Лабораторная работа

10. Коллекции и многопоточность

Варианты заданий

Выполнить задания 1-3 по предложенным вариантам:

	№ Задания			
	1	2	3	
Вариант 1	1.I	2.I	3.I	
Вариант 2	1.II	2.II	3.II	
Вариант 3	1.III	2.III	3.III	
Вариант 4	1.I	2.II	3.II	
Вариант 5	1.II	2.I	3.III	
Вариант 6	1.III	2.II	3.I	
Вариант 7	1.I	2.III	3.I	
Вариант 8	1.I	2.I	3.II	
Вариант 9	1.II	2.III	3.I	
Вариант 10	1.III	2.I	3.III	

Задание 1.

I

Разработать приложение для демонстрации работы с коллекцией ArrayList из строк: добавить в список различные строки, добавить несколько строк в начало списка, по указанному индексу. Изменить размер списка.

Реализовать возможность просмотра всех элементов в списке, с указанием числа элементов и емкости списка.

Разработать делегат для работы следующих методов:

-удалить лишние пробелы;

-удалить лишние знаки препинания;

-добавить пропущенные пробелы, после знаков препинания;

-для строк, содержащих числа, привести их к формату 0,##

II Разработать приложение для демонстрации работы с коллекцией ArrayList из строк: добавить в список различные строки, добавить несколько строк в начало списка, по указанному индексу. Изменить размер списка.

Реализовать возможность просмотра всех элементов в списке, с указанием числа элементов и емкости списка.

Разработать делегат для работы следующих методов:

-заменить пробелы на знак _

-заменить / на \

-заменить / на //

-перевод букв в верхний регистр

III Разработать приложение для демонстрации работы динамического массива из вещественных чисел (коллекция ArrayList): добавить в список различные элементы, добавить несколько элементов в начало списка, по указанному индексу. Изменить размер списка.

Реализовать возможность просмотра всех элементов в списке, с указанием числа элементов и емкости списка.

Делегат для ряда

-находит максимум;

-находит минимум;

-находит среднее;

-находит сумму.

Задание 2.

Разработать многопоточное приложение. Для генерации событий использовать потокобезопасные коллекции (BlockingCollection<T>). Приложение должно генерировать события на протяжении 15 секунд (полный цикл моделирования).

I Создать проект «Лабораторные работы», имитирующий сдачу лабораторных работ студентами. В начале работы программы предполагается, что студенты получают лабораторные задания. Каждый студент делает работу некоторое время, после чего сдает работу преподавателю. Сдача работы занимает некоторое время (преподаватель не может проверять 2 работы одновременно), после чего студент либо получает отметку о сдаче, либо дорабатывает лабораторную.

Протестировать программу для 20 студентов и 2 преподавателей.

События: запрос студентом на сдачу работы, принятие работы на проверку, сдача работы, возврат на доработку.

II Создать проект «СОЧИ-2014», имитирующий параллельные спортивные соревнования. Соревнования проходят одновременно по разным видам спорта на разных площадках.

Протестировать программу для 10 видов спорта (лыжный спуск, биатлон, бобслей, фигурное катание, керлинг и т.д.) на 5 площадках. В один момент времени на каждой площадке могут проводиться только одни соревнования.

События: тренировка, проведение этапа соревнования (начало соревнования, окончание соревнования).

III Создать проект «Аукцион», имитирующий проведение аукциона на нескольких площадках одновременно.

Протестировать программу для 10 участников и 5 лотов на каждой из трех площадок.
События: начало торгов на площадке, выдвижение лота, установка ставки, продажа лота.

Задание 3.
Разработать проект для демонстрации лямбда-выражения, генерирующего:

I Числа Фибоначчи (последующее число равно сумме двух предыдущих чисел).
II Совершенные числа (число равно сумме всех своих собственных делителей).
III Простые числа.

Методические указания

Оглавление

10.1. Делегаты

Делегат — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как любой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка (в том числе в многопоточных приложениях).

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

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

Допускаются спецификаторы `new`, `public`, `protected`, `internal` и `private`. *Тип* описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательными *параметрами* делегата являются параметры этих методов. Делегат может хранить ссылки на несколько методов и вызывать их поочередно; естественно, что сигнатуры всех методов должны совпадать.

Пример описания делегата:

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

Здесь описан тип делегата, который может хранить ссылки на методы, возвращающие `void` и принимающие один параметр целого типа. Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

Чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы.

Делегаты применяются в основном для следующих целей:

- получения возможности определять вызываемый метод не при компиляции, а во время выполнения программы;
- обеспечения связи между объектами по типу "источник — наблюдатель";
- создания универсальных методов, в которые можно передавать другие методы;
- поддержки механизма обратных вызовов.

Рассмотрим сначала пример реализации первой из этих целей.

```
using System;
namespace SimpleDelegate
{
    // Этот делегат может указывать на любой метод,
    // принимающий два целых и возвращающий целое.
    public delegate int BinaryOp(int x, int y);
    // Этот класс содержит методы, на которые будет указывать BinaryOp.
    public class SimpleMath
    {
        public static int Add(int x, int y)
        { return x + y; }
        public static int Subtract(int x, int y)
        { return x - y; }
```

```
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Simple Delegate Example *****\n");
            // Создать объект делегата BinaryOp, "указывающий" на
SimpleMath.Add().
            BinaryOp b1 = new BinaryOp(SimpleMath.Add);
            // Вызвать метод Add() непрямо с использованием объекта делегата.
            Console.WriteLine("10 + 10 is {0}", b1(10, 10));
            // Создать объект делегата BinaryOp, "указывающий" на
SimpleMath.Substract().
            BinaryOp b2 = new BinaryOp(SimpleMath.Substract);
            // Вызвать метод Substract() непрямо с использованием объекта
делегата.
            Console.WriteLine("10 - 10 is {0}", b2(10, 10));

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

Результат работы программы:

```
***** Simple Delegate Example *****

10 + 10 is 20
10 - 10 is 0
```

Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит *ссылки на несколько методов*, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

Добавление метода в список выполняется либо с помощью метода `Combine`, унаследованного от класса `System.Delegate`, либо, что удобнее, с помощью перегруженной операции сложения. Вот как выглядит измененный метод `Main` из предыдущего листинга, в котором одним вызовом делегата выполняется преобразование исходной строки сразу двумя методами.

```
static void Main()
{
        BinaryOp b1 = new BinaryOp(SimpleMath.Add);
        b1 += new BinaryOp(SimpleMath.Substract);              // добавление
метода в делегат
            Console.WriteLine("10 - 10 is {0}", b1(10, 10));
}
```

При вызове последовательности методов с помощью делегата необходимо учитывать следующее:

- сигнатура методов должна в точности соответствовать делегату;
- методы могут быть как статическими, так и обычными методами класса;
- каждому методу в списке передается один и тот же набор параметров;
- если параметр передается по ссылке, изменения параметра в одном методе отразятся на его значении при вызове следующего метода;

- если параметр передается с ключевым словом `out` или метод возвращает значение, результатом выполнения делегата является значение, сформированное последним из методов списка (в связи с этим рекомендуется формировать списки только из делегатов, имеющих возвращаемое значение типа `void`);
- если в процессе работы метода возникло исключение, не обработанное в том же методе, последующие методы в списке не выполняются, а происходит поиск обработчиков в объемлющих делегат блоках;
- попытка вызвать делегат, в списке которого нет ни одного метода, вызывает генерацию исключения `System.NullReferenceException`.

10.1.1. Паттерн "наблюдатель"

Рассмотрим применение делегатов для обеспечения связи между объектами по типу "источник — наблюдатель". В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов. При этом желательно избежать жесткой связанности классов, так как это часто негативно сказывается на возможности многократного использования кода.

Для обеспечения связи между объектами во время выполнения программы применяется следующая стратегия. Объект, называемый *источником (sender)*, при изменении своего состояния, которое может представлять интерес для других объектов, посылает им уведомления. Эти объекты называются *наблюдателями (receiver)*. Получив уведомление, наблюдатель опрашивает источник, чтобы синхронизировать с ним свое состояние. Примером такой стратегии может служить связь электронной таблицы с созданными на ее основе диаграммами.

Программисты часто используют одну и ту же схему организации и взаимодействия объектов в разных контекстах. За такими схемами закрепилось название *паттерны*, или *шаблоны проектирования*. Описанная стратегия известна под названием *паттерн "наблюдатель"*.

Наблюдатель (observer) определяет между объектами зависимость типа "один ко многим", так что при изменении состоянии одного объекта все зависящие от него объекты получают извещение и автоматически обновляются. Рассмотрим пример, в котором демонстрируется схема оповещения источником трех наблюдателей. Гипотетическое изменение состояния объекта моделируется сообщением "Упс!". Один из методов в демонстрационных целях сделан статическим.

```
using System;
namespace ConsoleApplication1
{
    public delegate void Del(object o);              // объявление делегата

    class Subj                                                  // класс-источник
    {
        Del dels;                              // объявление экземпляра делегата

        public void Register(Del d)                  // регистрация делегата
        {
            dels += d;
        }

        public void OOPS()                                  // что-то произошло
```

```
        {
            Console.WriteLine("Упс!");
            if (dels != null) dels(this);      // оповещение наблюдателей
        }
    }

    class ObsA                                          // класс-наблюдатель
    {
        public void Do(object o)            // реакция на событие источника
        {
            Console.WriteLine("Бывает!");
        }
    }

    class ObsB                                          // класс-наблюдатель
    {
        public static void See(object o)    // реакция на событие источника
        {
            Console.WriteLine("Видишь?");
        }
    }

    class Class1
    {
        static void Main()
        {
            Subj s = new Subj();                //  объект класса-источника

            ObsA o1 = new ObsA();                //                  объекты
            ObsA o2 = new ObsA();                //       класса-наблюдателя

            s.Register(new Del(o1.Do));       //      регистрация методов
            s.Register(new Del(o2.Do));       // наблюдателей в источнике
            s.Register(new Del(ObsB.See));    //  ( экземпляры делегата )

            s.OOPS();                                //    инициирование события
        }
    }
}
```

В источнике объявляется экземпляр делегата, в этот экземпляр заносятся методы тех объектов, которые хотят получать уведомление об изменении состояния источника. Этот процесс называется *регистрацией делегатов*. При регистрации имя метода добавляется к списку. При наступлении "часа X" все зарегистрированные методы поочередно вызываются через делегат.

Результат работы программы:

```
Упс!
Бывает!
Бывает!
Видишь?
```

Для обеспечения обратной связи между наблюдателем и источником делегат объявлен с параметром типа `object`, через который в вызываемый метод передается ссылка на вызывающий объект. Следовательно, в вызываемом методе можно получать информацию о состоянии вызывающего объекта и посылать ему сообщения.

Связь "источник — наблюдатель" устанавливается во время выполнения программы для каждого объекта по отдельности. Если наблюдатель больше не хочет получать уведомления от источника, можно удалить соответствующий метод из списка делегата с помощью метода `Remove` или перегруженной операции вычитания, например:

```
public void UnRegister( Del d )                    // удаление делегата
{
    dels -= d;
}
```

Примечание

Делегат, как и строка `string`, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

10.1.2. Передача делегатов в методы

Поскольку делегат является классом, его можно передавать в методы в качестве параметра. Таким образом обеспечивается *функциональная параметризация*: в метод можно передавать не только различные данные, но и различные функции их обработки. Функциональная параметризация применяется для создания универсальных методов и обеспечения возможности обратного вызова.

В качестве простейшего примера *универсального метода* можно привести метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции.

```
using System;
namespace ConsoleApplication1
{
    public delegate double Fun(double x);           // объявление делегата

    class Class1
    {
        public static void Table(Fun F, double x, double b)
        {
            Console.WriteLine(" ----- X ----- Y -----");
            while (x <= b)
            {
                Console.WriteLine("| {0,8:0.000} | {1,8:0.000} |", x, F(x));
                x += 1;
            }
            Console.WriteLine(" ----------------------");
        }

        public static double Simple(double x)
        {
            return 1;
        }

        static void Main()
        {
            Console.WriteLine(" Таблица функции Sin ");
            Table(new Fun(Math.Sin), -2, 2);

            Console.WriteLine(" Таблица функции Simple ");
```

```
            Table(new Fun(Simple), 0, 3);
        }
    }
}
```

Результат работы программы:

```
Таблица функции Sin
 ----- X ----- Y -----
|   -2,000 |   -0,909 |
|   -1,000 |   -0,841 |
|    0,000 |    0,000 |
|    1,000 |    0,841 |
|    2,000 |    0,909 |
 ---------------------
 Таблица функции Simple
 ----- X ----- Y -----
|    0,000 |    1,000 |
|    1,000 |    1,000 |
|    2,000 |    1,000 |
|    3,000 |    1,000 |
 ---------------------
```

Обратный вызов (callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра. Рассмотрим рисунок 10.1. Допустим, в библиотеке описана функция `A`, параметром которой является имя другой функции. В вызывающем коде описывается функция с требуемой сигнатурой (`B`) и передается в функцию `A`. Выполнение функции `A` приводит к вызову `B`, то есть управление передается из библиотечной функции обратно в вызывающий код.

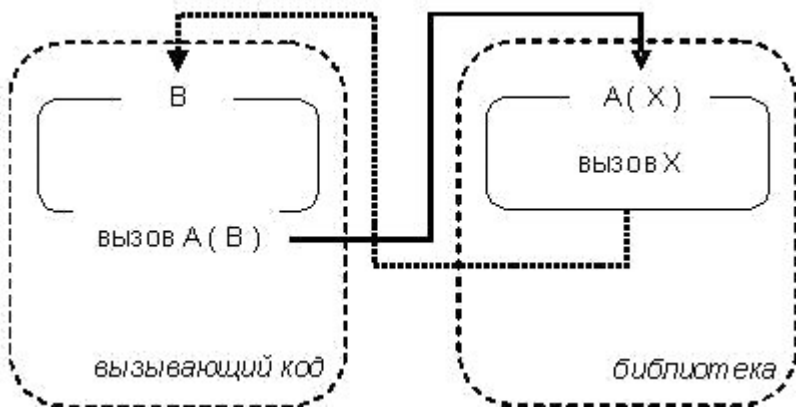

Рис. 10.1. Механизм обратного вызова

Механизм обратного вызова широко используется в программировании. Например, он реализуется во многих стандартных функциях Windows.

Начиная с версии 2.0 языка C#, можно применять упрощенный синтаксис для делегатов. Первое упрощение заключается в том, что в большинстве случаев явным образом создавать экземпляр делегата не требуется, поскольку он создается автоматически по контексту. Второе упрощение заключается в возможности создания так называемых *анонимных методов* — фрагментов кода, описываемых непосредственно в том месте, где используется делегат:

```
static void Main()
        {
            Console.WriteLine( " Таблица функции Sin " );
            Table( Math.Sin, -2, 2 );                          // упрощение 1

            Console.WriteLine( " Таблица функции Simple " );
            Table( delegate (double x ){ return 1; }, 0, 3 ); // упрощение 2
        }
    }
}
```

В первом случае экземпляр делегата, соответствующего функции `Sin`, создается автоматически. Чтобы это могло произойти, список параметров и тип возвращаемого значения функции должны быть совместимы с делегатом. Во втором случае не требуется оформлять простой фрагмент кода в виде отдельной функции `Simple`, как это было сделано в предыдущем листинге, — код функции оформляется как анонимный метод и встраивается прямо в место передачи.

10.2. Многопоточные приложения

Приложение .NET состоит из одного или нескольких *процессов*. Процессу принадлежат выделенная для него область оперативной памяти и ресурсы. Каждый процесс может состоять из нескольких *доменов* (частей) приложения, ресурсы которых изолированы друг от друга. В рамках домена может быть запущено несколько потоков выполнения. *Поток* (thread) представляет собой часть исполняемого кода программы. Иногда термин "thread" переводится буквально — "нить", чтобы отличить его от потоков ввода-вывода. Поэтому в литературе можно встретить и термин "многонитевые приложения".

В каждом процессе есть *первичный поток*, исполняющий роль точки входа в приложение. Для консольных приложений это метод `Main`.

Многопоточные приложения создают как для многопроцессорных, так и для однопроцессорных систем. Основной целью при этом являются повышение общей производительности и сокращение времени реакции приложения. Управление потоками осуществляет операционная система. Каждый поток получает некоторое количество квантов времени, по истечении которого управление передается другому потоку. Это создает у пользователя однопроцессорной машины впечатление одновременной работы нескольких потоков и позволяет, к примеру, выполнять ввод текста одновременно с длительной операцией по передаче данных.

Недостатки многопоточности:

- большое количество потоков ведет к увеличению накладных расходов, связанных с переключением потоков, что снижает общую производительность;
- возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков (например, если один поток начинает изменение общих данных, а отведенное ему время истекает, доступ к этим же данным может получить другой поток, который, изменяя данные, необратимо их повреждает).

10.2.1. События

Событие — это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, являющихся наблюдателями события, активизируются методы-обработчики этого события. Обработчики должны быть зарегистрированы в объекте-источнике события. Таким образом, механизм событий формализует на языковом уровне паттерн "наблюдатель", который рассматривался в предыдущем разделе.

Механизм событий можно также описать с помощью модели "публикация — подписка": один класс, являющийся *отправителем* (sender) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся *получателями* (receivers) сообщения, подписываются на получение этих событий.

События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:

- описание делегата, задающего сигнатуру обработчиков событий;
- описание события;
- описание метода (методов), инициирующих событие.

Синтаксис события похож на синтаксис делегата:

```
[ атрибуты ] [ спецификаторы ] event тип имя_события
```

Для событий применяются *спецификаторы* `new`, `public`, `protected`, `internal`, `private`, `static`, `virtual`, `sealed`, `override`, `abstract` и `extern`. Например, так же как и методы, событие может быть статическим (`static`), тогда оно связано с классом в целом, или обычным — в этом случае оно связано с экземпляром класса. *Тип* события — это тип делегата, на котором основано событие.

Пример описания делегата и соответствующего ему события:

```
public delegate void Del( object o );                    // объявление делегата
class A
{
    public event Del Oops;                                // объявление события
    ...
}
```

Обработка событий выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.

Как видите, это в точности тот же самый механизм, который рассматривался в предыдущем разделе. Единственное отличие состоит в том, что при использовании событий не требуется описывать метод, регистрирующий обработчики, поскольку события поддерживают операции `+=` и `-=`, добавляющие обработчик в список и удаляющие его из списка.

Переработаем пример паттерна «Наблюдатель» с использованием событий.

Рисунок 10.2 поясняет организацию обработки событий.

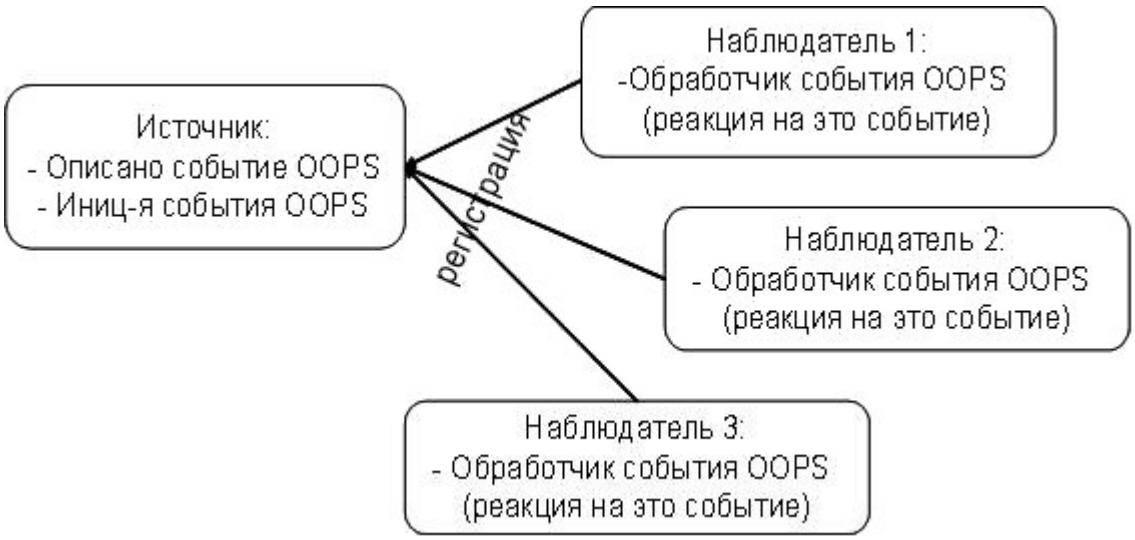

Рис. 10.2. Выполнение программы с двумя нулевыми коэффициентами

```
using System;
namespace ConsoleApplication1
{
    public delegate void Del();                          // объявление делегата

    class Subj                                               // класс-источник
    {
        public event Del Oops;                           // объявление события

        public void CryOops()                    // метод, инициирующий событие
        {
            Console.WriteLine( "ОЙ!" );
            if ( Oops != null ) Oops();
        }
    }

    class ObsA                                               // класс-наблюдатель
    {
        public void Do();                        // реакция на событие источника
        {
            Console.WriteLine( "Бедняжка!" );
        }
    }

    class ObsB                                               // класс-наблюдатель
    {
        public static void See()                 // реакция на событие источника
        {
            Console.WriteLine( "Да ну, ерунда!" );
        }
    }

    class Class1
    {
        static void Main()
        {
            Subj s  = new Subj();                //    объект класса-источника
```

```
            ObsA o1 = new ObsA();                   //                  объекты
            ObsA o2 = new ObsA();                   //          класса-наблюдателя

            s.Oops += new Del( o1.Do );             //                  добавление
            s.Oops += new Del( o2.Do );             //                обработчиков
            s.Oops += new Del( ObsB.See );          //                  к событию

            s.CryOops();                            //      инициирование события
        }
    }
}
```

Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции `+=` и `-=`. Тип результата этих операций — `void`, в отличие от операций сложного присваивания для арифметических типов.

Внутри класса, в котором описано событие, с ним можно обращаться, как с обычным полем, имеющим тип делегата: использовать операции отношения, присваивания и т. д. Значение события по умолчанию — `null`.

В библиотеке .NET описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

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

Если обработчикам события требуется специфическая информация о событии, то для этого создают класс, производный от стандартного класса `EventArgs`, и добавляют в него необходимую информацию. Если делегат не использует такую информацию, можно обойтись стандартным классом делегата `System.EventHandler`.

Имя обработчика события принято составлять из префикса `On` и имени события. Оформим код в соответствии со стандартными соглашениями .NET.

```
using System;
namespace ConsoleApplication1
{
    class Subj
    {
        public event EventHandler Oops;

        public void CryOops()
        {
            Console.WriteLine("ОЙ!");
            if (Oops != null) Oops(this, null);
        }
    }

    class ObsA
    {
        public void OnOops(object sender, EventArgs e)
        {
```

```
            Console.WriteLine("Бедняжка!");
        }
    }

    class ObsB
    {
        public static void OnOops(object sender, EventArgs e)
        {
            Console.WriteLine("Да ну, ерунда!");
        }
    }

    class Class1
    {
        static void Main()
        {
            Subj s = new Subj();

            ObsA o1 = new ObsA();
            ObsA o2 = new ObsA();

            s.Oops += new EventHandler(o1.OnOops);
            s.Oops += new EventHandler(o2.OnOops);
            s.Oops += new EventHandler(ObsB.OnOops);

            s.CryOops();
        }
    }
}
```

10.2.2. Специфика поведения объектов

Каждый объект является экземпляром некоторого класса. Класс задает свойства и поведение своих экземпляров. Методы класса определяют поведение объектов, свойства - их состояние. Все объекты обладают одними и теми же методами. Можно полагать, что методы задают врожденное поведение объектов. У всех объектов один и тот же набор свойств, но значения свойств объектов различны, так что объекты одного класса находятся в разных состояниях. Объекты класса "Человек" могут иметь разные значения свойства "Рост": один - высокий, другой - низкий, третий - среднего роста.

Хотя реализация методов у всех объектов одна, это не значит, что результат выполнения одного и того же метода, вызванного разыми объектами, будет один и тот же, поскольку выполнение метода зависит от свойств. Так что методы "Пробежать километр" и "Решить задачу" будут давать разные результаты для разных объектов класса "Человек".

Интерфейсные и многие другие программные объекты обладают стандартным набором предопределенных событий. Давайте разберемся, как для таких классов создаются и обрабатываются события. Класс, решивший иметь события, должен уметь выполнить, по крайней мере, три вещи:

- объявить событие в классе;
- зажечь в нужный момент событие, передав обработчику необходимые для обработки аргументы. (Под зажиганием или включением события понимается некоторый механизм, позволяющий объекту уведомить клиентов класса, что у него произошло событие);

- проанализировать, при необходимости, результаты события, используя значения выходных аргументов события, возвращенные обработчиком.

Заметьте, сам класс не может индивидуализировать поведение своих объектов, поскольку объекты создаются в клиентских классах, о которых класс ничего не знает. Именно клиенты класса создают объекты и индивидуализируют их поведение, создавая обработчики событий для объектов класса.

Давайте построим простую модель города, в котором возникает событие "пожар". В городе есть дома, и с некоторой вероятностью в каждом доме каждый день возможен пожар. Добавим поля к нашему классу `TownWithEvents`:

```
string townName;
int buildings;
int days;
double fireProbability;
Random random=new Random();
```

Поля, соответственно, задают название города, число домов в нем, число дней, в течение которых наблюдается жизнь города, вероятность возникновения пожара в доме. Переменная `random` класса `Random` позволит нам моделировать случайные события.

Приведу конструктор класса и метод-свойство, обеспечивающее доступ к одному из закрытых полей класса:

```
/// <summary>
  /// Конструктор, инициализирующий поля класса
  /// </summary>
  /// <param name="name">название города</param>
  /// <param name="buildings">число домов</param>
  /// <param name="days">число дней жизни города</param>
  /// <param name="fireProbability">
  /// вероятность пожара в доме в текущий день</param>
      public TownWithEvents(string name, int buildings,
      int days, double fireProbability)
      {
      townName = name;
      this.buildings = buildings;
      this.days = days;
      this.fireProbability = fireProbability;
      }
/// <summary>
  /// Доступ к полю townName
  /// </summary>
  public string TownName
  { get { return townName; } }
```

Ну а теперь рассмотрим наиболее интересный метод, моделирующий жизнь города. В этом методе день за днем для каждого строения моделируется случайное событие возникновения пожара. Если такое случается, то вызывается метод `OnFire` - событие зажигается, классы `receiver` получают уведомление о событии и обрабатывают его предусмотренным в них методом-обработчиком события.

```
/// <summary>
/// Моделирование жизни города
/// </summary>
public void TownLife()
```

```
  {
        const string OK =
            "В городе все спокойно! Пожаров не было.";
        bool wasFire = false;
     for(int day = 1; day < days; day++)
            for(int building = 1; building < buildings; building++)
                if (random.NextDouble() < fireProbability)
            {
               OnFire(day,building);
               wasFire = true;
            }
        if (!wasFire)
            Console.WriteLine(OK);
  }
```

Мы разобрались в том, как создаются события в классе `sender`. Давайте теперь рассмотрим классы, принимающие событие.

Объекты класса `sender` создают события и уведомляют о них объекты, возможно, разных классов, названных нами классами `receiver`, или клиентами. Давайте разберемся, как должны быть устроены **классы receiver**, чтобы вся эта схема заработала.

Понятно, что класс `receiver` должен:

- иметь обработчик события - процедуру, согласованную по сигнатуре с функциональным типом делегата, который задает событие;
- иметь ссылку на объект, создающий событие, чтобы получить доступ к событию - `event`-объекту;
- уметь присоединить обработчик события к `event`-объекту. Присоединение можно реализовать по-разному. Это можно делать непосредственно в конструкторе класса, которому передается в качестве аргумента объект, посылающий сообщения. В этом случае уже при создании объект, получающий сообщение, изначально готов принимать и обрабатывать сообщения о событиях. Такое решение хорошо, когда класс настроен на прием сообщений от одного объекта. В других ситуациях удобнее иметь специальный метод, которому передается объект класса `sender`. В этом методе и происходит присоединение обработчика события к событию того объекта, который передан методу.

Рассмотрим пример, демонстрирующий возможное решение проблем:

```
/// <summary>
/// Пожарная служба - класс Reciver
/// Принимает и обрабатывает событие FireEvent
/// пожар в городе, клиентом которого является класс.
/// </summary>
public class FireMen
{
    /// <summary>
    /// Город, клиентом которого является класс FireMen
    /// </summary>
    private TownWithEvents MyNativeTown;
    public FireMen(TownWithEvents twe)
    {
        this.MyNativeTown = twe;
        MyNativeTown.fireEvent += new FireEvent(FireHandler);
    }
    /// <summary>
    /// Обработчик события "пожар в городе"
    /// </summary>
```

```
    /// <param name="day">день пожара</param>
    /// <param name="buildings">строение</param>
    private void FireHandler(int day, int buildings)
    {
        string message =
            "В городе {0} произошел пожар! " +
            "В день {1}, в доме № {2}" +
            "  Пожар потушен!";
        Console.WriteLine(string.Format(message, MyNativeTown.TownName,
           day, buildings));
    }
    public void GoOut()
    {
        MyNativeTown.fireEvent -= new FireEvent(FireHandler);
    }
}//FireMan
```

В классе `Fireman` есть ссылка на объект класса `TownWithEvents`, создающий события. Сам объект передается конструктору класса. В конструкторе метод класса, задающий обработку события, присоединяется к списку вызовов `event`-объекта. Возможность комбинирования делегатов в полной мере проявляется при работе с событиями. Благодаря этой возможности одно и то же событие будет обрабатываться методами, принадлежащими разным классам `receiver`.

Роль обработчика события `FireHandler` в данном примере сводится к выводу на консоль сообщения о результатах своей работы. Приведу тестирующую процедуру из класса `Testing`, в которой создаются объекты классов `sender` и `receiver` и моделируется процесс возникновения события.

```
public void TestTown()
{
    string townName = "Энск";
    int buildings = 1000;
    int days = 30;
    double fireProbability = 1e-4;
    TownWithEvents town = new
        TownWithEvents(townName, buildings, days,fireProbability);
    FireMen fireMen = new FireMen(town);
    Console.WriteLine("События в городе " + townName);
    town.TownLife();
}
```

Вначале создается объект город "Энск", затем создается пожарная служба, обслуживающая этот город. Вызов метода `TownLife` моделирует жизнь города и возникновение в нем событий, которые, к сожалению, сводятся только к возникновению пожаров в домах города. Результаты работы этой процедуры показаны на рис. 10.3.

Рис. 10.3. Моделирование жизни города

10.2.3. Классы с событиями, допустимые в каркасе .Net Framework

Если создавать повторно используемые классы с событиями, работающие не только в проекте C#, то необходимо удовлетворять некоторым ограничениям. Эти требования предъявляются к делегату; они носят, скорее, синтаксический характер, не ограничивая существа дела.

Перечислю эти ограничения:

- делегат, задающий тип события, должен иметь фиксированную сигнатуру из двух аргументов: `delegate <Имя_делегата> (object sender, <Тип_аргументов> args)`;
- первый аргумент задает объект `sender`, создающий сообщение. Второй аргумент `args` задает остальные аргументы - входные и выходные, - передаваемые обработчику. Тип этого аргумента должен задаваться классом, наследником класса `EventArgs` из библиотеки классов FCL. Если обработчику никаких дополнительных аргументов не передается, то для второго аргумента следует просто указать класс `EventArgs`, передавая `null` в качестве фактического аргумента при включении события;
- рекомендуемое имя делегата - составное, начинающееся именем события, после которого следует слово `EventHandler`, например, `FireEventHandler`. Если никаких дополнительных аргументов обработчику не передается, то тогда можно вообще делегата не объявлять, а пользоваться стандартным делегатом с именем `EventHandler`.

10.2.4. Две проблемы с обработчиками событий

Объекты, создающие события, ничего не знают об объектах, обрабатывающих эти события. Объекты, обрабатывающие события, могут ничего не знать друг о друге, независимо выполняя свою работу. В такой модели могут возникать определенные проблемы. Рассмотрим некоторые из них.

Игнорирование коллег

Задумывались ли Вы, какую роль играет ключевое слово `event`, появляющееся при объявлении события? Событие, объявленное в классе, представляет экземпляр делегата.

Слово "`event`" играет важную роль, позволяя решить проблему, названную нами "**игнорированием коллег**". В чем ее суть? В том, что некоторые из классов `receiver` могут вести себя некорректно по отношению к своим коллегам, занимающимся обработкой того же события. При присоединении обработчика события в классе `receiver` можно попытаться вместо присоединения обработчика выполнить операцию присваивания, игнорируя уже присоединенный список обработчиков.

```
list.Changed += new ChangedEventHandler(ListChanged);
//list.Changed = new ChangedEventHandler(ListChanged);
```

Аналогично класс `receiver` может попытаться вместо отсоединения присвоить событию значение `null`, отсоединяя тем самым всех других обработчиков.

```
list.Changed -= new ChangedEventHandler(ListChanged);
//list.Changed = null;
```

С этим как-то нужно бороться. Ключевое слово "`event`" разрешает выполнять над событиями только операцию присоединения обработчика событий к списку вызовов "`+=`" и обратную операцию отсоединения "`-=`". Никаких других операций над событиями выполнять нельзя. Тем самым, к счастью, решается проблема игнорирования коллег. Ошибки некорректного поведения класса `receiver` ловятся еще на этапе трансляции. В приведенных примерах некорректные попытки работы закомментированы.

Изменение аргументов

То, что классы, обрабатывающие события, могут только отсоединять или присоединять обработчики события к общему списку вызовов, позволило решить одну из проблем совместной работы. Тем не менее, остаются проблемы, связанные с аргументами, которые передаются обработчикам события. Обработчику события, как правило, передаются входные и выходные аргументы, характеризующие событие. Они необходимы, чтобы обработчик мог нужным образом обработать событие. Но работа с аргументами требует аккуратного с ними обращения. При совместной работе обработчиков могут возникать проблемы, связанные с тем, что каждый из обработчиков может **изменить** значения переданных ему аргументов события в процессе своей работы.

Проблема защиты аргументов распадается на две проблемы - защита входных аргументов и защита выходных аргументов. Первая из этих проблем сводится к защите входных аргументов от попыток их изменений обработчиками события. Решить эту проблему можно достаточно просто при проектировании класса, порождающего события. Если быть более точным, проблема решается при проектировании класса, задающего входные и выходные аргументы события и являющегося наследником класса `EventArgs`. Поля этого класса делаются закрытыми, а методы свойства, реализующие доступ к полям, позволяют клиентам класса только чтение значений, запрещая запись в поля. Тем самым, ни один из классов `receiver` не сможет изменить значения входных аргументов.

Проблема с выходными аргументами является более сложной. В чем ее суть? Когда зажигается событие, поочередно начинают работать обработчики этого события. Обработчики вызываются в том порядке, в котором они присоединялись к событию. Обработчиков может быть много, а выходные аргументы - одни на всех, и в этом коллизия. Если обработчики работают, не обращая внимания друг на друга, то результаты работы обработчика события, сформировавшего значения выходных аргументов, будут потеряны из-за действий следующего в списке обработчика события. В конечном итоге значения выходных аргументов формирует обработчик события, работающий последним. Конечно, классы `receiver` могли бы договориться между собой и выработать некую общую стратегию формирования выходных аргументов. Но это не самый лучший выход, поскольку в процессе работы над системой могут появляться новые классы, и вообще классы `receiver` могут ничего не знать друг о друге.

Решение проблемы означает, что объект, зажигающий событие, может получить значения выходных аргументов от каждого из обработчиков события в отдельности. Как это сделать?

Когда объекту класса `sender` нужно зажечь событие, вместо посылки сообщения всем, кто его готов принять, объект получает список вызова, используя метод `GetInvocationList`, и поочередно вызывает обработчиков события, обрабатывая после вызова значения выходных аргументов.

10.2.5. Пример "Списки с событиями"

В этом примере строится класс `ListWithChangedEvent`, позволяющий работать со списками и являющийся потомком встроенного класса `ArrayList`. В класс добавляется событие `Changed`, сигнализирующее обо всех изменениях элементов списка. Начнем с объявления делегата:

```
// Объявление делегата
public delegate void ChangedEventHandler(object sender,
              ChangedEventArgs args);
```

Здесь объявлен делегат `ChangedEventHandler`, по всем правилам хорошего стиля - его имя и его форма соответствуют всем требованиям. Второй аргумент, задающий аргументы события, принадлежит классу `ChangedEventArgs`, производному от встроенного класса `EventArgs`. Рассмотрим, как устроен этот производный класс:

```
/// <summary>
/// Дополнительные аргументы события
/// </summary>
public class ChangedEventArgs : EventArgs
{
    string name;     //входной аргумент
    object item;     //входной аргумент
    bool permit;     //выходной аргумент
    public string Name
    { get { return name; } }     //только чтение
    public object Item
    { get { return (item); } }     //только чтение
    public bool Permit
    {
        get { return (permit); }
        set { permit = value; }  //чтение и запись
    }
    public ChangedEventArgs(string name, object item)
    {
        this.name = name; this.item = item;
    }
    public ChangedEventArgs()
    { }
}//class ChangedEventArgs
```

У класса три закрытых свойства, доступ к которым осуществляется через методы-свойства. Эти свойства задают дополнительные аргументы, которые передаются методам, обрабатывающим события. Два свойства - `name` и `item` - имя объекта и значение элемента списка - задают входную информацию, на основе которой обработчик события принимает решение. Третий аргумент - `permit` является выходным аргументом. Деление аргументов на входные и выходные выполняется на содержательном уровне и никак синтаксически не поддерживается. Разработчик класса, реализуя разные стратегии доступа к аргументам, тем самым разделяет их на входные и выходные. К входным аргументам открыт доступ только на чтение, так что ни один из обработчиков события не сможет изменить значения этих аргументов. Для выходного аргумента возможно как чтение, так и запись.

У класса два конструктора: один - без аргументов, второму передаются входные аргументы.

В модели, которую мы рассматриваем, предполагается, что обработчик события, получив уведомление об изменении элемента, анализирует ситуацию и может разрешить или не разрешить изменение, например, если значение элемента больше некоторого предельного значения.

Класс sender

Рассмотрим, как устроен в нашем примере класс, создающий события. Начнем определение класса с задания его свойств и конструктора:

```
/// <summary>
/// Класс, создающий событие.
 /// Потомок класса ArrayList.
/// </summary>
public class ListWithChangedEvent: ArrayList
{
 string name;     //имя объекта
   public event ChangedEventHandler Changed;    //событие
 bool permit;  //результат обработки события
   /// <summary>
      /// Конструктор
    /// </summary>
    /// <param name="name">имя объекта</param>
    public ListWithChangedEvent(string name)
     { this.name = name; }
     public string Name
     { get { return name; } }
```

Первое свойство задает имя объекта, чтобы обработчики могли узнать, кто послал сообщение. Второе свойство описывает событие `Changed`. Оно открыто, что позволяет присоединять к нему обработчиков событий. Третье свойство задает суммарный итог, сформированный на основании результатов работы всех обработчиков события.

Хороший стиль требует задания в классе процедуры `On`, включающей событие. Так и поступим:

```
/// <summary>
/// Процедура On, включающая событие
/// </summary>
/// <param name="args">аргументы события</param>
protected virtual void OnChanged(ChangedEventArgs args)
{
   if (Changed != null)
      Changed(this, args);
}
```

Процедура `OnChanged` соответствует ранее описанному образцу. Если список обработчиков не пуст, то зажигается событие - посылается сообщение всем обработчикам события. Синтаксически конструкция `Changed(this, args)` - это вызов списка методов, поскольку `Changed` - объект функционального типа, к которому прикреплен список последовательно работающих методов.

Если в списке аргументов `args` есть выходные аргументы, то для решения проблемы коллизии совместной работы обработчиков посылку сообщения обработчикам события следует устроить более сложным образом:

```
protected virtual void OnChanged(ChangedEventArgs args)
   {
     int countYes = 0, countNo = 0;
     if (Changed != null)
     {
        foreach (ChangedEventHandler del in Changed.GetInvocationList())
        {
             del(this, args);
             if (args.Permit) countYes++;
             else countNo++;
        }
        permit = (countYes >= countNo);
     }
   }
```

Метод `GetInvocationList` позволяет получить список обработчиков события, а цикл `foreach` - вызывать один обработчик за другим. После того, как обработчик завершится, можно понять, каково значение сформированного выходного аргумента события. В данном примере окончательное решение принимается по большинству голосов. Счетчики `countYes` и `countNo` считают голоса "за" и "против". Представленное здесь решение демонстрирует корректный способ работы с событиями, когда обработчикам необходимо передавать входные и выходные аргументы.

Перейдем теперь к рассмотрению того, как в нашем классе возникают события. Наш класс, являясь наследником класса `ArrayList`, наследует все его методы. Переопределим методы, изменяющие элементы:

- метод `Add`, добавляющий новый элемент в конец списка;
- индексатор `this`, дающий доступ к элементу списка по индексу;
- метод `Clear`, производящий чистку списка.

```
// Переопределяемые методы, вызывающие событие Changed
      public override int Add(object value)
      {
         int index = -1;
         ChangedEventArgs evargs =
                new ChangedEventArgs(name, value);
         OnChanged(evargs);
         if (permit)
            index = base.Add(value);
         return index;
      }
```

Обратите внимание на схему включения события в процедуре `Add`. Вначале создается объект `evargs` - аргументы события, который передается методу `OnChanged`. Этот метод поочередно вызовет обработчики события и сформирует итоговый результат их работы. Анализ переменной `permit` позволяет установить, получено ли разрешение на изменение значения. При истинности значения этой переменной вызывается родительский метод `Add`, осуществляющий изменение значения. Аналогично устроены и другие методы, в которых возникает событие `Changed`.

```
public override void Clear()
{
   ChangedEventArgs evargs =
    new ChangedEventArgs(name, 0);
   OnChanged(evargs);
   base.Clear();
```

```
}

public override object this[int index]
{
   set
   {
      ChangedEventArgs evargs =
              new ChangedEventArgs(name, value);
      OnChanged(evargs);
          if (permit)
         base[index] = value;
   }
   get   {return(base[index]);}
}
```

Это достаточно типичная схема организации класса с событиями.

Классы receiver

Построим два класса, объекты которых способны получать и обрабатывать событие `Changed`, возникающее у объектов класса `ListWithChangedEvent`. Вначале разберемся с устройством одного из этих классов, названного `Receiver1`. Вот его код:

```
class Receiver1
{
    private ListWithChangedEvent list;
    /// <summary>
    /// Конструктор
    /// Присоединяет обработчик к событию
    /// </summary>
    /// <param name="list">объект, посылающий сообщение</param>
    public Receiver1(ListWithChangedEvent list)
    {
        this.list = list;
        // Присоединяет обработчик к событию.
        list.Changed += new ChangedEventHandler(ListChanged);
    }
    public void OffConnect()
    {
        // Отсоединяет обработчик
        list.Changed -= new ChangedEventHandler(ListChanged);
    }
}//class Receiver1
```

Класс `Receiver1` является примером класса, чьи объекты настроены на прием сообщения от одного объекта класса `ListWithChangedEvent`. Такой объект передается конструктору класса, который и выполняет всю необходимую работу, присоединяя обработчик события к событию переданного конструктору объекта, так что вызов конструктора класса `Receiver1` приводит к созданию объекта, способного принимать и обрабатывать сообщения от объекта, задающего список.

Давайте подробнее рассмотрим устройство обработчика события:

```
private void ListChanged(object sender, ChangedEventArgs args)
  {
    Console.WriteLine("Сообщение послал {0}, " +
    "элемент = {1}. " + " Сообщение получил: Receiver1 - ",
    args.Name, args.Item);
```

```
    if(args.Permit = (int)args.Item < 10)
        Console.WriteLine("Изменения разрешаю");
    else Console.WriteLine("Изменения не разрешаю");
      }
```

С входными аргументами все просто. Они используются для формирования сообщения и формирования принимаемого решения. Но стоит подробнее разобраться, как в данном конкретном случае формируется выходной аргумент. Свое решение "Разрешить или не разрешить изменение" обработчик принимает в зависимости от величины элемента (`item`). Свое решение обработчик события принимает независимо, без оглядки на методы, совместно с ним обрабатывающими то же событие.

Класс `Receiver2` в отличие от класса `Receiver1` позволяет слушать и обрабатывать сообщения нескольких объектов класса `sender`. С конструктора класса снимается задача связывания события с обработчиком события. Ему не нужно теперь передавать объект класса `sender`. У класса появляется специальный метод `OnConnect`, которому передается объект класса `sender`. Присоединение обработчика события к событию объекта `sender` выполняется при каждом вызове метода `OnConnect`.

События, приходящие от разных объектов, могут обрабатываться одним обработчиком класса `receiver`. Возможно существование в классе набора обработчиков событий, чтобы разные объекты могли иметь и разные методы, обрабатывающие приходящее сообщение о возникшем событии. Так строятся инерфейсные классы, где у каждой командной кнопки свой обработчик события. В нашем примере предусмотрен один обработчик для всех объектов одного класса.

```
class Receiver2
{
    void ListChanged(object sender, ChangedEventArgs args)
    {
        Console.WriteLine("Сообщение послал {0}, " +
        "элемент = {1}. " + " Сообщение получил: Receiver2 - ",
        args.Name, args.Item);
        if (args.Permit = (int)args.Item < 20)
            Console.WriteLine("Изменения разрешаю");
        else Console.WriteLine("Изменения не разрешаю");
    }
    public void OnConnect(ListWithChangedEvent list)
    {
        list.Changed += new ChangedEventHandler(ListChanged);
        //list.Changed = new ChangedEventHandler(ListChanged);
    }
    public void OffConnect(ListWithChangedEvent list)
    {
        list.Changed -= new ChangedEventHandler(ListChanged);
        //list.Changed = null;
    }
}//class Receiver2
```

Классы созданы, теперь осталось создать объекты и заставить их взаимодействовать, чтобы одни создавали события, а другие их обрабатывали. Эту часть работы будет выполнять тестирующая процедура класса `Testing`:

```
public void TestChangeList()
{
   // Создаются два объекта, вырабатывающие события
   ListWithChangedEvent list1 = new ListWithChangedEvent("list1");
```

```
      ListWithChangedEvent list2 = new ListWithChangedEvent("list2");

   // Создаются два объекта классов Receiver1 и Receiver2,
   //способные обрабатывать события класса ListWithChangedEvent
   Receiver1 receiver1 = new Receiver1(list1);
   Receiver2 receiver2 = new Receiver2();
      receiver2.OnConnect(list1);
   receiver2.OnConnect(list2);

// Работа с объектами, приводящая к появлению событий
   Random rnd = new Random();
   list1.Add(rnd.Next(20)); list1.Add(rnd.Next(20));
      list1.Add(33); list1[1] = 17;
   list2.Add(10);   list2[0] = 25;
      list2.Clear();

   //Отсоединение обработчика событий
   receiver1.OffConnect();
   list1.Add(21);
      list1.Clear();
}
```

В тестирующей процедуре моделируется процесс работы с объектами, посылающими сообщения о событиях и принимающими эти сообщения. Два созданных объекта `list1` и `list2` посылают сообщения о событиях всякий раз, когда в список добавляется новый элемент или изменяется значение существующего элемента. Два созданных объекта `receiver1` и `receiver2` получают приходящие сообщения. Первый из них получает сообщения только от объекта `list1`, второй - от двух объектов `list1` и `list2`. В некоторых ситуациях оба получателя сообщений "дают добро" на изменение элемента, в других ситуациях оба запрещают изменения.

В заключение взгляните на результаты работы этой процедуры.

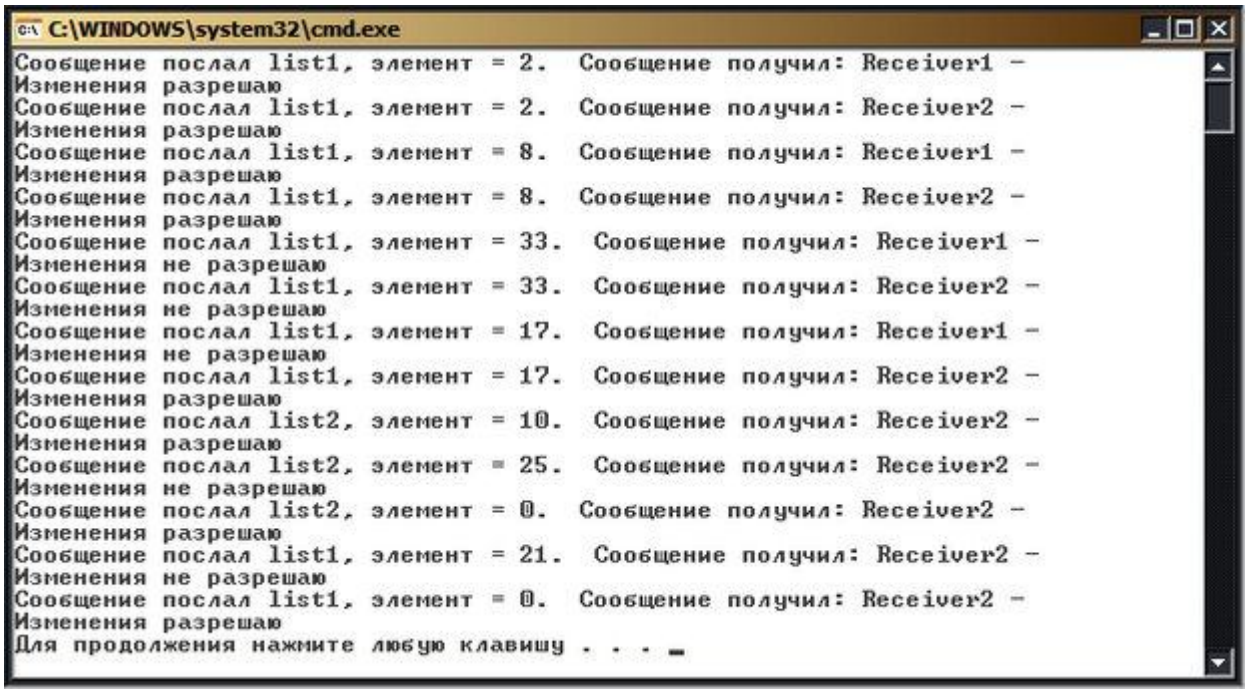

Рис. 10.4. События в мире объектов list

10.2.6. Классы с большим числом событий

Как было сказано, каждое событие класса представляется полем этого класса. Если у класса много объявленных событий, а реально возникает лишь малая часть из них, то предпочтительнее **динамический подход**, когда память отводится только событиям, для которых есть связанные с ними обработчики события. Это несколько замедляет время выполнения, но экономит память. Решение зависит от того, что в данном контексте важнее - память или время. Для реализации динамического подхода в момент объявления события в языке предусмотрена возможность задания пользовательских методов `Add` и `Remove`. Это и есть другая форма объявления события, упоминавшаяся ранее. Вот ее примерный синтаксис:

```
public event <Имя Делегата> <Имя события>
{
   add {…}
   remove {…}
}
```

Оба метода должны быть реализованы, при этом для хранения делегатов используется некоторое хранилище. Именно так реализованы классы для большинства интерфейсных объектов, использующие хэш-таблицы для хранения делегатов.

Давайте построим небольшой пример, демонстрирующий такой способ объявления и работы с событиями. Вначале построим класс с несколькими событиями:

```
class ManyEvents
{
    //хэш-таблица для хранения делегатов
    Hashtable DStore = new Hashtable();
    public event EventHandler Ev1
    {
        add { DStore["Ev1"] = (EventHandler)DStore["Ev1"] + value; }
        remove { DStore["Ev1"] = (EventHandler)DStore["Ev1"] - value; }
    }
    //Аналогично объявляются события Ev2, Ev3, Ev4
    public void SimulateEvs()
    {
        EventHandler ev = (EventHandler)DStore["Ev1"];
        if (ev != null) ev(this, null);
        ev = (EventHandler)DStore["Ev3"];
        if (ev != null) ev(this, null);
    }
}//class ManyEvents
```

В нашем классе созданы четыре события и хэш-таблица `DStore` для их хранения. Все события принадлежат встроенному классу `EventHandler`. Когда к событию будет присоединяться обработчик, автоматически будет вызван метод `add`, который динамически создаст элемент хэш-таблицы, Ключом элемента в данном случае является строка с именем события. При отсоединении обработчика будет исполняться метод `remove`, выполняющий аналогичную операцию над соответствующим элементом хэш-таблицы. В классе определен также метод `SimulateEvs`, при вызове которого зажигаются два из четырех событий - `Ev1` и `Ev3`.

Рассмотрим теперь класс `Receiver3`, который слушает события, приходящие от объектов класса `ManyEvents`. Этот класс построен по описанным ранее правилам. В нем есть четыре

обработчика события и метод `OnConnect`, связывающий обработчиков с событиями. Вот код класса:

```
class Receiver3
{
    string name;
    public Receiver3(string name)
    {
        this.name = name;
    }
    public string Name { get { return name; } }
    public void OnConnect(ManyEvents me, int number)
    {
        switch (number)
        {
            case 1: { me.Ev1 += new EventHandler(H1); break; }
            case 2: { me.Ev2 += new EventHandler(H2); break; }
            case 3: { me.Ev3 += new EventHandler(H3); break; }
            case 4: { me.Ev4 += new EventHandler(H4); break; }
        }
    }
    public void H1(object s, EventArgs e)
    {
        Console.WriteLine("Слушатель " + name +
            " : Событие Ev1");
    }
    public void H2(object s, EventArgs e)
    {
        Console.WriteLine("Слушатель " + name +
            " : Событие Ev2");
    }
    public void H3(object s, EventArgs e)
    {
        Console.WriteLine("Слушатель " + name +
            " : Событие Ev3");
    }
    public void H4(object s, EventArgs e)
    {
        Console.WriteLine("Слушатель " + name +
            " : Событие Ev4");
    }
}
```

Методу `OnConnect` передается объект класса `ManyEvents` и номер события, которое следует слушать. При выполнении этого метода происходит связывание события с его обработчиком и, как следствие, динамическое создание события и заполнение соответствующей строки хэш-таблицы объекта.

В тестирующей процедуре создаются один объект класса `ManyEvents` и два объекта класса `Receiver3`.

```
public void TestEvents()
  {
      ManyEvents me = new ManyEvents();
      Receiver3 rev1 = new Receiver3("First");
      rev1.OnConnect(me, 1);
      rev1.OnConnect(me, 3);
      Receiver3 rev2 = new Receiver3("Second");
      rev2.OnConnect(me, 2);
      rev2.OnConnect(me, 3);
```

```
        me.SimulateEvs();
    }
```

Объект `rev1` слушает первое и третье события, приходящие от объекта `me`. Объект `rev2` слушает второе и третье события, приходящие от объекта `me`. Метод `Simulate` зажигает первое и третье события. Слушатели сообщений получают уведомления о возникших событиях и реагируют должным образом. Результаты работы показаны на рис. 10.5.

Рис. 10.5. Объекты с множественными событиями

10.2.7. Проект "Город и его службы"

Построим проект, в котором собраны все рассматриваемые аспекты работы с событиями. Вернемся к моделированию жизни города, происходящих в нем событий, реакции на них городских служб. Наша главная цель в данном проекте еще раз показать, как возникающее событие, в данном случае - пожар в одном из домов города, обрабатывается по-разному городскими службами - пожарными, милицией, скорой помощью. В нашей модели сообщение о событии получает не одна, как ранее, а несколько служб. Событие имеет как входные, так и выходные аргументы, так что городу необходимо будет анализировать результаты работы каждой службы. Стоит обратить внимание и на организацию взаимодействия между классами, задающего город и его службы. Конечно, все упрощено, в реальном городе событиями являются не только пожары и преступления, но и более приятные ситуации: день города, открытие фестивалей и выставок, строительство новых театров и институтов.

Начнем с описания делегата, задающего событие "пожар" и построенного по всем правилам.

```
public delegate void FireEventHandler(object sender, FireEventArgs e);
```

А теперь рассмотрим свойства класса, определяющего новый город.

```
/// <summary>
   /// Модель города с событиями
    /// и следящих за ними службами города
   /// </summary>
   public class NewTown
   {
      //свойства
    string townName;     //название города
      int buildings;        //число домов в городе
      int days;          //число дней наблюдения
      //городские службы
      Police policeman;
      Ambulance ambulanceman;
      FireDetect fireman;
      //события в городе
      public event FireEventHandler Fire;
```

```
        string[] resultService;      //результаты действий служб
      //моделирование случайных событий
      private Random rnd = new Random();
      //вероятность пожара в доме в текущий день
        double fireProbability;
```

В нашем городе есть дома; есть время, текущее день за днем; городские службы; событие "пожар", которое, к сожалению, может возникать с заданной вероятностью каждый день в каждом доме. Рассмотрим конструктор объектов нашего класса:

```
/// <summary>
      /// Конструктор города
        /// Создает службы и включает наблюдения
        /// за событиями
      /// </summary>
      /// <param name="name">название города</param>
      /// <param name="buildings">число домов</param>
      /// <param name="days">число дней наблюдения</param>
      public NewTown(string name, int buildings, int days)
      {
       townName = name;
       this.buildings = buildings;
         this.days = days;
       fireProbability = 1e-3;
            //Создание служб
         policeman = new Police(this);
         ambulanceman= new Ambulance(this);
         fireman= new FireDetect(this);
            //Подключение к наблюдению за событиями
         policeman.On();
         ambulanceman.On();
         fireman.On();
      }
```

Конструктору передается имя города, число домов в нем и период времени, в течение которого будет моделироваться жизнь города. Конструктор создает службы города - объекты соответствующих классов `Police`, `Ambulance`, `FireDetect`, передавая им ссылку на сам объект "город". После создания служб вызываются их методы `On`, подключающие обработчики события `Fire` каждой из этих служб к событию.

А теперь самое главное - определим метод `OnFire`, зажигающий события, с учетом того, что несколько служб слушают событие и у события есть выходные аргументы, задающие результаты работы каждой службы.

```
/// <summary>
/// Зажигается событие.
/// Поочередно вызываются обработчики события
/// </summary>
/// <param name="e">
/// входные и выходные аргументы события
/// </param>
protected virtual void OnFire(FireEventArgs e)
      {
    const string MESSAGE_FIRE =
                "В городе {0} пожар! Дом {1}. День {2}-й";
    Console.WriteLine(string.Format(MESSAGE_FIRE, townName,
        e.Building, e.Day));
    if (Fire != null)
    {
```

```
        Delegate[] eventHandlers =
            Fire.GetInvocationList();
        resultService = new string[eventHandlers.Length];
        int k = 0;
        foreach (FireEventHandler evHandler in
            eventHandlers)
        {
            evHandler(this, e);
            resultService[k++] = e.Result;
      }
    }
  }
```

Обратите внимание: метод `GetInvocatonList` возвращает массив объектов класса `Delegate`, который является абстрактным классом и родителем для классов событий, в частности для класса `FireEventHandler`. Получив этот массив, далее с ним можно работать привычным образом. В цикле по элементам массива вызывается очередной обработчик события, результаты его работы сохраняются в специально созданном массиве `resultService`.

Где и когда будет включаться событие `Fire`? Напишем метод, моделирующий жизнь города, где для каждого дома каждый день будет проверяться, а не возник ли пожар, и, если это случится, будет включено событие `Fire`:

```
/// <summary>
/// Моделирование жизни города
/// </summary>
  public void LifeOurTown()
{
      const string OK =
         "В городе {0} все спокойно! Пожаров не было.";
      bool wasFire = false;
      for(int day = 1; day <= days; day++)
      for(int building = 1; building <= buildings; building++)
      {
              if (rnd.NextDouble() < fireProbability)
              {
                  FireEventArgs e = new FireEventArgs(building, day);
                  OnFire(e);
                  wasFire = true;
                  for (int i = 0; i < resultService.Length; i++)
                      Console.WriteLine(resultService[i]);
              }
      }
      if (!wasFire)
          Console.WriteLine(string.Format(OK, townName));
}
```

Рассмотрим теперь классы `receiver`, обрабатывающие событие `Fire`. Их у нас три, по одному на каждую городскую службу. Все три класса устроены по одному образцу. Напомню: каждый такой разумно устроенный класс, кроме обработчика события, имеет конструктор, инициализирующий ссылку на объект, создающий события, методы подключения и отсоединения обработчика от события. В такой ситуации целесообразно построить вначале абстрактный класс `Receiver`, в котором будет предусмотрен обработчик события, но не задана его реализация, а затем для каждой службы построить класс потомок. Начнем с описания родительского класса:

```
public abstract class Receiver
```

```
{
    protected NewTown town;
    protected Random rnd = new Random();
    public Receiver(NewTown town)
    { this.town = town; }

    public void On()
    {
        town.Fire += new FireEventHandler(It_is_Fire);
    }
    public void Off()
    {
        town.Fire -= new FireEventHandler(It_is_Fire);
    }
    public abstract void It_is_Fire(object sender, FireEventArgs e);
}//class Receiver
```

Каждый из классов потомков устроен одинаково - имеет конструктор и задает реализацию абстрактного метода `It_is_Fire`. Вот описания этих классов:

```
public class Police : Receiver
{
    public Police(NewTown town) : base(town) { }
    public override void It_is_Fire(object sender, FireEventArgs e)
    {
        const string OK =
            "Милиция нашла виновных!";
        const string NOK =
            "Милиция не нашла виновных! Следствие продолжается.";
        if (rnd.Next(0, 10) > 6)
            e.Result = OK;
        else e.Result = NOK;
    }
}// class Police
public class FireDetect : Receiver
{
    public FireDetect(NewTown town) : base(town) { }
    public override void It_is_Fire(object sender, FireEventArgs e)
    {
        const string OK =
             "Пожарные потушили пожар!";
        const string NOK =
             "Пожар продолжается! Требуется помощь.";
        if (rnd.Next(0, 10) > 4)
            e.Result = OK;
        else e.Result = NOK;
    }
}// class FireDetect
public class Ambulance : Receiver
{
    public Ambulance(NewTown town) : base(town) { }
    public override void It_is_Fire(object sender, FireEventArgs e)
    {
        const string OK =
            "Скорая оказала помощь!";
        const string NOK =
            "Есть пострадавшие! Требуются лекарства.";
        if (rnd.Next(0, 10) > 2)
            e.Result = OK;
        else e.Result = NOK;
    }
}// class Ambulance
```

Для полноты картины необходимо показать, как выглядит класс, задающий аргументы события, который, как и положено, является потомком класса `EventArgs`:

```
/// <summary>
/// Класс,задающий входные и выходные аргументы события
/// </summary>
public class FireEventArgs : EventArgs
{
    int building;
    int day;
    string result;
    //Доступ к входным и выходным аргументам
    public int Building
    { get { return building; } }
    public int Day
    { get { return day; } }
    public string Result
    {
        get { return result; }
        set { result = value; }
    }
    public FireEventArgs(int building, int day)
    {
        this.building = building; this.day = day;
    }
}//class FireEventArgs
```

Входные аргументы события - `build` и `day` защищены от обработчиков события, а корректность работы с выходным аргументом гарантируется аккуратным программированием вызова обработчиков.

Для завершения проекта нам осталось определить тестирующую процедуру в классе `Testing`, создающую объекты и запускающую моделирование жизни города:

```
public void TestLifeTown()
      {
       NewTown sometown = new NewTown("Канск", 20, 100);
         sometown.LifeOurTown();
      }
```

Результаты ее работы зависят от случайных событий. Вот как выглядит один из экспериментов.

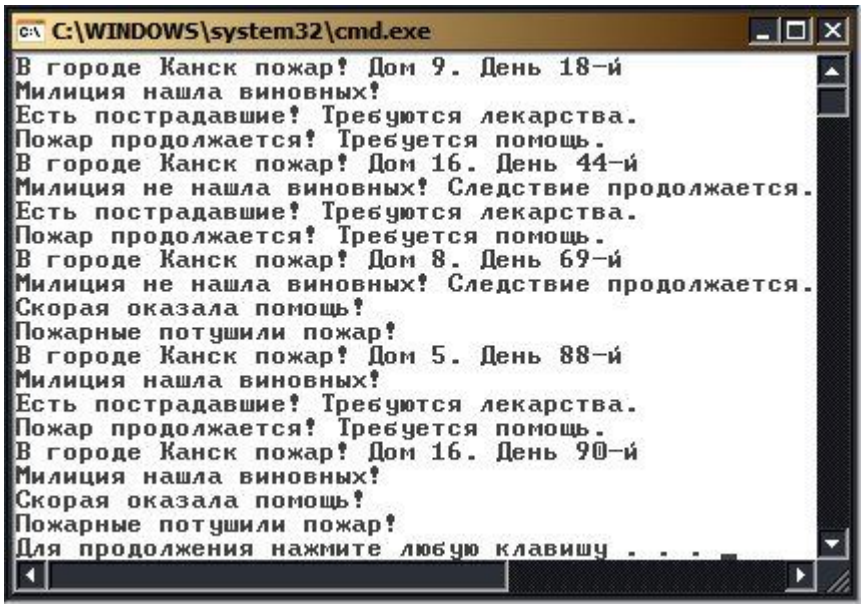

Рис. 10.6. События в жизни города и три обработчика

10.2.8. Класс Thread

В .NET многопоточность поддерживается в основном с помощью пространства имен `System.Threading`. Некоторые типы этого пространства описаны в таблице 10.1.

Таблица 10.1. Некоторые типы пространства имен System.Threading

Тип	Описание
Monitor	Класс, обеспечивающий синхронизацию доступа к объектам
Mutex	Класс-примитив синхронизации, который используется также для синхронизации между процессами
Thread	Класс, который создает поток, устанавливает его приоритет, получает информацию о состоянии
ThreadPool	Класс, используемый для управления набором взаимосвязанных потоков — пулом потоков
Timer	Класс, определяющий механизм вызова заданного метода в заданные интервалы времени для пула потоков
WaitHandle	Класс, инкапсулирующий объекты синхронизации, которые ожидают доступа к разделяемым ресурсам
ThreadStart	Делегат, представляющий метод, который должен быть выполнен при запуске потока
TimerCallback	Делегат, представляющий метод, обрабатывающий вызовы от класса `Timer`
WaitCallback	Делегат, представляющий метод для элементов класса `ThreadPool`
ThreadPriority	Перечисление, описывающее приоритет потока
ThreadState	Перечисление, описывающее состояние потока

Первичный поток создается автоматически. Для запуска вторичных потоков используется класс `Thread`. При создании объекта-потока ему передается делегат, определяющий метод, выполнение которого выделяется в отдельный поток:

```
Thread t = new Thread ( new ThreadStart( имя_метода ) );
```

После создания потока заданный метод начинает в нем свою работу, а первичный поток продолжает выполняться. Рассмотрим пример одновременной работы двух потоков.

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
    class Program
    {
        static public void Hedgehog()          // метод для вторичного потока
        {
            for (int i = 0; i < 6; ++i)
            {
                Console.Write(" " + i); Thread.Sleep(1000);
            }
        }

        static void Main()
        {
            Console.WriteLine("Первичный поток " +
                Thread.CurrentThread.GetHashCode());

            Thread ta = new Thread(new ThreadStart(Hedgehog));
            Console.WriteLine("Вторичный поток " + ta.GetHashCode());
            ta.Start();

            for (int i = 0; i > -6; --i)
            {
                Console.Write(" " + i); Thread.Sleep(400);
            }
        }
    }
}
```

Результат работы программы:

```
Первичный поток 1
Вторичный поток 2
 0 0 -1 -2 1 -3 -4 2 -5 3 4 5
```

В листинге используется метод `Sleep`, останавливающий функционирование потока на заданное количество миллисекунд. Как видите, оба потока работают одновременно. Если бы они работали с одним и тем же файлом, он был бы испорчен так же, как и приведенный вывод на консоль, поэтому такой способ распараллеливания вычислений имеет смысл только для работы с различными ресурсами.

В таблице 10.2 перечислены основные элементы класса `Thread`.

Таблица 10.2. Основные элементы класса Thread

Элемент	Вид	Описание
`CurrentThread`	Статическое свойство	Возвращает ссылку на выполняющийся поток (только для чтения)
`Name`	Свойство	Установка текстового имени потока
`Priority`	Свойство	Получить/установить приоритет потока (используются значения перечисления `ThreadPriority`)
`ThreadState`	Свойство	Возвращает состояние потока (используются значения перечисления `ThreadState`)

Abort	Метод	Генерирует исключение `ThreadAbortException`. Вызов этого метода обычно завершает работу потока
Sleep	Статический метод	Приостанавливает выполнение текущего потока на заданное количество миллисекунд
Interrupt	Метод	Прерывает работу текущего потока
Join	Метод	Блокирует вызывающий поток до завершения другого потока или указанного промежутка времени и завершает поток
Resume	Метод	Возобновляет работу после приостановки потока
Start	Метод	Начинает выполнение потока, определенного делегатом `ThreadStart`
Suspend	Метод	Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется

Можно создать несколько потоков, которые будут совместно использовать один и тот же код.

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
    class Class1
    {
        public void Do()
        {
            for (int i = 0; i < 4; ++i)
            { Console.Write(" " + i); Thread.Sleep(3); }
        }
    }

    class Program
    {
        static void Main()
        {
            Class1 a = new Class1();
            Thread t1 = new Thread(new ThreadStart(a.Do));
            t1.Name = "Second";
            Console.WriteLine("Поток " + t1.Name);
            t1.Start();

            Thread t2 = new Thread(new ThreadStart(a.Do));
            t2.Name = "Third";
            Console.WriteLine("Поток " + t2.Name);
            t2.Start();
        }
    }
}
```

Результат работы программы:

```
Поток Second
Поток Third
 0 0 1 1 2 2 3 3
```

Варианты вывода могут несколько различаться, поскольку один поток прерывает выполнение другого в неизвестные моменты времени.

Для того чтобы блок кода мог использоваться в каждый момент только одним потоком, применяется оператор `lock`. Формат оператора:

```
lock ( выражение ) блок_операторов
```

Выражение определяет объект, который требуется заблокировать. Для обычных методов в качестве выражения используется ключевое слово `this`, для статических — `typeof(` *класс* `)`. *Блок операторов* задает критическую секцию кода, которую требуется заблокировать.

Например, блокировка операторов в приведенном ранее методе `Do` выглядит следующим образом:

```
public void Do()
{
    lock( this )
    {
        for ( int i = 0; i < 4; ++i )
            { Console.Write( " " + i ); Thread.Sleep( 30 ); }
    }
}
```

Для такого варианта метода результат работы программы изменится:

```
Поток Second
Поток Third
 0 1 2 3 0 1 2 3
```

10.2.9. Асинхронные делегаты

Делегат можно вызвать на выполнение либо синхронно, как во всех приведенных ранее примерах, либо асинхронно с помощью методов `BeginInvoke` и `EndInvoke`.

При вызове делегата с помощью метода `BeginInvoke` среда выполнения создает для исполнения метода отдельный поток и возвращает управление оператору, следующему за вызовом. При этом в исходном потоке можно продолжать вычисления.

Если при вызове `BeginInvoke` был указан метод обратного вызова, этот метод вызывается после завершения потока. Метод обратного вызова также задается с помощью делегата, при этом используется стандартный делегат `AsyncCallback`. В методе обратного вызова для получения возвращаемого значения и выходных параметров применяется метод `EndInvoke`.

Если метод обратного вызова не был указан в параметрах метода `BeginInvoke`, метод `EndInvoke` можно использовать в потоке, инициировавшем запрос.

Далее приводится два примера асинхронного вызова метода, выполняющего разложение числа на множители. Листинг приводится по документации Visual Studio с некоторыми изменениями.

Класс `Factorizer` содержит метод `Factorize`, выполняющий разложение на множители. Этот метод асинхронно вызывается двумя способами: в методе `Num1` метод обратного

вызова задается в `BeginInvoke`, в методе `Num2` имеют место ожидание завершения потока и непосредственный вызов `EndInvoke`.

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;
                                                    // асинхронный делегат
public delegate bool AsyncDelegate ( int Num, out int m1, out int m2 );

                           // класс, выполняющий разложение числа на множители
public class Factorizer
{
    public bool Factorize( int Num, out int m1, out int m2 )
    {
        m1 = 1;    m2 = Num;
        for ( int i = 2; i < Num; i++ )
            if ( 0 == (Num % i) ) { m1 = i; m2 = Num / i; break; }

        if (1 == m1 ) return false;
        else          return true;
    }
}

                                     // класс, получающий делегат и результаты
public class PNum
{
    private int Number;
    public PNum( int number ) { Number = number; }

    [OneWayAttribute()]
                                                // метод, получающий результаты
    public void Res( IAsyncResult ar )
    {
        int m1, m2;
                                          // получение делегата из AsyncResult
        AsyncDelegate ad = (AsyncDelegate)((AsyncResult)ar).AsyncDelegate;

                         // получение результатов выполнения метода Factorize
        ad.EndInvoke( out m1, out m2, ar );
                                                          // вывод результатов
        Console.WriteLine( "Первый способ : множители {0} : {1} {2}",
                           Number, m1, m2 );
    }
}
                                                  // демонстрационный класс
public class Simple
{
                          // способ 1: используется функция обратного вызова
    public void Num1()
    {
        Factorizer     f = new Factorizer();
        AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

        int Num = 1000589023, tmp;
                        // создание экземпляра класса, который будет вызван
                                  // после завершения работы метода Factorize
        PNum n = new PNum( Num );

                                  // задание делегата метода обратного вызова
        AsyncCallback callback = new AsyncCallback( n.Res );

                                       // асинхронный вызов метода Factorize
```

```
        IAsyncResult ar = ad.BeginInvoke(
                                Num, out tmp, out tmp, callback, null );
        //
        // здесь - выполнение неких дальнейших действий
        // ...
    }
                    // способ 2: используется ожидание окончания выполнения
    public void Num2()
    {
        Factorizer     f = new Factorizer();
        AsyncDelegate ad = new AsyncDelegate ( f.Factorize );

        int Num = 1000589023, tmp;

                        // создание экземпляра класса, который будет вызван
                                // после завершения работы метода Factorize
        PNum n = new PNum( Num );

                                // задание делегата метода обратного вызова
        AsyncCallback callback = new AsyncCallback( n.Res );

                                    // асинхронный вызов метода Factorize
        IAsyncResult ar = ad.BeginInvoke(
                                Num, out tmp, out tmp, null, null );
                                                    // ожидание завершения
        ar.AsyncWaitHandle.WaitOne( 10000, false );

        if ( ar.IsCompleted )
        {
            int m1, m2;
                            // получение результатов выполнения метода Factorize
            ad.EndInvoke( out m1, out m2, ar );
                                                        // вывод результатов
            Console.WriteLine( "Второй способ : множители {0} : {1} {2}",
                                Num, m1, m2 );
        }
    }

    public static void Main()
    {
        Simple s = new Simple();
        s.Num1();
        s.Num2();
    }
}
```

Результат работы программы:

```
Первый способ : множители 1000589023 : 7 142941289
Второй способ : множители 1000589023 : 7 142941289
```

Примечание

Атрибут `[OneWayAttribute()]` *помечает метод как не имеющий возвращаемого значения и выходных параметров.*

10.2.10. Параллельные коллекции

В версию 4.0 среды .NET Framework добавлено новое пространство имен System.Collections.Concurrent. Оно содержит коллекции, которые являются потокобезопасными и специально предназначены для параллельного программирования. Это означает, что они могут безопасно использоваться в многопоточной программе, где возможен одновременный доступ к коллекции со стороны двух или больше параллельно исполняемых потоков.

Для безопасного в отношении потоков доступа к коллекциям определен интерфейс IProducerConsumerCollection<T>. Наиболее важными методами этого интерфейса являются TryAdd() и TryTake(). Метод TryAdd() пытается добавить элемент в коллекцию, но это может не получиться, если коллекция заблокирована от добавления элементов. Метод возвращает булевское значение, сообщающее об успехе или неудаче операции.

TryTake() работает аналогичным образом, информируя вызывающий код об успехе или неудаче, и в случае успеха возвращает элемент из коллекции. Ниже перечислены классы из пространства имен System.Collections.Concurrent с кратким описанием их функциональности:

ConcurrentQueue<T>

Этот класс коллекции реализован со свободным от блокировок алгоритмом и использует 32 массива, которые внутренне скомбинированы в связный список. Для доступа к элементам очереди применяются методы Enqueue(), TryDequeue() и TryPeek(). Имена этих методов очень похожи на уже известные методы Queue<T>, но с добавлением префикса Try к тем из них, которые могут дать сбой. Поскольку этот класс реализует интерфейс IProducerConsumerCollection<T>, методы TryAdd() и TryTake() просто вызывают Enqueue() и TryDequeue().

ConcurrentStack<T>

Очень похож на ConcurrentQueue<T>, но с другими методами доступа к элементам. Класс ConcurrentStack<T> определяет методы Push(), PushRange(), TryPeek(), TryPop() и TryPopRange(). Внутри этот класс использует связный список для хранения элементов.

ConcurrentBag<T>

Этот класс не определяет никакого порядка для добавления или извлечения элементов. Он реализует концепцию отображения потоков на используемые внутренне массивы, и старается избежать блокировок. Для доступа к элементам применяются методы Add(), TryPeek() и TryTake().

ConcurrentDictionary<TKey, TValue>

Безопасная в отношении потоков коллекция ключей и значений. Для доступа к членам в неблокирующем режиме служат методы TryAdd(), TryGetValue(), TryRemove() и TryUpdate(). Поскольку элементы основаны на ключах и значениях, ConcurrentDictionary<TKey, TValue> не реализует интерфес IProducerConsumerCollection<T>.

ConcurrentXXX

Эти коллекции безопасны к потокам в том смысле, что возвращают false, если какое-то действие над ними невозможно при текущем состоянии потоков. Прежде чем предпринимать какие-то дальнейшие действия, всегда следует проверять успешность добавления или извлечения элементов. Полностью доверять коллекции решение задачи нельзя.

BlockingCollection<T>

Коллекция, которая осуществляет блокировку и ожидает, пока не появится возможность выполнить действие по добавлению или извлечению элемента. BlockingCollection<T> предлагает интерфейс для добавления и извлечения элементов методами Add() и Take(). Эти методы блокируют поток и затем ожидают, пока не появится возможность выполнить задачу.

Метод Add() имеет перегрузку, которой можно также передать CancellationToken. Эта лексема всегда отменяет блокирующий вызов.

Если не нужно, чтобы поток ожидал бесконечное время, и не хотите отменять вызов извне, доступны также методы TryAdd() и TryTake(). В них можно указать значение таймаута — максимального периода времени, в течение которого вы готовы блокировать поток и ждать, пока вызов не даст сбой.

Давайте рассмотрим пример применения параллельных коллекций:

В следующем примере описывается параллельное добавление и получение элементов из коллекции блокировки.

```
class AddTakeDemo
{
        // Demonstrates:
        //      BlockingCollection<T>.Add()
        //      BlockingCollection<T>.Take()
        //      BlockingCollection<T>.CompleteAdding()
        static void Main()
        {
            BlockingCollection<int> bc = new BlockingCollection<int>();

            // Spin up a Task to populate the BlockingCollection
            Task t1 = Task.Factory.StartNew(() =>
            {
                bc.Add(1);
                bc.Add(2);
                bc.Add(3);
                bc.CompleteAdding();
            });

            // Spin up a Task to consume the BlockingCollection
            Task t2 = Task.Factory.StartNew(() =>
            {
                try
                {
                    // Consume bc
                    while (true) Console.WriteLine(bc.Take());
                }
                catch (InvalidOperationException)
                {
```

```
                    // IOE means that Take() was called on a completed collection
                    Console.WriteLine("That's All!");
                }
            });

            Task.WaitAll(t1, t2);
        }
}
```

10.3. Анонимные методы и лямбда-выражения

Анонимные методы прекрасно справляются с поставленной перед ними задачей. Но в C#, начиная с версии 3.0 пошли дальше, введя более привычную для математиков форму записи анонимного метода в виде лямбда-выражения. Вместо записи анонимного метода в форме:

```
delegate [(<сигнатура метода>)] <тело метода>
```

используется форма, задающая лямбда-выражение:

```
[(<сигнатура метода>)] =>   <тело метода>
```

В методе `TestAnonymous` анонимный метод определяется следующим образом:

```
double result = integral.EvalIntegral(a, b, eps,
         delegate(double x)
         { return  Math.Sin(x) + Math.Cos(x); });
```

Заменим это определение лямбда-выражением:

```
double result = integral.EvalIntegral(a, b, eps,
  (double x) =>
  { return Math.Sin(x) + Math.Cos(x); });
```

Результат будет тот же. Приведу теперь аналог метода `TestAnonymToDelegate`, заменив прежние определения анонимных методов лямбда-выражениями.

```
public void TestAnonymToLambda()
 {
     D1 d1 = (s, x) => { return "OK!"; };
     D1 d12 = (string s, double x) => { return s + x; };
     Console.WriteLine(d1("s", 5));
     Console.WriteLine(d12("12", 3));
     string res;
     D2 d2 = (string s, out string r) => { r = s + s; };
     d2("Hello ", out res);
     Console.WriteLine(res);
     D3 d3 = item => { return item.ToString(); };
     Console.WriteLine(d3(new Son()));
 }
```

Обратите внимание: в лямбда-выражениях можно опускать задание типов аргументов, оставив только имена аргументов. В этом случае компилятор попытается вывести их типы из контекста, задающего тело метода. Для облегчения его работы полезно явно задавать

сигнатуру метода, как это и сделано в некоторых из приведенных примеров. Стоит ли говорить, что результаты работы этого метода эквиваленты тем, что получены при старых определениях.

Анонимный метод, заданный лямбда-выражением, может также использовать контекст. Приведу аналог метода `Traverse` из класса `WithAnonymous` :

```
public void TraverseL(double min, out int N, out double res)
 {
     int n = 0;
     Walk(array, out res, x =>
     {
         if (x > min && x < max) { n++; return x; }
         else return 0;
     });
     N = n;
 }
```

Лямбда-выражения и анонимные типы данных активно используются при работе с запросами к базам данных, так что, надеюсь, к этой теме мы еще вернемся в следующих разделах этого курса. А сейчас продолжим рассмотрение случаев использования функционального типа данных.

Источники:

1. Биллиг В. Основы программирования на C#. Режим доступа: http://www.intuit.ru/studies/courses/2247/18/info
2. Биллиг В. Основы программирования на C# 3.0. Ядро языка. Режим доступа: http://www.intuit.ru/studies/courses/1094/428/info
3. Павловская Т. Программирование на ЯВУ. Делегаты и события. Режим доступа: http://www.intuit.ru/studies/courses/629/485/info