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 += '0';
            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:

```
cOO1 hackers
COO1 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.