МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по Microsoft .NET и языку программирования C#

Учебно-методическое пособие

Часть 3. Делегаты и события

2018 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по Microsoft .NET и языку программирования C#. Учебно-методическое пособие. Часть 3. Делегаты и события. Озерск: ОТИ НИЯУ МИФИ, 2018. — 19 с.

В пособии предлагаются практические работы по изучению платформы Microsoft .NET и языка программирования C#.

В этой части рассматриваются делегаты и события.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1. Синяков В. Е., начальник УИТ ФГУП «ПО «Маяк».
2. Зубаиров А. Ф., ст. преподаватель кафедры ПМ ОТИ НИЯУ МИФИ.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

В ходе практических работ изучаются основы использования классов в рамках методологии объектно-ориентированного программирования.

В этой части работ рассматриваются следующие темы:

- делегаты C#;
- события C#.

К практическим работам приписаны контрольные вопросы и упражнения. Контрольные вопросы могут быть заданы преподавателем в ходе защиты работы, однако преподаватель может задавать и другие вопросы, не указанные в списке.

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения. Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

Примеры программ данного сборника работ заимствованы из книги «Троэлсен Э. C# и платформа .NET. СПб.: Питер, 2005. — 796 с.: ил.».

1. Работа CS-301. Делегаты

Цели:
- изучение делегатов C#.
Задачи:
- описание делегата;
- назначение делегата;
- методы делегата;
- операции с делегатами.
Опорные документы:

1.1. Введение в делегаты

Делегат — это безопасный эквивалент указателя на функцию в Си.

Делегат является типом, производным от MulticastDelegate (многозначный делегат). Его особенность — он может указывать своими представителями на произвольные методы, которые соответствуют сигнатуре делегата, и вызывать эти методы через своего представителя. Одному представителю делегата можно назначить несколько однотипных методов.

Создадим проект консольного приложения C#.

Название проекта — SharpDelegate.

Сначала создадим класс Point, описывающий точку на плоскости, и класс Fraction, описывающий число в виде дроби двух целых чисел.

В классе Point два защищенных элемента данных x и y, конструктор по умолчанию, конструктор преобразования. В классе Fraction два защищенных элемента данных num и den, конструктор по умолчанию, конструктор преобразования.

Далее описываем делегат в пространстве имен программы (на том же уровне, что и классы, перед методом Main):

```
public delegate void dlgSet(int a, int b);
```

Этот делегат вводит новый тип dlgSet. Для использования этого делегата нужно, чтобы классы Point и Fraction определяли соответствующие сигнатуре делегата методы void SetValues(int, int).

Определим эти методы в классах Point и Fraction.

Кроме того, переопределим в этих классах метод ToString.

Для класса Point:

```
public override string ToString() {
    return "(" + x.ToString() + "," + y.ToString() + ")";
}
```

Для класса Fraction:

```
public override string ToString() {
    return "(" + num.ToString() + "/" + den.ToString() + ")";
}
```

После того, как все подготовлено, в Main создаем три объекта:

```
static void Main(string[] args) {
    Point p1 = new Point(1, 1);
    Point p2 = new Point(2, 2);
    Fraction f1 = new Fraction(1, 1);
}
```

Создаем представителя делегата d1:

```
dlgSet d1 = p1.SetValues;
```

Теперь делегат указывает на метод SetValues объекта p1. Поскольку делегат фактически является указателем на функцию, его можно использовать как функцию:

```
d1(5, 5);
Console.WriteLine("p1" + p1.ToString());
```

При этом вызывается метод SetValues объекта p1 и объект изменяется, как будто мы вызывали метод p1.SetValues. Наверное, в этом не было бы никакого особого смысла, если бы не методы MulticastDelegate, позволяющие комбинировать в делегате несколько методов.

Например, можно назначить делегату методы SetValues всех объектов сразу, используя метод Combine, перегруженный как операция "+=":

```
d1 += p2.SetValues;
d1 += f1.SetValues;
d1(6, 6);
Console.WriteLine("p1" + p1.ToString());
Console.WriteLine("p2" + p2.ToString());
Console.WriteLine("f1" + f1.ToString());
```

От делегата можно отсоединять присоединенные методы, например:

```
d1 -= p1.SetValues;
d1(7, 7);
Console.WriteLine("p1" + p1.ToString());
Console.WriteLine("p2" + p2.ToString());
Console.WriteLine("f1" + f1.ToString());
```

Создадим делегаты d2 и df:

```
dlgSet d2 = p2.SetValues;
dlgSet df = f1.SetValues;
d2(4, 4);
df(5, 5);
Console.WriteLine("p2" + p2.ToString());
Console.WriteLine("f1" + f1.ToString());
d1 -= p1.SetValues;
d1 -= f1.SetValues;
```

Мы также отсоединили от делегата d1 все методы.

А теперь нагрузим делегат d1 методами делегатов d2 и df:

```
d1 = d2 + df;
d1(9, 9);
Console.WriteLine("p2" + p2.ToString());
Console.WriteLine("f1" + f1.ToString());
```

1.2. Делегаты и операции

Создадим проект консольного приложения C#.
Название проекта — SharpPointDelegate.
Определим класс Point (точка на плоскости):

```
class Point {
    protected int x = 0;
    protected int y = 0;
    public Point() {
    }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Определим в классе программы два статических метода.
Первый метод прибавляет точку b к точке a:

```
    static void PointAdd(Point a, Point b) {
        a.x += b.x;
        a.y += b.y;
    }
```

Второй метод вычитает из точки a точку b:

```
    static void PointSub(Point a, Point b) {
        a.x -= b.x;
        a.y -= b.y;
    }
```

Определим в пространстве имен сборки следующего делегата:

```
delegate void dlgPointOp(Point a, Point b);
```

Определим в классе Point метод для выполнения операции с точкой:

```
    public void Operation(dlgPointOp d, Point b) {
        d(this, b);
    }
```

Заместим в классе Point метод ToString:

```
    public override string ToString() {
        return "(" + x.ToString() + "," + y.ToString() + ")";
    }
```

В методе Main создадим три точки:

```
        Point a = new Point(3, 3);
        Point b = new Point(2, 2);
        Point c = new Point(1, 1);
```

Создадим представителя делегата с именем add, назначим ему метод PointAdd. Создадим представителя делегата с именем sub, назначим ему метод PointSub:

```
        dlgPointOp add = PointAdd;
        dlgPointOp sub = PointSub;
```

Выполняем операции с точками, не затрагивая класс Point.

Например, при помощи делегата add мы можем сложить точки:

```
a.Operation(add, b);
Console.WriteLine("a" + a.ToString());
```

При помощи делегата sub можно вычесть одну точку из другой:

```
a.Operation(sub, c);
Console.WriteLine("a" + a.ToString());
```

Этот пример показывает, как можно получить разную функциональность метода Operation. Однако еще лучше показывает применение делегатов следующий пример.

Добавим в using строку:

```
using System.Collections;
```

Опишем класс Points, наследующий интерфейс IEnumerable:

```
class Points : IEnumerable {
    private Point[] points;
    public Points() {
        points = new Point[1];
        points[0] = new Point();
    }
    public Points(int number) {
        points = new Point[number];
        for (int i = 0; i < number; i++) {
            points[i] = new Point();
        }       }
    public IEnumerator GetEnumerator() {
        for (int i = 0; i < points.Length; i++) {
            yield return points[i];
        }
    }
}
```

Определим в классе Points индексатор:

```
    public Point this[int index] {
        get {
            return points[index];
        }
    }
```

Определим в классе Points метод для выполнения операций:

```
    public void Operation(dlgPointOp d, Point b) {
        foreach (Point a in points) {
            d(a, b);
        }
    }
```

В методе Main создаем представителя Points:

```
Points po = new Points(3);
```

Выполняем начальную инициализацию элементов делегатом add:

```
po.Operation(add, c);
```

После этого просматриваем объекты:

```
foreach (Point p in po) {
    Console.WriteLine("p" + p.ToString());
}
```

Определим еще одного делегата следующего вида:

```
delegate void dlgPointOp1(Point a);
```

Он выполняет операцию над точкой.

Определим в классе Points другой метод для выполнения операций с точками:

```
public void Operation1(dlgPointOp1 d) {
    foreach (Point a in points) {
        d(a);
    }
}
```

Определим в классе программы операцию с точкой:

```
static void PointIncrement(Point a) {
    a.x += 1;
    a.y -= 1;
}
```

Создаем представителя делегата dlgPointOp1 в Main:

```
dlgPointOp1 inc = PointIncrement;
```

Применяем делегата по назначению:

```
po.Operation1(inc);
```

Выводим список точек:

```
foreach (Point p in po) {
    Console.WriteLine("p" + p.ToString());
}
```

Приведенные приемы можно использовать, например, для преобразований точек на плоскости и в пространстве в компьютерной графике.

1.3. Гараж и мойка

Создадим проект консольного приложения C#, название SharpWasher.

Разработайте следующие классы: автомобиль Car, гараж Garage, мойка Washer. Гараж — это коллекция автомобилей. Мойка — независимое предприятие, которое может только мыть автомобиль методом Wash.

Делегируйте помывку всех автомобилей этому предприятию.

1.4. Контрольные вопросы и упражнения

1. Поясните назначение делегатов.
2. Опишите синтаксис объявления делегата.
3. Поясните использование делегата.
4. Опишите операции с делегатами.

2. Работа CS-302. События

Цели:
- изучение событий C#.
Задачи:
- описание событий;
- генерация событий;
- обработка событий;
- стандартные события.

Опорные документы:

2.1. Введение в события

События — это уведомления об изменениях, происходящих в объектах. При наступлении события из объекта, в котором сгенерировано событие, вызывается метод объекта, который подписался получать событие, этот метод называют обработчиком события. На событие одного объекта могут подписаться одновременно несколько других объектов. Тогда обработчики события вызываются последовательно один за другим.

Модель событий C# основана на делегатах. Делегаты в этом случае обозначают методы, обрабатывающие события.

Создадим проект консольного приложения C#.

Название проекта — SharpEvents.

Опишем класс Point:

```
class Point {
    public int x = 0, y = 0;
    public Point() {
    }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void SetValues(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

В Main создадим представителя класса:

```
class Program {
    static void Main(string[] args) {
        Point a = new Point(1, 1);
    }
}
```

2.2. Событие без параметров

Чтобы генерировать событие, нужен делегат.

1. Объявляем делегата перед классом программы:

```
// делегат события Change1
delegate void dlgChange1();
```

2. Теперь в классе Point объявляем событие Change1.
Событие должно иметь тип делегата dlgChange1:

```
class Point {
    // объявление события Change1
    public event dlgChange1 Change1;
    . . .
}
```

3. В методе SetValues генерируем событие Change1. При этом нужно убедиться, что обработчик присоединен к делегату и не равен нулю:

```
public void SetValues(int x, int y) {
    this.x = x;
    this.y = y;
    if (Change1 != null) {
        Change1();
    }
}
```

4. Осталось найти метод, который обработает событие. Метод должен удовлетворять сигнатуре dlgChange1. Следующий метод подходит для этой цели. Он выводит в консоль «Changed1»:

```
class Program {
    static void Handler1() {
        Console.WriteLine("Change1");
    }
    . . .
}
```

5. Теперь нужно присоединить обработчик события Handler1 к событию. Создаем представителя делегата dlgChange1, присоединяем его к событию, после чего изменяем координаты точки:

```
static void Main(string[] args) {
    Point a = new Point(1, 1);
    a.Change1 += new dlgChange1(Handler1);
    a.SetValues(2, 2);
}
```

Запускаем программу и наблюдаем вывод в консоль «Change1».
Обработчик события можно отсоединить:

```
Point a = new Point(1, 1);
a.Change1 += new dlgChange1(Handler1);
a.SetValues(2, 2);
a.Change1 -= new dlgChange1(Handler1);
Console.WriteLine("-- Remove Handler1 --");
a.SetValues(1, 1);
```

Запускаем программу и наблюдаем вывод в консоль.
После метки «Remove Handler1» событие не возникает.

2.3. Подробности событий

Найдем и откроем приложение ILdasm.exe. С его помощью найдем и откроем файл сборки SharpEvents.exe, он находится в каталоге bin\Debug.

Посмотрим, как описано событие Change1:

```
Change1 : private class SharpEvents.dlgChange1 Change1
```

Это поле класса, которому назначен закрытый статический класс, цель которого — привязывать событие к соответствующему делегату.

Событие на самом деле — это набор из двух скрытых методов, определенных как public: метод add_Change1 и метод remove_Change1. Первый метод добавляет обработчик события, второй — удаляет. Если посмотреть код первого метода, то найдем метод Combine, а если посмотреть код второго метода, то найдем метод Remove класса System.Delegate. Если посмотреть код события, то обнаружим метки методов add_Change1 и remove_Change1, которые вызываются при присоединении и отсоединении обработчика. Если посмотреть код метода Main, то обнаружим, что эти методы там вызываются.

2.4. Событие с объектом события

Рассмотрим формирование события, в котором передается объект, который генерирует событие. Иногда этого бывает достаточно.

1. Описываем делегат с одним параметром типа object:

```
// делегат события Change2
delegate void dlgChange2(object sender);
```

2. Описываем в классе Point новое событие:

```
// объявление события Change2
public event dlgChange2 Change2;
```

3. Генерируем событие в методе SetValues:

```
if (Change2 != null) {
    Change2((object)this);
}
```

4. К одному событию может быть присоединено несколько обработчиков. Объявим два обработчика события типа Change2:

```
static void Handler2a(object o) {
    Console.WriteLine("Changed2a");
}
static void Handler2b(object o) {
    Console.WriteLine("Changed2b" +
        " to (" + ((Point)o).x + "," + ((Point)o).x + ")");
}
}
```

5. В Main присоединяем обработчики к событию Change2:

```
static void Main(string[] args) {
    Point a = new Point(1, 1);
    a.Change1 += new dlgChange1(Handler1);
    a.SetValues(2, 2);
    a.Change2 += new dlgChange2(Handler2a);
    a.Change2 += new dlgChange2(Handler2b);
    a.SetValues(3, 3);
}
```

Запускаем программу и наблюдаем вывод в консоль:

2.5. Событие с параметрами

Теперь рассмотрим событие, которое передает какие-либо параметры, кроме объекта источника, например, значения координат x и y.

1. Описываем третий делегат:

```
// делегат события Change3
delegate void dlgChange3(object sender, int x, int y);
```

2. В классе описываем событие:

```
// объявление события Change3
public event dlgChange3 Change3;
```

3. В методе SetValues генерируем новое событие:

```
if (Change3 != null) {
    Change3((object)this, x, y);
}
```

4. Описываем обработчик:

```
static void Handler3(object o, int x, int y) {
    Console.WriteLine("Changed3" +
        " to (" + x + "," + y + ")");
}
```

5. В Main присоединяем обработчик, изменяем точку:

```
static void Main(string[] args) {
    . . .
    a.SetValues(4, 4);
    Console.WriteLine("-- Handler3 --");
    a.Change3 += new dlgChange3(Handler3);
    a.SetValues(5, 5);
}
```

Наблюдаем вывод в консоль.

Описанная форма передачи параметров является нестандартной и не рекомендуемой.

2.6. Стандартные параметры

Для передачи параметров через события есть класс EventArgs. Его можно и нужно использовать, когда к событию прикрепляются какие-либо параметры.

1. Сначала описываем класс, который передает параметры:

```
class PointEventArgs : EventArgs {
    public int x, y;
    public PointEventArgs(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

2. Описываем делегат, передающий параметры:

```
// делегат события Change4
delegate void dlgChange4(object sender, PointEventArgs e);
```

3. В классе описываем событие Change4:

```
    // объявление события Change4
    public event dlgChange4 Change4;
```

4. В методе SetValues генерируем событие Change4:

```
        if (Change4 != null) {
            Change4((object)this, new PointEventArgs(x, y));
        }
```

5. Описываем обработчик события Change4:

```
    static void Handler4(object sender, PointEventArgs e) {
        Console.WriteLine("Changed4" +
            " to (" + e.x + "," + e.y + ")");
    }
```

6. В Main присоединяем обработчик, изменяем точку:

```
        Console.WriteLine("-- PointEventArgs --");
        a.Change4 += new dlgChange4(Handler4);
        a.SetValues(6, 6);
```

Наблюдаем вывод в консоль.

2.7. Стандартный делегат события

Для описания событий лучше всего использовать стандартный делегат события EventHandler. Описывать этот делегат не нужно, он существует. С его помощью можно передавать события с параметрами и без них.

2.7.1. Событие без параметров

Рассмотрим использование стандартного делегата, когда параметры передавать не требуется.

1. Описываем событие Change в классе Point, используя стандартный делегат события EventHandler:

```
    // объявление события Change
    public event EventHandler Change;
```

2. Генерируем событие в методе SetValues. Мы генерируем событие Change дважды, чтобы показать разные способы формирования пустых параметров:

```
if (Change != null) {
    Change((object)this, EventArgs.Empty);
    Change((object)this, new EventArgs());
}
```

3. Описываем обработчик стандартного события:

```
static void Handler(object sender, EventArgs e) {
    Console.WriteLine("Changed - empty EventArgs");
}
```

4. В Main присоединяем обработчик, изменяем точку:

```
Console.WriteLine("-- Standard Delegate --");
a.Change += new EventHandler(Handler);
a.SetValues(7, 7);
```

Наблюдаем вывод в консоль.

2.7.2. Событие с параметрами

1. Сначала описываем класс, передающий параметры.
Подходящий класс уже есть, это класс PointEventArgs.
2. Описываем событие в классе Point:

```
// объявление события Change5
public event EventHandler<PointEventArgs> Change5;
```

3. Генерируем событие в методе SetValues:

```
if (Change5 != null) {
    Change5((object)this, new PointEventArgs(x, y));
}
```

4. Описываем обработчик с параметрами типа PointEventArgs.
Подходящий обработчик уже есть, это Handler4.
5. В Main присоединяем обработчик, изменяем точку:

```
Console.WriteLine("-- Standard Params --");
a.Change5 += new EventHandler<PointEventArgs>(Handler4);
a.SetValues(8, 8);
```

Наблюдаем вывод в консоль.

2.8. Классы как приемники событий

Создадим консольное приложение C#, название SharpEventsArray.
Обычно в качестве приемников событий используются классы.
В данной работе демонстрируется этот подход.
Добавим инструкцию using:

```
using System.Collections;
```

Создаем коллекцию на основе ArrayList:

```
class ArrayWithEvents : ArrayList {
}
```

Описываем в классе событие, которое удовлетворяет стандартному соглашению:

```
// событие со стандартным делегатом
public event EventHandler Changed;
```

Далее описываем метод для генерации события Changed:

```
// метод для генерация стандартного события
protected virtual void OnChanged(EventArgs e) {
    if (Changed != null) {
        Changed(this, e);
    }
}
```

Замещаем основные методы ArrayList:

```
// добавление элемента
public override int Add(object value) {
    int i = base.Add(value);
    OnChanged(EventArgs.Empty);
    return i;
}

// очистка объекта
public override void Clear() {
    base.Clear();
    OnChanged(EventArgs.Empty);
}

// индексатор
public override object this[int index] {
    set {
        base[index] = value;
        OnChanged(EventArgs.Empty);
    }
}
```

Далее создаем класс, который будет прослушивать события списка:

```
// класс для прослушивания событий ArrayWithEvents
class ListEventListener {
}
```

Закрытый элемент этого класса хранит объект для прослушивания:

```
// хранитель объекта для прослушивания
private ArrayWithEvents list;
```

Следующий метод обрабатывает событие:

```
// метод для обработки события Changed
private void ListChanged(object sender, EventArgs e) {
    Console.WriteLine("Event fired.");
}
```

Конструктор класса инициализирует объект для прослушивания и подключается к событию:

```
// конструктор
public ListEventListener(ArrayWithEvents list) {
    this.list = list;
    this.list.Changed += new EventHandler(ListChanged);
}
```

Специальный метод Detach отключает прослушивание:

```
    // отключает прослушивание и ссылку на объект прослушивания
    public void Detach() {
        this.list.Changed -= new EventHandler(ListChanged);
        this.list = null;
    }
```

В Main создаем объект list класса ArrayWithEvents, затем создаем объект listener класса ListEventListener, передавая ему параметр list, добавляем в список list произвольные объекты, очищаем список и, наконец, вызываем метод Detach объекта listener:

```
    static void Main(string[] args) {
        ArrayWithEvents list = new ArrayWithEvents();
        ListEventListener listener = new ListEventListener(list);
        list.Add("abc");
        list.Add(123);
        list.Clear();
        listener.Detach();
    }
```

Запускаем программу, наблюдаем вывод в консоль:

```
Event fired.
Event fired.
Event fired.
```

2.9. Наследуем интерфейс с событием

Создадим проект консольного приложения C#.
Название проекта — SharpEventsInterface.
Создадим интерфейс, описывающий событие:

```
// интерфейс, описывающий событие Changed
interface IEventInterface {
    event EventHandler Changed;
    void FireEvent(EventArgs e);
}
```

Описываем класс, наследующий этот интерфейс:

```
class TestEventInterface : IEventInterface {
    // наследуемое событие
    public event EventHandler Changed;
    // наследуемый метод
    public void FireEvent(EventArgs e) {
        EventHandler t = Changed;
        if (t != null) {
            t((object)this, e);
        }
    }
    public void ChangeData() {
        // что-то делаем с классом
        FireEvent(EventArgs.Empty);
    }
}
```

Описываем метод, который обрабатывает события:

```
static void Handler(object sender, EventArgs e) {
    Console.WriteLine("Event fired.");
}
```

В Main создаем представителя класса TestEventInterface:

```
IEventInterface IE = new TestEventInterface();
```

Присоединяем обработчик события к событию Changed:

```
IE.Changed += new EventHandler(Handler);
```

Наконец, возбуждаем события двумя разными способами:

```
IE.FireEvent(EventArgs.Empty);
((TestEventInterface)IE).ChangeData();
```

Запускаем программу, наблюдаем вывод в консоль:

```
Event fired.
Event fired.
```

2.10. Наследуем два интерфейса с событиями

Создадим проект консольного приложения C#.

Название проекта — SharpEventsInterface2.

Пусть есть два следующих интерфейса, описывающих одинаковые по названию события, и класс, наследующий эти интерфейсы:

```
interface IEvents1 {
    event EventHandler SampleEvent;
}
interface IEvents2 {
    event SampleEventHandler SampleEvent;
}
class TestTwoEvents : IEvents1, IEvents2 {
}
```

Описываем делегат SampleEventHandler:

```
delegate void SampleEventHandler(string s);
```

Одно из событий класс реализует обычным образом, например:

```
// номальная реализация события IEvents1.SampleEvent
public event EventHandler SampleEvent;
```

Другое событие класс реализует при помощи внутреннего посредника и явного описания аксессоров события add и remove:

```
// посредник для события IEvents2.SampleEvent
private SampleEventHandler SampleEventStorage;
// явное описание элементов события
event SampleEventHandler IEvents2.SampleEvent {
    add {
        SampleEventStorage += value;
    }
    remove {
        SampleEventStorage -= value;
    }
}
```

Демонстрационный метод генерирует оба события:

```
public void FireEvents() {
    if (SampleEvent != null) {
        SampleEvent((object)this, EventArgs.Empty);
    }
    if (SampleEventStorage != null) {
        SampleEventStorage("SampleEventStorage fired.");
    }
}
```

Для приема стандартного события создаем обработчик Handler:

```
static void Handler(object sender, EventArgs e) {
    Console.WriteLine("IEvents1.SampleEvent fired.");
}
```

Для приема события типа SampleEventHandler создаем еще один обработчик Handler2:

```
static void Handler2(string s) {
    Console.WriteLine(s);
}
```

Переходим в метод Main.

Создаем представителя TestTwoEvents:

```
TestTwoEvents test = new TestTwoEvents();
```

Далее создаем делегата для первого события, присоединяем его к событию и генерируем событие:

```
EventHandler temp = new EventHandler(Handler);
((IEvents1)test).SampleEvent += temp;
test.FireEvents();
```

Далее создаем делегата для второго события, присоединяем к событию и генерируем событие:

```
SampleEventHandler t2 = new SampleEventHandler(Handler2);
((IEvents2)test).SampleEvent += t2;
test.FireEvents();
```

Запускаем программу, наблюдаем вывод в консоль:

```
IEvents1.SampleEvent fired.
IEvents1.SampleEvent fired.
SampleEventStorage fired.
```

2.11. Контрольные вопросы и упражнения

1. В чем роль событий?
2. Как события соотносятся с делегатами?
3. Как события устроены внутренне?
4. Какова стандартная форма события?
5. Как описывается событие с аксессорами?