Глава 10
Делегаты, события и потоки выполнения

В этой главе рассматриваются делегаты и события — два взаимосвязанных средства языка C#, позволяющие организовать эффективное взаимодействие объектов. Во второй части главы приводятся начальные сведения о разработке многопоточных приложений.

Делегаты

Делегат — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка. Рассмотрим сначала второй случай.

Описание делегатов

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

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

Спецификаторы делегата имеют тот же смысл, что и для класса, причем допускаются только 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.

ПРИМЕЧАНИЕ

Делегат, как и всякий класс, представляет собой тип данных. Его базовым классом является класс System.Delegate, снабжающий своего «отпрыска» некоторыми полезными элементами, которые мы рассмотрим позже. Наследовать от делегата нельзя, да и нет смысла.

Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

Использование делегатов

Для того 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 динамически 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.

Все эти варианты подробно обсуждаются далее. Рассмотрим сначала пример реализации первой из этих целей. В листинге 10.1 объявляется делегат, с помощью которого один и тот же оператор используется для вызова двух разных методов (COOl и Hack).

Листинг 10.1. Простейшее использование делегата

```
using System;
namespace ConsoleApplication1
{
    delegate void Del ( ref string s );                // объявление делегата

    class Class1
    {
        public static void COOl ( ref string s )                    // метод 1
        {
            string temp = "";
            for ( int i = 0; i < s.Length; ++i )
            {
```

продолжение ⇨

Листинг 10.1 (*продолжение*)

```
            if      ( s[i] == 'o' || s[i] == 'O') temp += 'O';
            else if ( s[i] == 'l' )               temp += '1';
            else                                  temp += s[i];
        }
        s = temp;
    }

    public static void Hack ( ref string s )                    // метод 2
    {
        string temp = "";
        for ( int i = 0; i < s.Length; ++i )
            if ( i / 2 * 2 == i ) temp += char.ToUpper( s[i] );
            else                  temp += s[i];

        s = temp;
    }

    static void Main()
    {
        string s = "cool hackers";
        Del d;                                      // экземпляр делегата

        for ( int i = 0; i < 2; ++i )
        {
            d = new Del( COOl );                // инициализация методом 1
            if ( i == 1 ) d = new Del(Hack);    // инициализация методом 2

            d( ref s );     // использование делегата для вызова методов
            Console.WriteLine( s );
        }
    }
  }
}
```

Результат работы программы:

```
cOOl hackers
COOl hAcKeRs
```

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()
{
        string s = "cool hackers";
        Del d = new Del( COOl );
        d += new Del( Hack );              // добавление метода в делегат

        d( ref s );
        Console.WriteLine( s );            // результат: COOl hAcKeRs
}
```

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`.

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 гибкой, динамической связи между объектами во время выполнения программы применяется следующая стратегия. Объект, называемый *источником*, при изменении своего состояния, которое может представлять интерес для других объектов, посылает им уведомления. Эти объекты называются *наблюдателями*. Получив уведомление, наблюдатель опрашивает источник, чтобы синхронизировать с ним свое состояние.

Примером такой стратегии может служить связь объекта с различными его представлениями, например, 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 (листинг 10.2), в котором демонстрируется схема оповещения источником трех наблюдателей. Гипотетическое изменение состояния объекта моделируется сообщением «OOPS!». Один из методов в демонстрационных целях сделан статическим.

Листинг 10.2. Оповещение наблюдателей с помощью делегата

```
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( "OOPS!" );
            if ( dels != null ) dels( this );     // оповещение наблюдателей
        }
    }

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

    class ObsB                                          // класс-наблюдатель
    {
        public static void See( object o )   // реакция на событие источника
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
```

```
    }

    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. Обратите внимание: для статического метода указывается имя класса, а для обычного метода — имя объекта. При наступлении «часа X» все зарегистрированные методы поочередно вызываются через делегат.

Результат работы программы:

```
OOPS!
Вижу, что OOPS!
Вижу, что OOPS!
Я тоже вижу, что OOPS!
```

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 (то есть вызывать методы этого объекта).

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;
}
```

Операции

Делегаты можно *сравнивать на равенство и неравенство*. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке. Сравнивать можно даже делегаты различных типов при условии, что они имеют один и тот же тип возвращаемого значения и одинаковые списки параметров.

С делегатами одного типа можно *выполнять операции* простого и сложного *присваивания*, например:

```
Del d1 = new Del( o1.Do );        // o1.Do
Del d2 = new Del( o2.Do );        // o2.Do
Del d3 = d1 + d2;                 // o1.Do и o2.Do
d3 += d1;                         // o1.Do, o2.Do и o1.Do
d3 -= d2;                         // o1.Do и o1.Do
```

Эти операции могут понадобиться, например, в том случае, если в разных обстоятельствах требуется вызывать разные наборы и комбинации наборов методов.

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

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. Этот пример приводится далее.

Обратный вызов (callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра. Рассмотрим рис. 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.

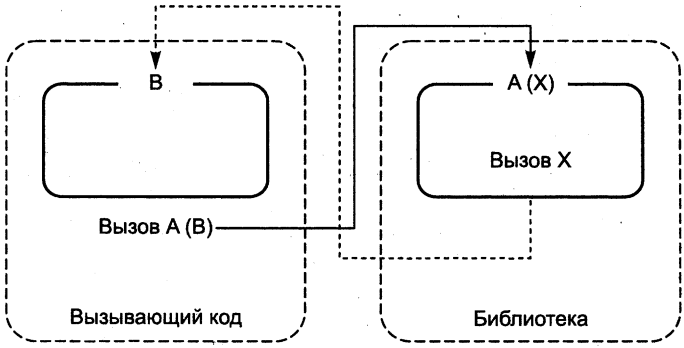

Рис. 10.1. Механизм обратного вызова

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

Пример передачи делегата в качестве параметра приведен в листинге 10.3. Программа выводит таблицу значений функции на заданном интервале с шагом, равным единице.

Листинг 10.3. Передача делегата через список параметров

```
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 " );
```

продолжение

Листинг 10.3 (*продолжение*)

```
            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 |
 ---------------------
```

В среде Visual Studio 2005, использующей версию 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. В листинге 10.4 использованы оба упрощения для реализации тех же действий, что и листинге 10.3.

Листинг 10.4. Передача делегата через список параметров (версия 2.0)

```
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( " --------------------" );
        }

        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[1]. 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.

Альтернативой использованию делегатов в качестве параметров являются виртуальные методы. Универсальный метод вывода таблицы значений функции можно реализовать с помощью абстрактного базового класса, содержащего два метода: метод вывода таблицы и абстрактный метод, задающий вид вычисляемой функции. Для вывода таблицы конкретной функции необходимо создать производный класс, переопределяющий этот абстрактный метод. Реализация метода вывода таблицы с помощью наследования и виртуальных методов приведена в листинге 10.5.

Листинг 10.5. Альтернатива параметрам-делегатам

```
using System;
namespace ConsoleApplication1
{
    abstract class TableFun
    {
        public abstract double F( double x );

        public void Table( double x, double b )
        {
            Console.WriteLine( " ----- X ----- Y -----" );
            while ( x <= b )
```

продолжение ⇨

[1] В результате в 2005 году язык C# в этой части вплотную приблизился к синтаксису старого доброго Паскаля, в котором передача функций в качестве параметров была реализована еще в 1992 году, если не раньше.

Листинг 10.5 (*продолжение*)

```
            {
                Console.WriteLine( "| {0,8:0.000} | {1,8:0.000} |", x, F(x));
                x += 1;
            }
            Console.WriteLine( " --------------------" );
        }
    }

    class SimpleFun : TableFun
    {
        public override double F( double x )
        {
            return 1;
        }
    }

    class SinFun : TableFun
    {
        public override double F( double x )
        {
            return Math.Sin(x);
        }
    }

    class Class1
    {   static void Main()
        {
            TableFun a = new SinFun();
            Console.WriteLine( " Таблица функции Sin " );
            a.Table( -2, 2 );

            a = new SimpleFun();
            Console.WriteLine( " Таблица функции Simple " );
            a.Table( 0, 3 );
        }
    }
}
```

Результат работы этой программы такой же, как и предыдущей, но, на мой взгляд, в данном случае применение делегатов предпочтительнее.

Обработка исключений при вызове делегатов

Ранее говорилось о том, что если в одном из методов списка делегата генерируется исключение, следующие методы не вызываются. Этого можно избежать, если обеспечить явный перебор всех методов в проверяемом блоке и обрабатывать возникающие исключения. Все методы, заданные в экземпляре делегата, можно

получить с помощью унаследованного метода GetInvocationList. Этот прием иллюстрирует листинг 10.6, представляющий собой измененный вариант листинга 10.1.

Листинг 10.6. Перехват исключений при вызове делегата

```
using System;
namespace ConsoleApplication1
{
    delegate void Del ( ref string s );

    class Class1
    {
        public static void COOl ( ref string s )
        {
            Console.WriteLine( "вызван метод COOl" );
            string temp = "";
            for ( int i = 0; i < s.Length; ++i )
            {
                if      ( s[i] == 'o' || s[i] == 'O') temp += '0';
                else if ( s[i] == 'l' )               temp += '1';
                else                                  temp += s[i];
            }
            s = temp;
        }

        public static void Hack ( ref string s )
        {
            Console.WriteLine( "вызван метод Hack" );
            string temp = "";
            for ( int i = 0; i < s.Length; ++i )
                if ( i / 2 * 2 == i ) temp += char.ToUpper( s[i] );
                else                  temp += s[i];

            s = temp;
        }

        public static void BadHack ( ref string s )
        {
            Console.WriteLine( "вызван метод BadHack" );
            throw new Exception();                        // имитация ошибки
        }

        static void Main()
        {
            string s = "cool hackers";
            Del d = new Del( COOl );          // создание экземпляра делегата
            d += new Del( BadHack );          // дополнение списка методов
            d += new Del( Hack );             // дополнение списка методов
```

продолжение ⇨

Листинг 10.6 (*продолжение*)

```
            foreach ( Del fun in d.GetInvocationList() )
            {
                try
                {
                    fun( ref s );          // вызов каждого метода из списка
                }
                catch ( Exception e )
                {
                    Console.WriteLine( e.Message );
                    Console.WriteLine( "Exception in method " +
                                       fun.Method.Name);
                }
            }
            Console.WriteLine( "результат - " + s );
        }
    }
}
```

Результат работы программы:

```
вызван метод C001
вызван метод BadHack
Exception of type System.Exception was thrown.
Exception in method BadHack
вызван метод Hack
результат - C001 hAcKeRs
```

В этой программе помимо метода базового класса `GetInvocationList` использовано свойство `Method`. Это свойство возвращает результат типа `MethodInfo`. Класс `MethodInfo` содержит множество свойств и методов, позволяющих получить полную информацию о методе, например его спецификаторы доступа, имя и тип возвращаемого значения. Мы рассмотрим этот интересный класс в разделе «Рефлексия» главы 12.

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[1]:

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

Для событий применяются *спецификаторы* new, public, protected, internal, private, static, virtual, sealed, override, abstract и extern, которые изучались при рассмотрении методов классов. 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.

ПРИМЕЧАНИЕ

Событие — это удобная абстракция для программиста. На самом деле оно состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.

В листинге 10.7 приведен код из листинга 10.2, переработанный с использованием событий.

[1] Приводится упрощенный вариант. В общем случае имеется возможность задавать несколько имен событий, инициализаторы и методы добавления и удаления событий.

Листинг 10.7. Оповещение наблюдателей с помощью событий

```
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( "OOPS!" );
            if ( Oops != null ) Oops();
        }
    }

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

    class ObsB                                           // класс-наблюдатель
    {
        public static void See()                // реакция на событие источника
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
    }

    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. Иного способа доступа к списку обработчиков нет.

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. Например, в методе CryOops выполняется проверка на null для того, чтобы избежать генерации исключения System.NullReferenceException.

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 не описывать делегата и собственный тип аргументов, а обойтись стандартным классом делегата System.EventHandler.

Имя обработчика события принято составлять из префикса On и имени события. В листинге 10.8 приведен пример из листинга 10.7, оформленный в соответствии со стандартными соглашениями .NET. Найдите восемь отличий!

Листинг 10.8. Использование стандартного делегата EventHandler

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

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

    class ObsA
    {
```

продолжение ⇨

Листинг 10.8 (*продолжение*)

```
        public void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Вижу, что OOPS!" );
        }
    }

    class ObsB
    {
        public static void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
    }

    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();
        }
    }
}
```

Те, кто работает с C# версии 2.0, могут упростить эту программу, используя новую возможность неявного создания делегатов при регистрации обработчиков событий. Соответствующий вариант приведен в листинге 10.9. В демонстрационных целях в код добавлен новый *анонимный обработчик* — еще один механизм, появившийся в новой версии языка.

Листинг 10.9. Использование делегатов и анонимных методов (версия 2.0)

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

        public void CryOops()
        {
            Console.WriteLine( "OOPS!" );
```

```
            if ( Oops != null ) Oops( this, null );
        }
    }

    class ObsA
    {
        public void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Вижу, что OOPS!" );
        }
    }

    class ObsB
    {
        public static void OnOops( object sender, EventArgs e )
        {
            Console.WriteLine( "Я тоже вижу, что OOPS!" );
        }
    }

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

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

            s.Oops += o1.OnOops;
            s.Oops += o2.OnOops;
            s.Oops += ObsB.OnOops;
            s.Oops += delegate ( object sender, EventArgs e )
                      { Console.WriteLine( "Я с вами!" ); };

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

События включены во многие стандартные классы .NET, например, в классы пространства имен `Windows.Forms`, используемые для разработки Windows-приложений. Мы рассмотрим эти классы в главе 14.

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[1]) представляет собой часть исполняемого кода программы. 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 их переключением, что снижает общую производительность системы;
- в многопоточных приложениях 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).

Класс Thread

Поддержка многопоточности осуществляется в .NET в основном с помощью пространства имен `System.Threading`. Некоторые типы этого пространства описаны в табл. 10.1.

Таблица 10.1. Некоторые типы пространства имен System.Threading

Тип	Описание
`Interlocked`	Класс, обеспечивающий синхронизированный доступ к переменным, которые используются в разных потоках
`Monitor`	Класс, обеспечивающий синхронизацию доступа к объектам
`Mutex`	Класс-примитив синхронизации, который используется также для синхронизации между процессами
`ReaderWriterLock`	Класс, определяющий блокировку, поддерживающую один доступ на запись и несколько — на чтение
`Thread`	Класс, который создает поток, устанавливает его приоритет, получает информацию о состоянии

[1] Иногда этот термин переводится буквально — «нить», чтобы отличить его от потоков ввода-вывода, которые рассматриваются в следующей главе. Поэтому в литературе можно встретить и термин «многонитевые приложения».

Тип	Описание
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
IOCompletionCallback	Класс, получающий сведения о завершившейся операции ввода-вывода
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. В листинге 10.10 приведен пример одновременной работы двух потоков.

Листинг 10.10. Создание вторичного потока

```
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.WriteLine( i ); Thread.Sleep( 1000 );
            }
        }

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

продолжение ☞

Листинг 10.10 (*продолжение*)

```
            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 табл. 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)
IsAlive	Свойство	Возвращает true или false в зависимости от того, запущен поток или нет
IsBackground	Свойство	Возвращает или устанавливает значение, которое показывает, является ли этот поток фоновым
Name	Свойство	Установка текстового имени потока
Priority	Свойство	Получить/установить приоритет потока (используются значения перечисления ThreadPrority)
ThreadState	Свойство	Возвращает состояние потока (используются значения перечисления ThreadState)
Abort	Метод	Генерирует исключение ThreadAbortException. Вызов этого метода обычно завершает работу потока
GetData, SetData	Статические методы	Возвращает (устанавливает) значение для указанного слота в текущем потоке
GetDomain, GetDomainID	Статические методы	Возвращает ссылку на домен приложения (идентификатор домена приложения), в рамках которого работает поток

Элемент	Вид	Описание
GetHashCode	Метод	Возвращает хеш-код для 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. Пример приведен в листинге 10.11.

Листинг 10.11. Потоки, использующие один объект

```
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
```

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.

В листинге 10.11 приводятся 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.

Листинг 10.11. Асинхронные делегаты

```
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 делегата 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 );
```

продолжение ⇨

Листинг 10.11 (*продолжение*)

```
                                                    // 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.

Рекомендации по программированию

Делегаты широко применяются в библиотеке .NET как самостоятельно, так и для поддержки механизма *событий*, который имеет важнейшее значение при программировании под Windows.

Делегат представляет собой особый вид класса, несколько напоминающий интерфейс, но, в отличие от него, задающий только одну сигнатуру метода. В языке C++ аналогом делегата является указатель на функцию, но он не обладает безопасностью и удобством использования делегата. Благодаря делегатам становится возможной гибкая организация взаимодействия, позволяющая поддерживать согласованное состояние взаимосвязанных объектов.

Начиная с версии 2.0, в C# поддерживаются возможности, упрощающие процесс программирования с применением делегатов — неявное создание делегатов при регистрации обработчиков событий и анонимные обработчики.

Основной целью создания *многопоточных приложений* является повышение общей производительности программы. Однако разработка многопоточных приложений сложнее, поскольку при этом возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков.