProcessKey** и **CountKeys**, а ссылки на их методы **keyhandler()** добавляются в список вызовов, реализуемый с помощью событийного объекта **kevt.KeyPress**. Затем начинает работать цикл, в котором при каждом нажатии клавиши вызывается метод **kevt.OnKeyPress()**, в результате чего зарегистрированные обработчики уведомляются о событии.

Результат выполнения программы представлен на рис. 10.

```
using System;
namespace ConsoleApplication1
{
    // Объявляем собственный класс EventArgs, который будет хранить код клавиши
    class KeyEventArgs : EventArgs
    {
        public char ch;
    }
    // Объявляем делегат для события
```

```
    delegate void KeyHandler(object source, KeyEventArgs arg);
    // Объявляем класс события, связанного с нажатием клавиши на клавиатуре
    class KeyEvent
    {
        public event KeyHandler KeyPress;
        // Этот метод вызывается при нажатии какой-нибудь клавиши
        public void OnKeyPress(char key)
        {
            KeyEventArgs k = new KeyEventArgs();
            if (KeyPress != null)
            {
                k.ch = key;
                KeyPress(this, k);
            }
        }
    }
    // Класс, который принимает уведомления о нажатии клавиши
    class ProcessKey
    {
        public void keyhandler(object source, KeyEventArgs arg)
        {
            Console.WriteLine("Получено сообщение о нажатии клавиши : " + arg.ch);
        }
    }
    // Еще один класс, который принимает уведомления о нажатии клавиши
    class CountKeys
    {
        public int count = 0;
        public void keyhandler(object source, KeyEventArgs arg)
        {
            count++;
        }
    }
    namespace Primer11
    {
        class Program
        {
            static void Main()
            {
                Console.Title = "Демонстрация события о нажатии клавиши";
                Console.BackgroundColor = ConsoleColor.White;
                Console.Clear();
                Console.ForegroundColor = ConsoleColor.Black;
                KeyEvent kevt = new KeyEvent();
                ProcessKey pk = new ProcessKey();
                CountKeys ck = new CountKeys();
                char ch;
                kevt.KeyPress += new KeyHandler(pk.keyhandler);
                kevt.KeyPress += new KeyHandler(ck.keyhandler);
                Console.WriteLine("Введите несколько символов. " + "Для останова
введите точку.");
                do
                {
                    ch = (char)Console.Read();
                    kevt.OnKeyPress(ch);
```

```
                } while (ch != '.');
                Console.WriteLine(" Было нажато " + ck.count + " клавиш.");
                Console.Write("Для завершения работы приложения нажмите клавишу
<Enter>");
                Console.Read();
            }
        }
    }
}
```

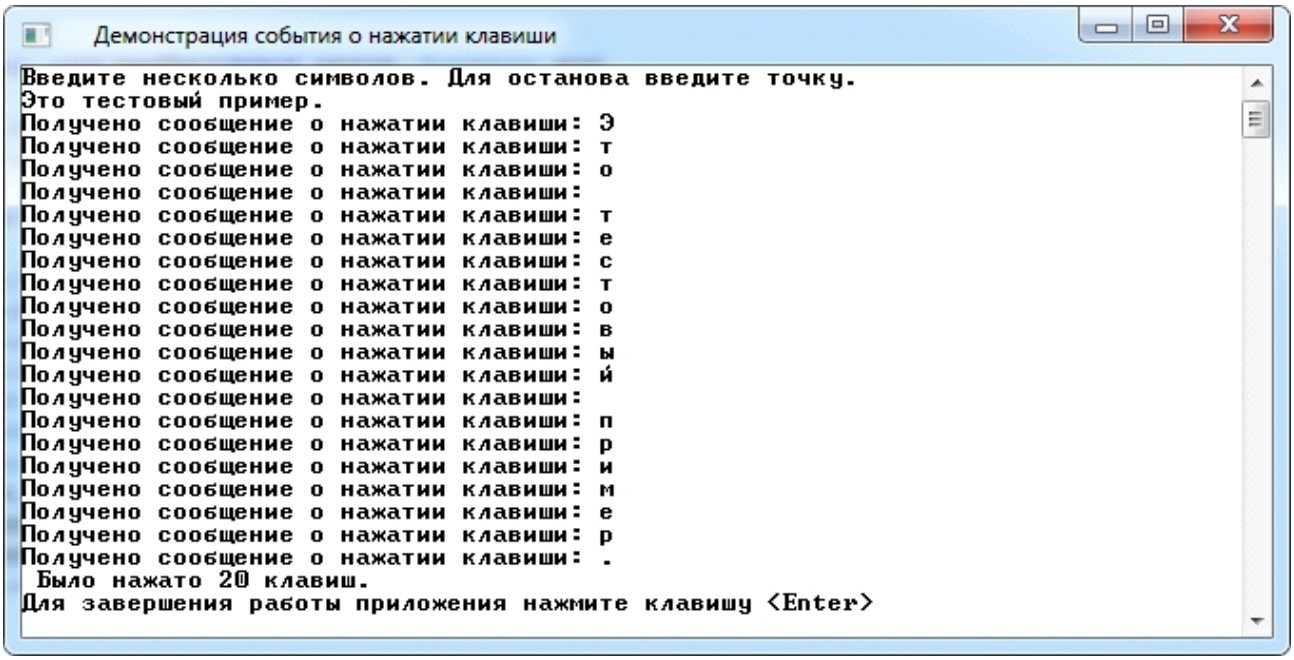
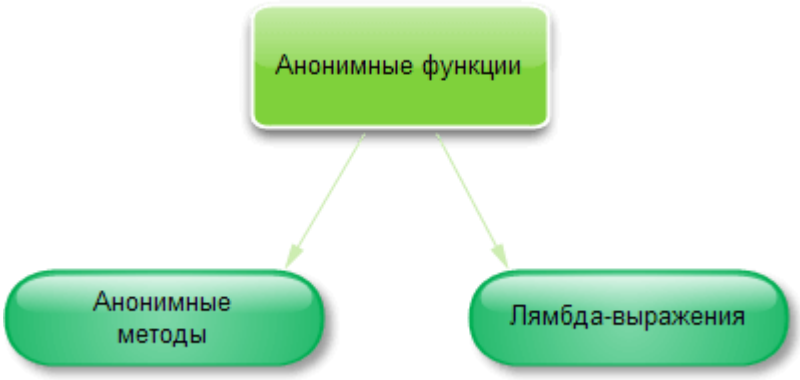


Рис. 10. Результат выполнения примера №11.

3.3 Анонимные методы

Метод, на который ссылается делегат, нередко используется только для этой цели. Иными словами, 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***, по существу, представляет собой безымянный кодовый блок, передаваемый конструктору делегата. Преимущество анонимной функции состоит, в частности, в ее простоте. Благодаря ей отпадает необходимость объявлять отдельный метод, единственное назначение которого состоит в том, что он передается делегату.

Начиная с версии 3.0, в C# предусмотрены две разновидности анонимных функций - анонимные методы и лямбда-выражения:

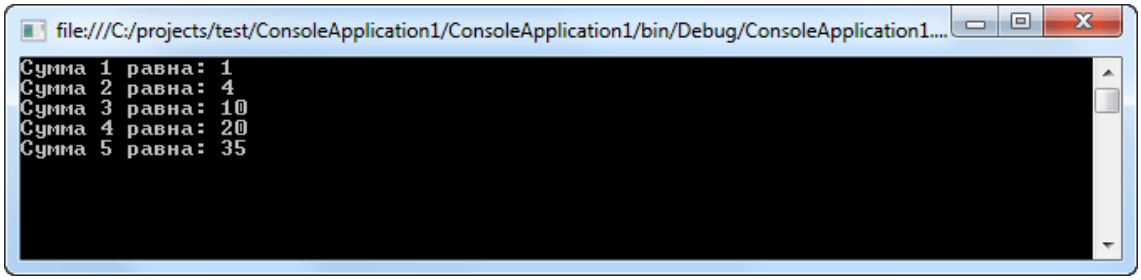


Анонимные методы были внедрены в C# еще в версии 2.0, а лямбда-выражения – в версии 3.0. В целом лямбда-выражение совершенствует принцип действия анонимного метода и в настоящее время считается более предпочтительным для создания анонимной функции. Но анонимные методы широко применяются в существующем коде C# и поэтому по-прежнему являются важной составной частью C#. А поскольку анонимные методы предшествовали появлению лямбда-выражений, то ясное представление о них позволяет лучше понять особенности лямбда-выражений. К тому же 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. Для создания анонимного метода достаточно указать кодовый блок после ключевого слова delegate. Покажем, как это делается, на конкретном примере:

```
using System;
using System.Collections.Generic;

namespace ConsoleApplication1
{
    delegate int Sum(int number);

    class Program
    {
        static Sum SomeVar()
        {
            int result = 0;

            // Вызов анонимного метода
            Sum del = delegate (int number)
            {
                for (int i = 0; i <= number; i++)
                    result += i;
                return result;
            };
            return del;
        }

        static void Main()
        {
            Sum del1 = SomeVar();
            for (int i = 1; i <= 5; i++)
            {
                Console.WriteLine("Сумма {0} равна: {1}",i,del1(i));
```

```
            }
            Console.ReadLine();
        }
    }
}
```



Обратите внимание на получившийся результат. Локальная переменная, в область действия которой входит анонимный метод, называется *внешней переменной*. Такие переменные доступны для использования в анонимном методе. И в этом случае внешняя переменная считается захваченной. Захваченная переменная существует до тех пор, пока захвативший ее делегат не будет собран в "мусор". Поэтому если локальная переменная, которая обычно прекращает свое существование после выхода из кодового блока, используется в анонимном методе, то она продолжает существовать до тех пор, пока не будет уничтожен делегат, ссылающийся на этот метод.

Захват локальной переменной может привести к неожиданным результатам, как в приведенном выше примере, где локальная переменная result не обнуляется после каждого вызова анонимного метода. В результате получается необычный результат суммы чисел.

3.4 Лямбда-выражения

Начиная с C# 3.0, доступен новый синтаксис для назначения реализации кода делегатам, называемый ***лямбда-выражениями (lambda expression)***. Лямбда-выражения могут использоваться везде, где есть параметр типа делегата.

Синтаксис лямбда-выражений проще синтаксиса анонимных методов. В случае если подлежащий вызову метод имеет параметры, а эти параметры не нужны, синтаксис анонимных методов проще, поскольку в этом случае указывать параметры не потребуется.

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. Обе разновидности лямбда-выражений рассматриваются далее по отдельности.

3.4.1 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. Ниже приведена общая форма одиночного лямбда-выражения, принимающего единственный параметр:

параметр => выражение

Если же требуется указать несколько параметров, то используется следующая форма:

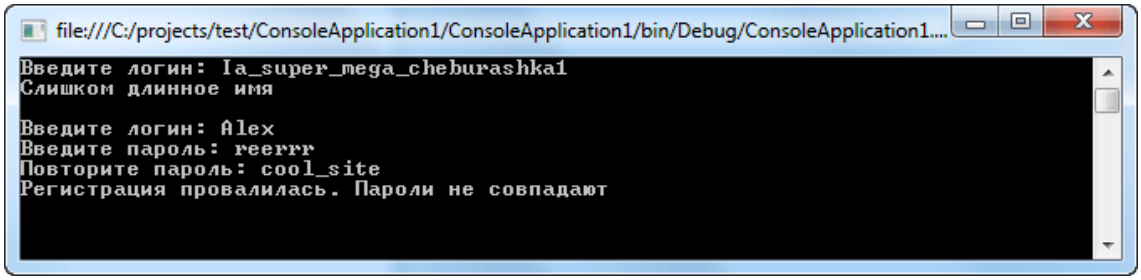
(список_параметров) => выражение

Таким образом, когда требуется указать два параметра или более, их следует заключить в скобки. Если же выражение не требует параметров, то следует использовать пустые скобки.

Лямбда-выражение применяется в два этапа. Сначала объявляется тип делегата, совместимый с лямбда-выражением, а затем экземпляр делегата, которому присваивается лямбда-выражение. После этого лямбда-выражение вычисляется при обращении к экземпляру делегата. Результатом его вычисления становится возвращаемое значение. Давайте рассмотрим пример:

```
using System;

namespace ConsoleApplication1
{
    // Создадим несколько делегатов имитирующих
    // простейшую форму регистрации
    delegate int LengthLogin(string s);
    delegate bool BoolPassword(string s1, string s2);

    class Program
    {
        private static void SetLogin()
        {
            Console.Write("Введите логин: ");
            string login = Console.ReadLine();

            // Используем лямбда-выражение
            LengthLogin lengthLoginDelegate = s => s.Length;

            int lengthLogin = lengthLoginDelegate(login);
            if (lengthLogin > 25)
            {
                Console.WriteLine("Слишком длинное имя\n");

     // Рекурсия на этот же метод, чтобы ввести заново логин
                SetLogin();
            }
        }

        static void Main()
        {
            SetLogin();

            Console.Write("Введите пароль: ");
            string password1 = Console.ReadLine();
```

```
                Console.Write("Повторите пароль: ");
                string password2 = Console.ReadLine();

                // Используем лямбда выражение
                BoolPassword bp = (s1, s2) => s1 == s2;

                if (bp(password1, password2))
                    Console.WriteLine("Регистрация удалась!");
                else
                    Console.WriteLine("Регистрация провалилась. Пароли не совпада-
ют");

                Console.ReadLine();
            }
        }
    }
```



3.4.2 Блочные лямбда-выражения

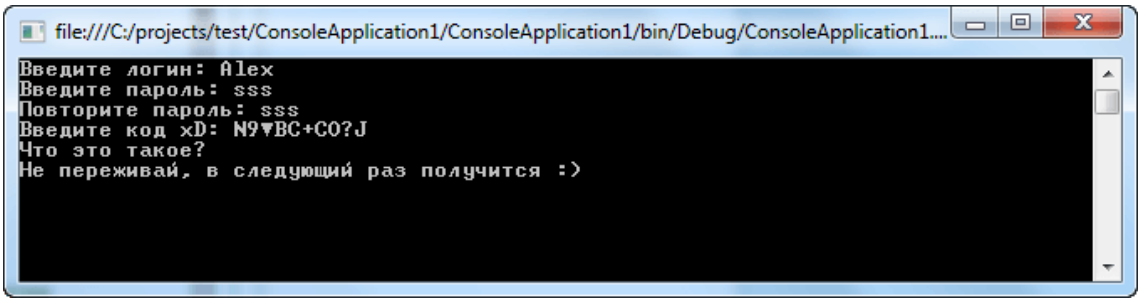
Второй разновидностью лямбда-выражений является **блочное лямбда-выражение**. Для такого лямбда-выражения характерны расширенные возможности выполнения различных операций, поскольку в его теле допускается указывать несколько операторов. Например, в блочном лямбда-выражении можно использовать циклы и условные операторы if, объявлять переменные и т.д. Создать блочное лямбда-выражение нетрудно. Для этого 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.

Давайте модифицируем предыдущий пример, добавив капчу в форму регистрации:

```
using System;

namespace ConsoleApplication1
{
    // Создадим несколько делегатов имитирующих
    // простейшую форму регистрации
    delegate int LengthLogin(string s);
    delegate bool BoolPassword(string s1, string s2);
    delegate void Captha(string s1, string s2);

    class Program
    {
        private static void SetLogin()
```

```
            {
                Console.Write("Введите логин: ");
                string login = Console.ReadLine();

                // Используем лямбда-выражение
                LengthLogin lengthLoginDelegate = s => s.Length;

                int lengthLogin = lengthLoginDelegate(login);
                if (lengthLogin > 25)
                {
                    Console.WriteLine("Слишком длинное имя\n");

        // Рекурсия на этот же метод, чтобы ввести заново логин
                    SetLogin();
                }
            }

            static void Main()
            {
                SetLogin();

                Console.Write("Введите пароль: ");
                string password1 = Console.ReadLine();
                Console.Write("Повторите пароль: ");
                string password2 = Console.ReadLine();

                // Используем лямбда выражение
                BoolPassword bp = (s1, s2) => s1 == s2;

                if (bp(password1, password2))
                {
                    Random ran = new Random();
                    string resCaptha = "";
                    for (int i = 0; i < 10; i++)
                        resCaptha += (char)ran.Next(0, 100);
              Console.WriteLine("Введите код xD: " + resCaptha);
                    string resCode = Console.ReadLine();

                    // Реализуем блочное лямбда-выражение
                    Captha cp = (s1, s2) =>
                    {
                     if (s1 == s2)
                      Console.WriteLine("Регистрация удалась!");
                        else
                            Console.WriteLine("Не переживай, в следующий раз полу-
чится :)");
                        return;
                    };
                    cp(resCaptha, resCode);
                }
                else
                    Console.WriteLine("Регистрация провалилась. Пароли не совпада-
ют");

                Console.ReadLine();
```

```
        }
    }
}
```


file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1....
Введите логин: Alex
Введите пароль: sss
Повторите пароль: sss
Введите код xD: N9▼BC+CO?J
Что это такое?
Не переживай, в следующий раз получится :)

4 Задания

Задание 1. Руководствуясь теоретическим материалом раздела 1 изучить возможности языка C# по созданию приложений, использующие делегаты, события и анонимные методы и выполнить примеры, описанные в этом разделе.

Задание 2. Используя обработчик события и событие **KeyPress** приложения «Демонстрация события о нажатии клавиши» модифицировать нижеприведенный код приложения «Создание массива символов» таким образом, чтобы выдача данных о клавиши и символе осуществлялась в обработчиках события.

```
            static void Main(string[] args)
            {
                Console.Title = "Создание массива символов";
                ConsoleKeyInfo cki;
                string Str = "";
                Console.WriteLine(" Ввод символов в массив. " + "(для прекращения ввода
нажмите клавишу <F1>)\n");
                do
                {
                    Console.Write("  Введите символ.  ");
                    cki = Console.ReadKey(true);
                    Console.Write("Вы нажали клавишу ");
                    Console.WriteLine("{0} (символ '{1}')", cki.Key, cki.KeyChar);
                    Str += cki.KeyChar;
                } while (cki.Key != ConsoleKey.F1);
                Console.WriteLine(" Массив символов: " + Str);
            }
```

Метод ReadKey() получает следующий нажатый пользователем символ или функциональную клавишу. Объект ConsoleKeyInfo содержит данные, описывающие константу ConsoleKey (элемент Key хранит имя клавиши) и символ Юникода (элемент KeyChar хранит символ).

Задание 3. Разработать приложение «Название автомобиля», с помощью которого можно отслеживать изменения в названии автомобиля. Обработчик события должен выдавать сообщение “Название изменилось!” при внесении изменения названия автомобиля.

Один из вариантов выполнения приложения представлен на рис. 11.

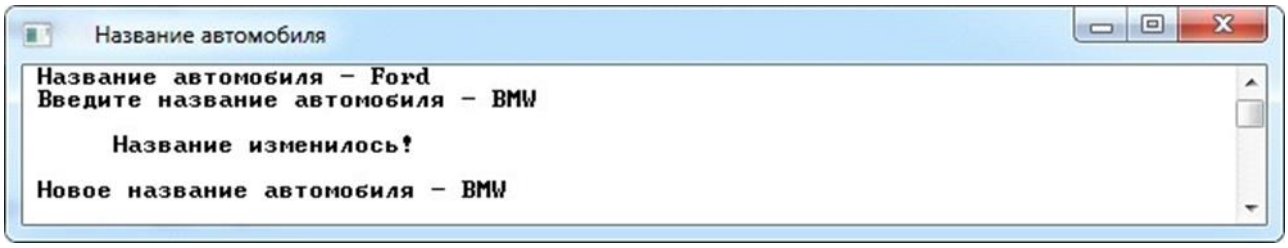


Рис. 11. Результат выполнения приложения «Название автомобиля»

Задание 4. Модифицировать приложение «Название автомобиля» таким образом, чтобы пользователю дать возможность принять решение об изменении или отказе от изменения названия автомобиля.

Один из вариантов выполнения приложения представлен на рис. 12. При нажатии клавиши <N> должно выдавать сообщение «Название не изменилось! Название автомобиля - Ford»

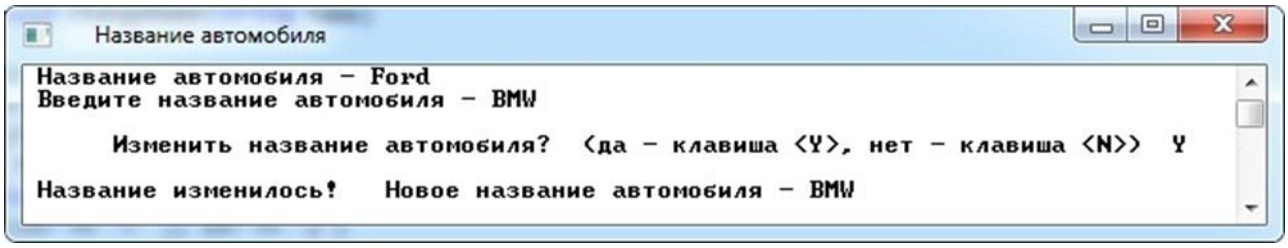


Рис. 12. Результат выполнения модифицированного приложения «Название автомобиля»

Задание 4. Изменить пример 1 таким образом, чтобы были использованы лямбда-выражения и анонимные методы.

Задание 5. Разработать программу, с использованием делегатов, событий, лямбда-выражений по следующей теме «**Имитация работы Инфокиоска банка». (повышенный уровень)**

5 Контрольные вопросы

1. Что понимается под термином «делегат»?
2. В чем состоят преимущества использования делегатов?
3. В какой момент осуществляется выбор вызываемого метода в случае использования делегатов?
4. Что является значением делегата?
5. Приведите синтаксис делегата в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
6. Что понимается под термином «многоадресность»? В чем состоит практическое значение многоадресности?
7. Каким образом осуществляется создание цепочки методов для многоадресных делегатов?
8. Какие операторы языка C# используются для создания цепочки методов для многоадресных делегатов?
9. Каким образом осуществляется удаление цепочки методов для многоадресных делегатов?
10. Какие операторы языка C# используются для удаления цепочки методов для многоадресных делегатов?
11. Каким должен быть тип возвращаемого значения для многоадресных делегатов?
12. Что такое событие?
13. Как объявляются события?
14. В чем заключается механизм события?
15. Каков порядок создания пользовательского события?
16. Как используются методы класса в роли обработчика события?
17. Что такое лямбда-выражение? Анонимный метод? В чем их отличия?

Литература

1. Голощапов А.Л. Microsoft Visual Studio 2010. – СПб.:БХВ-Петербург, 2011. – 544 с.: ил.
2. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. Пер. с англ. - М.: «Русская Редакция», 2002.- 576 с.: ил.
3. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 2. Пер. с англ. - М.: «Русская Редакция», 2002.- 624 с.: ил.
4. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0. Пер. с англ. - М.: Издательский дом "Вильямс", 2011. — 1392 с.: ил.
5. Фленов М.Е. Библия C#. - СПб.: БХВ-Петербург, 2011.– 560с.: ил.
6. Шилдт Г. C# Учебный курс . – СПб.: Питер, Издательская группа BHV, 2003. – 512 с.: ил.