

I segreti dei delegate



I delegate giocano un ruolo fondamentale nell'intera infrastruttura .NET. Conoscerli vi aiuterà a realizzare applicazioni più potenti e a sfruttare meglio le librerie del .NET Framework

di Alberto Falossi

La maggior parte delle applicazioni .NET utilizza – esplicitamente o implicitamente – i delegate. Gli eventi stessi, che sono uno tra i più importanti meccanismi del framework, si servono internamente dei delegate. I programmatori che vogliono sfruttare al massimo le funzionalità del framework Microsoft devono conoscere approfonditamente i delegate. Accanto alle funzionalità più note, come l'invocazione sincrona e asincrona dei metodi, ci sono alcune caratteristiche – in alcuni casi praticamente non documentate – che i programmatori ignorano, ma non per questo hanno minore importanza.

A cosa servono i delegate

Nel mondo reale, i delegati sono da sempre coloro che svolgono un'attività per conto di qualcun'altro. Bene, i delegati .NET hanno molte analogie con i delegati reali. Nel .NET Framework un *delegate* indica la posizione in memoria di un metodo (detto *metodo target*), e permette di richiamarlo indirettamente anche senza conoscerne il nome: in altre parole, invece di chiamare direttamente un metodo, potete incaricare un delegate di svolgere la chiamata per voi (**Figura 1**). Chi ha dimestichezza col C/C++ può considerare i delegate l'evoluzione object oriented dei puntatori a funzione (vedi **Riquadro 1**).

Vediamo uno scenario in cui i delegate si rivelano di estrema utilità. Il **Listato 1** mostra una procedura *FindFiles* che ricerca uno o più file nell'hard disk e stampa il nome a video. Suppo-

niamo di voler inserire i nomi in una listbox invece di stamparli nella console: basterà sostituire una linea, ma sarete costretti ad intaccare l'algoritmo di ricerca e ricompilare l'eventuale libreria che lo contiene.

Per migliorare il codice potreste aumentare il *grado di coesione*, cioè isolare l'algoritmo di ricerca dal codice che fa uso dei risultati: se la routine restituisce un array di stringhe con i nomi dei file trovati sarebbe sufficiente elaborare in maniera differente l'array ricevuto come risultato. Tuttavia sareste costretti ad aspettare la fine della ricerca prima di poter stampare i nomi, mentre *FindFiles* mostra i risultati in tempo reale. Grazie ad un delegate potete avere il meglio delle due soluzioni (visualizzazione in tempo reale e alto grado di coesione del codice). Il **Listato 2** mostra come fare: la procedura *FindFiles2* accetta come terzo parametro un oggetto delegate, la cui creazione ed impostazione spetta all'applicazione chiamante.

Ogni volta che viene trovato un file, la routine si limita a passare il nome del file al delegate e dare il via libera per la chiamata, ma non sa niente del metodo puntato; il delegate, da parte sua, richiamerà il metodo target e passerà il nome del file (**Figura 1**).

Così facendo, potete personalizzare il comportamento di *FindFiles2* semplicemente passando un oggetto delegate differente:

Alberto Falossi è esperto di programmazione Windows e .NET. Lavora per Code Architects (www.codearchitects.com), dove svolge attività di consulenza e formazione nell'ambito di .NET e tecnologie correlate. Si occupa del coordinamento editoriale di VBJ e collabora con numerose riviste di programmazione italiane e straniere. È membro del team di VB2TheMax (www.vb2themax.com). Può essere contattato tramite e-mail all'indirizzo afalossi@infomedia.it

C#

```
// stampa i file nella console window
FileFoundDelegate d = new FileFoundDelegate(Console.WriteLine);
FindFiles2("C:\\", "*.txt", d);

// stampa i file nella output window
d = new FileFoundDelegate(System.Diagnostics.Debug.WriteLine);
FindFiles2("C:\\", "*.txt", d);
```

Se inserita in una libreria, la routine potrà essere richiamata da applicazioni differenti ed ognuna potrà personalizzarne il comportamento.

Qualcuno potrà osservare che una soluzione simile si può ottenere definendo il terzo parametro di FindFiles2 come interfaccia e passando ad esso un opportuno oggetto che la implementa: in effetti, ci sono alcune analogie tra la programmazione per delegate e programmazione per interfacce.

Ce ne occuperemo nel prossimo numero, quando tratteremo in dettaglio gli eventi e saremo dunque in grado

Listato 1 Una routine per la ricerca ricorsiva dei file

```
[C#]

using System;
using System.IO;

namespace DelegateTest
{
    class Class1
    {
        [STAThread]
        static void Main()
        {
            // cerca i file di testo in C:\
            FindFiles("C:\\", "*.txt");
        }

        static void FindFiles(string path, string fileName)
        {
            foreach (string file
                in Directory.GetFiles(path, fileName))
            {
                // stampa il nome di ogni file trovato
                Console.WriteLine(file);
            }

            foreach (string subdir in
                Directory.GetDirectories(path))
            {
                // ricerca ricorsivamente nelle sottodirectory
                FindFiles(subdir, fileName);
            }
        }
    }
}
```

di confrontare le differenze nell'utilizzo delegate, eventi ed interfacce, dando dei consigli per scegliere la giusta tecnica nelle proprie applicazioni.

Come utilizzare i delegate

Vediamo un po' più in dettaglio come utilizzare i delegate. Un delegate deve essere prima dichiarato e poi istanziato.

C#

```
// dichiarazione
delegate void SampleDelegate(string s);
```

VB .NET

```
' dichiarazione
Delegate Sub SampleDelegate(ByVal s As String)
```

Listato 2

La routine di ricerca può essere personalizzata grazie ad un delegate

```
[C#]

using System;
using System.IO;

namespace DelegateTest
{
    class Class1
    {
        // dichiarazione del delegate
        delegate void FileFoundDelegate(string fileName);

        [STAThread]
        static void Main()
        {
            // stampa i file nella console window
            FileFoundDelegate d = new FileFoundDelegat
                (Console.WriteLine);
            FindFiles2("C:\\", "*.txt", d);
        }

        static void FindFiles2(string path,
            string fileName, FileFoundDelegate d)
        {
            foreach (string file in Directory.GetFiles(path,
                fileName))
            {
                // per ogni file trovato invoca il metodo
                // del delegate
                d(file);
            }

            foreach (string subdir in
                Directory.GetDirectories(path))
            {
                // ricerca ricorsivamente nelle sottodirectory
                FindFiles2(subdir, fileName, d);
            }
        }
    }
}
```

La dichiarazione dei delegate è una via di mezzo tra quella di un metodo e quella di una variabile: dovete specificare la *signature* del metodo target, cioè i parametri di ingresso e il valore di ritorno. In questo caso, le istanze di `SampleDelegate` potranno richiamare qualsiasi metodo che accetta una stringa, e non ha valore di ritorno. Ecco un altro esempio più complesso:

C#

```
// parametri: intero, stringa e array di oggetti
// valore di ritorno: array di stringhe
delegate string[] SampleDelegate2(int i, string s, object[] o);
```

VB .NET

```
Delegate Function SampleDelegate2(ByVal i As Integer,
    ByVal s As String, ByVal o As Object()) As String()
```

Listato 3 Esempi di utilizzo di delegate

```
[C#]
using System;
using System.IO;

namespace DelegateTest
{
    // dichiarazione del delegate
    delegate void SampleDelegate(string s);

    class Class1
    {
        [STAThread]
        static void Main()
        {
            // utilizzo di delegate con metodo statico
            SampleDelegate d1 = new
                SampleDelegate(StaticMethod);
            d1("Hello World");

            // utilizzo di delegate con metodo d'istanza
            // di classe
            Class1 c = new Class1();
            SampleDelegate d2 = new
                SampleDelegate(c.InstanceMethod);
            d2("Hello World");
        }

        // metodo d'istanza
        public void InstanceMethod(string s)
        {
            Console.WriteLine(s);
        }

        // metodo statico
        static void StaticMethod(string s)
        {
            Console.WriteLine(s);
        }
    }
}
```

Una volta dichiarato, il delegate è pronto per essere istanziato (come vedremo tra poco un delegate è una classe) ed utilizzato. Al momento dell'istanziamento di un delegate, il programmatore deve indicare il metodo target con la seguente sintassi:

C#

```
SampleDelegate d = new SampleDelegate(SampleMethod);
```

VB .NET

```
Dim d As New SampleDelegate(AddressOf SampleMethod)
```

Le precedenti istruzioni creano un oggetto `SampleDelegate` in grado di richiamare il metodo `SampleMethod`. Se `SampleMethod` non rispetta la signature del delegate, otterrete un errore a tempo di compilazione.

VB .NET permette di utilizzare una sintassi ancora più comoda:

VB .NET

```
Dim d As SampleDelegate = AddressOf SampleMethod
```

Le due istruzioni di VB .NET sono equivalenti: la seconda è forse più semplice, ma è meno comprensibile da parte di un programmatore non VB.

Riquadro 1 Confronto tra i delegate e i puntatori a funzione C/C++

In C/C++ un puntatore a funzione è semplicemente un indirizzo in memoria. Tale indirizzo non permette di sapere il nome del metodo puntato o i parametri da passare: assicurarsi che la chiamata al metodo sia effettuata correttamente è un onere che spetta al programmatore (come se non avesse nessun altro problema...) Ecco la causa di molti bug dei programmi C/C++ unmanaged: se il programmatore – volontariamente o involontariamente – accede ad un indirizzo non valido o passa i parametri sbagliati al metodo (come un intero al posto di una stringa), causerà una violazione di accesso alla memoria, con conseguente crash a runtime dell'applicazione.

I delegate .NET conservano la potenza dei puntatori ma sono sicuri da utilizzare. Un delegate è *type safe*, cioè può accedere solo al metodo indicato dal programmatore, e obbliga a passare i parametri corretti, pena un errore in fase di compilazione.

Inoltre un delegate è una vera e propria classe, e porta con sé le informazioni sul metodo puntato; ad esempio, potete accedere in sola lettura all'oggetto che espone il metodo target con la proprietà `Target`:

```
C#
// Target è null (Nothing in VB) se
// il metodo puntato è statico.
object o = d.Target;
```

Per ulteriori informazioni sui membri dei delegate consultare la documentazione online del .NET Framework SDK.

D'ora in avanti potrete richiamare `SampleMethod` tramite `SampleDelegate` con l'istruzione:

```
C#
d("Hello World");
```

```
VB .NET
d("Hello World")
```

A prima vista può sembrare che `d` sia il nome di un metodo, mentre è una variabile delegate. Logicamente, è come se al posto della precedente istruzione ci sia:

```
C#
SampleMethod("Hello World");
```

```
VB .NET
SampleMethod("Hello World")
```

Un'eccezione lanciata dal metodo target si propaga al codice che utilizza il delegate; per catturarla basta ricorrere al solito `try...catch`.

Una notevole caratteristica dei delegate è il polimorfismo: un delegate può essere riutilizzato per richiamare qualsiasi metodo con la signature del delegate.

`SampleDelegate` è adatto a tutti i metodi che accettano una stringa (infatti lo abbiamo utilizzato con `Console.WriteLine` e `Debug.WriteLine`). Il **Listato 3** mostra altri casi di impiego di `SampleDelegate`: notate che un delegate può richiamare sia metodi statici che metodi d'istanza.

Performance

È interessante sapere che l'overhead aggiunto dai delegate è praticamente nullo, perché la chiamata al metodo avviene in *early binding* (il metodo è noto a tempo di compilazio-

Listato 4 Struttura interna di un delegate

```
[C#]

public class SampleDelegate : System.MulticastDelegate
{
    // costruttore
    public SampleDelegate(object target,
                          int methodID) { ... }

    // invocazione sincrona
    public virtual void Invoke(string s) { ... }

    // invocazione asincrona
    public virtual IAsyncResult BeginInvoke(string s,
                                             AsyncCallback callback, object obj) { ... }
    public virtual void EndInvoke(IAsyncResult
                                  result) { ... }
}
```

ne). Dai miei test risulta che la chiamata via delegate è tre volte più lenta di quella diretta, ma per apprezzare la differenza ho dovuto richiamare un metodo 40 milioni di volte, e – nonostante ciò – la differenza non superava i due decimi di secondo! In definitiva, nella maggior parte delle applicazioni non noterete degrado di prestazioni, anche nel caso di molte chiamate.

Struttura interna di un delegate

Un delegate è in realtà una classe che eredita da `System.MulticastDelegate`. È per questo che può essere dichiarato solo dove può essere dichiarata una classe (vedi **Listato 2** e **Listato 3**): se provate a definire un

Listato 5 Invocazione asincrona del metodo target

```
[C#]

using System;
using System.Threading;

namespace DelegateTest
{
    // dichiarazione del delegate
    delegate void SampleDelegate(string s);

    class Class1
    {
        [STAThread]
        static void Main()
        {
            SampleDelegate d = new
            SampleDelegate(StaticMethod);

            // invoca asincronamente d
            IAsyncResult result = d.BeginInvoke("Hello
            World", null, null);

            // stampa l'hash code del thread corrente
            // nota: l'hash code identifica univocamente un
            thread
            Console.WriteLine("Main thread hash code: {0}",
            Thread.CurrentThread.GetHashCode());

            // si assicura che la chiamata sia terminata
            (opzionale in altri casi)
            d.EndInvoke(result);
        }

        // metodo target
        static void StaticMethod(string s)
        {
            // stampa l'hash code del thread corrente
            Console.WriteLine("StaticMethod thread hash code:
            {0}", Thread.CurrentThread.GetHashCode());

            Console.WriteLine(s);
        }
    }
}
```

delegate all'interno di un metodo, otterrete un errore di compilazione.

C# e VB .NET si avvalgono di una sintassi semplificata per le istruzioni che dichiarano i delegate, ma durante la compilazione tali istruzioni subiscono una trasformazione.

Listato 6 Un metodo richiamato con BeginInvoke è eseguito in un thread diverso da quello dell'applicazione; per modificare controlli e form bisogna isolare il codice di aggiornamento in una routine (UpdateControls) e richiamare quest'ultima con il metodo Invoke delle form o dei controlli (da non confondere con Invoke dei delegate)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace DelegateTest
{
    public class Form1 : System.Windows.Forms.Form
    {
        // codice di inizializzazione (omesso,
        // vedi progetto allegato)

        // dichiarazione del delegate
        delegate void SampleDelegate();

        private void button1_Click(object sender,
            System.EventArgs e)
        {
            lblStatus.Text = "Invoking...";
            Refresh();

            // 1 - crea e utilizza il delegate
            SampleDelegate d = new SampleDelegate(DoSomething);
            IAsyncResult result = d.BeginInvoke(null, null);
        }

        private void DoSomething()
        {
            // 2 - fa qualcosa
            // :
            System.Threading.Thread.Sleep(2000);

            // 3- non aggiorna i controlli direttamente;
            // utilizza una helper
            // routine separata e la invoca con il
            // metodo Control.Invoke()
            this.Invoke(new MethodInvoker(UpdateControls));
        }

        // 4- helper routine che aggiorna i controlli
        private void UpdateControls()
        {
            lblStatus.Text = "Done";
        }
    }
}
```

Il delegate SampleDelegate del precedente paragrafo è tradotto nella classe del **Listato 4**.

I due parametri del costruttore indicano l'istanza dell'oggetto che espone il metodo target e un ID usato internamente dal runtime per identificare il metodo (ricavato con l'istruzione IL *ldftrn*). Se ci fate caso, voi passate solo un nome di metodo:

C#

```
// creazione di delegate via costruttore
SampleDelegate d = new SampleDelegate(SampleMethod);
```

VB .NET

```
' creazione di delegate via costruttore
Dim d As New SampleDelegate(AddressOf SampleMethod)
```

Durante la compilazione, i compilatori modificheranno le precedenti istruzioni in modo da passare al costruttore i giusti parametri. È qui che avviene l'early binding: grazie ai due parametri, il metodo target può essere determinato a tempo di compilazione.

Il metodo Invoke del **Listato 4** si occupa di invocare il metodo. Infatti, ogni istruzione

C#

```
d("Hello World");
```

VB .NET

```
d("Hello World")
```

è tradotta dal compilatore in

C#

```
d.Invoke("Hello World");
```

VB .NET

```
d.Invoke("Hello World")
```

Curiosamente, VB .NET permette di richiamare Invoke esplicitamente anche da codice, mentre C# no. Notate che

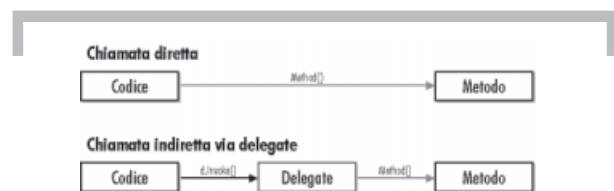


Figura 1 Nella normale chiamata diretta ad un metodo, il codice accede direttamente alla locazione in memoria del metodo. Un delegate permette di richiamare un metodo indirettamente, cioè senza conoscerne la locazione in memoria: il codice "autorizza" il delegate ad effettuare il delegate tramite Invoke e il delegate effettua la chiamata diretta vera e propria

Listato 7 Invocazione in late binding del metodo target

```
[C#]

using System;
using System.Reflection;

namespace DelegateTest
{
    // dichiarazione dei delegate
    delegate void SampleDelegate1(string s);
    delegate int SampleDelegate2();

    class Class1
    {
        [STAThread]
        static void Main()
        {
            string delegateType, targetType, methodName = null;
            object[] args = null;

            // cambiare in false per eseguire l'altro metodo
            if (false)
            {
                // imposta tipo del delegate,
                // metodo target e argomenti
                delegateType = "DelegateTest.SampleDelegate1";
                targetType = "DelegateTest.Class1";
                methodName = "SampleMethod1";
                args = new object[1];
                args[0] = "Hello world";
            }
            else
            {
                // imposta tipo del delegate,
                // metodo target e argomenti
                delegateType = "DelegateTest.SampleDelegate2";
                targetType = "DelegateTest.Class1";
                methodName = "SampleMethod2";
            }

            // crea il delegate dinamicamente
            Delegate d =
                Delegate.CreateDelegate(Type.GetType(delegateType),
                    Type.GetType(targetType), methodName);

            // invoca il metodo target
            object result = d.DynamicInvoke(args);

            if (result != null) Console.WriteLine("Result:
{0}", result);
        }

        public static void SampleMethod1(string s)
        {
            Console.WriteLine("SampleMethod1");
        }

        public static int SampleMethod2()
        {
            Console.WriteLine("SampleMethod2");
            return 123;
        }
    }
}
```

la signature di Invoke è di volta in volta modellata dal compilatore in modo da essere uguale a quella del metodo puntato. È questo il "segreto" della robustezza dei delegate: non è possibile passare parametri diversi da quelli richiesti, perché Invoke li rifiuterebbe.

Invocare asincronamente un metodo

Una straordinaria caratteristica dei delegate è di permettere l'invocazione di un metodo sia in modalità sincrona che asincrona. Gli esempi mostrati fino ad ora effettuavano una chiamata sincrona, tramite Invoke. I metodi *BeginInvoke* e *EndInvoke* del **Listato 4** si occupano della chiamata asincrona. Notate che, come Invoke, anche *BeginInvoke* è modellato di volta in volta sulla signature del metodo target (a parte i due argomenti finali, che sono sempre presenti e servono per controllare l'esecuzione asincrona).

Grazie a *BeginInvoke* potete invocare un metodo e non aspettare la sua terminazione (**Listato 5**); il metodo *StaticMethod* è eseguito asincronamente in un altro thread. L'utilizzo di *BeginInvoke* e *EndInvoke* è ben documentato nell'SDK. Quello che mi preme farvi notare sono invece alcuni dettagli che non ritengo altrettanto ben enfatizzati. Quando si invoca *BeginInvoke* il metodo è eseguito in un thread *diverso* da quello dell'applicazione. Il delegate si serve di un thread preso in prestito da un thread pool interno del CLR. Se i due thread accedono a variabili condivise dovete sincronizzare l'accesso concorrente (ad esempio con *lock* in C# e *SyncLock* in VB .NET). Un ulteriore problema si ha con le applicazioni Windows Forms. Un metodo eseguito da *BeginInvoke* non può modificare i controlli o le form, perché form e controlli possono essere modificati solo dal thread che li ha creati. Per risolvere il problema dovete usare il metodo *Invoke* dei controlli (da non confondere con *Invoke* dei delegate): la tecnica è illustrata schematicamente nel **Listato 6**, ma potete trovare ulteriori dettagli in [2].

Late binding

In alcuni casi può essere necessario invocare un metodo in late binding, cioè determinando a runtime il tipo del delegate e il nome dell'oggetto target. Per questo bisogna creare dina-

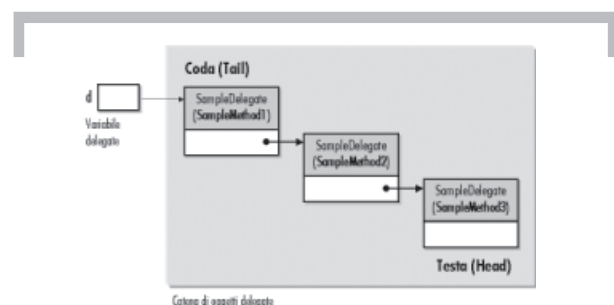


Figura 2 I delegate possono essere combinati tra loro per richiamare più metodi uno dopo l'altro; in questo caso la catena è quella definita nel **Listato 8**

micamente il delegate con *Delegate.CreateDelegate* e richiamare il metodo con *DynamicInvoke*. Il **Listato 7** mostra come fare: i delegate devono comunque essere dichiarati, ma è possibile scegliere quale utilizzare semplicemente cambiando la stringa *delegateType*. Stesso discorso per l'oggetto e il metodo target. Un altro caso in cui può essere utile Dynami-

Invoke è una routine che accetta come parametro una generica variabile *Delegate*: dato che Invoke è disponibile solo nelle classi derivate da *System.MulticastDelegate*, l'unico modo per richiamare il metodo target è attraverso Invoke.

DynamicInvoke accetta un array di oggetti con i parametri del metodo da richiamare e restituisce un oggetto col valore di ritorno. Nel caso della routine generica citata qualche riga fa, tale array dovrà essere passato assieme alla variabile *Delegate* nei parametri.

DynamicInvoke è circa 10-20 volte più lento di *Invoke*. Ricordo che se volete semplicemente creare un oggetto dinamicamente (come in VB6 con *CreateObject*) e invocare un metodo in late binding senza delegate, potete usare *Reflection*:

Listato 8 Definizione e utilizzo di una catena di delegate

```
[C#]
using System;

namespace DelegateTest
{
    // dichiarazione del delegate
    delegate void SampleDelegate();

    class Class1
    {
        [STAThread]
        static void Main()
        {
            // crea tre oggetti delegate
            SampleDelegate d1 = new
                SampleDelegate(SampleMethod1);
            SampleDelegate d2 = new
                SampleDelegate(SampleMethod2);
            SampleDelegate d3 = new
                SampleDelegate(SampleMethod3);

            // concatena gli oggetti delegate
            SampleDelegate d = (SampleDelegate)
                Delegate.Combine(d1, d2);
            d = (SampleDelegate) Delegate.Combine(d, d3);

            d();

            // visualizza:

            // SampleMethod1
            // SampleMethod2
            // SampleMethod3
        }

        public static void SampleMethod1()
        {
            Console.WriteLine("SampleMethod1");
        }

        public static void SampleMethod2()
        {
            Console.WriteLine("SampleMethod2");
        }

        public static void SampleMethod3()
        {
            Console.WriteLine("SampleMethod3");
        }
    }
}
```

C#

```
Type t = Type.GetType("DelegateTest2.Class3");
MethodInfo mi = t.GetMethod("SampleMethod");
object result = mi.Invoke(null, args);
```

L'invocazione tramite *Reflection* richiede metà tempo rispetto a *DynamicInvoke*.

Invocare più metodi con un solo delegate

Gli oggetti delegate possono essere combinati assieme; in questo modo potrete richiamare più metodi in un colpo solo (**Listato 8**). I metodi sono eseguiti uno dopo l'altro, non contemporaneamente. Le liste di delegate non possono essere invocate asincronamente con *BeginInvoke*.

Per concatenare più delegate si usa il metodo statico *Delegate.Combine*:

C#

```
d = (FileFoundDelegate) Delegate.Combine(d1, d2);
```

La variabile *d* del **Listato 8** riferisce una lista di tre delegate (**Figura 2**). Richiamando *d.Invoke*, sarà richiamato *Invoke* su tutti e tre gli oggetti delegate. Per darvi un'idea di quanto possa essere utile tale caratteristica, basti pensare che gli eventi .NET sono basati proprio su questa tecnica: tutti gli event handler che vogliono ricevere un evento sono memorizzati in una catena, e al momento della notifica viene effettuata una chiamata a *Invoke*. L'ordine di invocazione è dall'ultimo al primo, dalla testa (*head*) alla coda (*tail*). Il valore di ritorno, se presente, è quello della coda, mentre gli altri vengono scartati. Tale scelta è da ricercare nell'implementazione (non documentata) di *Invoke*:

Psuedo-C#

```
Public virtual int Invoke(string s)
```

```
{
    if (hai un predecessore)
predecessore.Invoke(s);
```

```
return MetodoTarget(s);
```

```
}
```

Listato 9 Personalizzazione del comportamento standard di Invoke

```
[C#]
using System;

namespace DelegateTest
{
    // dichiarazione del delegate
    delegate int SampleDelegate();

    class Class1
    {
        [STAThread]
        static void Main()
        {
            int total = 0;

            // crea una lista di tre oggetti delegate (uguali)
            SampleDelegate d1 = new
                SampleDelegate(SampleMethod);
            SampleDelegate d2 = new
                SampleDelegate(SampleMethod);
            SampleDelegate d3 = new
                SampleDelegate(SampleMethod);
            SampleDelegate list = d1 + d2 + d3;

            // ricava la lista
            Delegate[] arr = list.GetInvocationList();

            // scorre la lista in ordine inverso
            for (int i = arr.GetUpperBound(0);
                i >= arr.GetLowerBound(0); i--)
            {
                try
                {
                    // invoca il delegate
                    SampleDelegate del = (SampleDelegate) arr[i];
                    int retValue = del();

                    // somma i valori di ritorno
                    total += retValue;
                }
                catch
                {
                    // gestione eccezioni
                }
            }

            Console.WriteLine("Total: {0}", total);
        }

        public static int SampleMethod()
        {
            // genera un numero tra 0 e 30
            int retValue = (new System.Random()).Next(0, 30);

            System.Diagnostics.Debug.WriteLine(retValue);

            return retValue;
        }
    }
}
```

Se durante l'esecuzione di Invoke di un delegate della lista viene generata un'eccezione, i predecessori del delegate della catena non saranno chiamati e l'eccezione si propagerà fino a chi ha invocato Invoke per primo.

Ogni volta che si invoca Combine, il delegate viene aggiunto in testa. Per rimuovere un delegate dalla lista si usa Remove:

C#

```
d = (FileFoundDelegate) Delegate.Remove(d, d2);
```

C# ridefinisce gli operatori +, -, += e -= in modo che richiamino Combine e Remove:

C#

```
// crea una lista di due delegate
```

```
d = d1 + d2;
```

```
// rimuove d1 dalla catena d
```

```
d3 = d - d1;
```

```
// aggiunge d1 a d
```

```
d += d1;
```

```
// rimuove d1 da d
```

```
d -= d1;
```

Personalizzare Invoke

In alcuni casi, potreste aver bisogno di modificare il comportamento standard di Invoke. Ad esempio, potreste aver bisogno di ottenere il valore di ritorno di tutti i delegate, non solo di quello in coda. Oppure potreste aver bisogno di richiamare i delegate in un ordine diverso da quello standard. Per fare questo, dovete acquisire il diretto controllo sulla lista di delegate, tramite il metodo *GetInvocationList*. Il **Listato 9** mostra come richiamare i delegate in ordine inverso e sommare i valori di ritorno.

Conclusioni

Dopo aver esaminato tutte le caratteristiche, è chiaro che dietro un'apparente semplicità di utilizzo i delegate sono strumenti estremamente potenti. Aver capito la struttura e funzionamento vi permetterà di sfruttarli al 100% nelle applicazioni. Nel prossimo articolo ci occuperemo degli eventi, mostrando la loro struttura interna e la relazione con i delegate e spiegando come personalizzarne il comportamento standard.

Bibliografia

[1] Richter, Balena - *Applied .NET Framework Programming in VB .NET*, Microsoft Press, 2002

Riferimenti

[2] Chris Sells - *Safe, Simple, Multithreading in Windows Forms*, MSDN Online, <http://msdn.microsoft.com/library/en-us/dnforms/html/winforms06112002.asp>