

Capitolo 12

I delegati e gli eventi

Dovendo descrivere i delegati probabilmente molto vi risponderanno “sono molto simili ad un puntatore a funzione in C++”. Questo, andando anche più indietro nel tempo nell’ambito della programmazione, ci porta al concetto di callback ovvero la capacità di una funzione di richiamare un’altra, una tecnica molto usata in ambito Windows per la gestione asincrona dei processi, ad esempio. L’accostamento è certamente corretto ma è altrettanto vero che si può vivere su questo pianeta senza sapere che diavolo sia un puntatore a funzione. Mi piace di più la definizione che ad esempio si trova sul sito di MSDN e che afferma che un delegato altro non è che un tipo che viene associato ad un metodo e si comporta esattamente come esso, in termini di parametrizzazione e valori di ritorno e può essere utilizzato come qualsiasi altro metodo. Proprio **nome, valore di ritorno e argomenti (che devono apparire con lo stesso ordine oltre che avere lo stesso tipo)**, evidentemente del metodo correlato, sono le 3 grandezze che bisogna rispettare per definire un delegato e in pratica definiscono un protocollo al quale le istanze del delegato come vedremo si conformano. Ancora, più che con una definizione si può pensare di descrivere un delegato con una similitudine ovvero vedendolo come un qualcosa che attribuisce un nome alla firma di un metodo; come vedremo quest’ultimo concetto è abbastanza astuto (è una alternativa che ho trovato su un testo anni fa, mi spiace non ricordare quale). E’ importante annotare e ricordare la correlazione tra delegati e metodi. E’ intuitivo che un delegato “maschera” il metodo o i metodi che richiama. Comunque sia, anche se lo scopo è quello di fare le veci di un puntatore a funzione i delegati di C# (e più in generale di .Net) sono object oriented, sicuri e type-safe, il che è un vantaggio non da poco. Infatti è estremamente vantaggioso che sia il CLR ad occuparsi dei delegati in quanto ciò implica in pratica che essi, paragonandoli ai puntatori a funzione caratteristici del mondo C, punteranno sempre ad un metodo valido e non saranno esposti a pericoli insidiosi come l’accedere ad aree di memoria non di competenza o ad indirizzi non validi.

La keyword che introduce un delegato non lascia adito a dubbi sul suo scopo: **delegate**. Ad esempio:

```
public delegate int firstdel (int x);
```

in questa riga di codice viene definita un delegato avente nome *firstdel* che può essere usato per richiamare qualsiasi funzione che restituisca un intero accettando un intero come parametro. In generale il formato di un delegato è la seguente:

```
[modificatore] delegate tipo nome [(parametri)];
```

Come al solito tra parentesi quadre sono indicati i dati opzionali. Un altro esempio:

```
public delegate string del()
```

che rappresenta un delegato di tipo stringa e non ha parametri in ingresso. Molto simile ad un metodo ma, come è evidente, non c’è il corpo. All’origine di un delegato c’è una classe speciale del framework .Net ovvero **System.MulticastDelegate** a sua volta discendente di **System.Delegate**, ovviamente parecchi dettagli di queste importanti classi. La cosa un po’ singolare di queste è che il programmatore non può derivare da esse mentre può farlo il compilatore che da queste parte per arrivare generare un “tipo” delegato. Questa apparente rigidità comportamentale è peraltro comprensibile considerando che i delegati sono alla base di molto complessi e delicati meccanismi. Il compilatore quindi deriva una classe da System.(Multicast)Delegate mentre sarà il runtime a dar corso all’implementazione dei metodi. Un eventuale analisi dell’IL generato quindi non direbbe nulla di particolarmente interessante pertanto.

Bene, a questo punto è abbastanza semplice comprendere che un delegato è una istanza di una classe visto che da questa trae le sue origini. Perciò anche esso andrà generato tramite l’operatore new. Quindi, ripartendo dall’esempio precedente:

```
firstdel del = new firstdel(metodo)
```

E’ interessante notare, come avrete capito, che il metodo sottolineato nell’argomento è quello che il nostro delegto maschera e che deve avere i giusti requisiti di firma.

A questo punto vediamo un po’ di codice:

| C# | Esempio 6.1 |
|----|--|
| 1 | using System; |
| 2 | |
| 3 | public delegate int OperaSuInteri (int x, int y); |
| 4 | |
| 5 | public class Operazioni |
| 6 | { |
| 7 | public int somma(int a, int b) |
| 8 | { |
| 9 | return (a + b); |
| 10 | } |
| 11 | public int sottrai(int a, int b) |
| 12 | { |
| 13 | return (a - b); |
| 14 | } |
| 15 | public int moltiplica(int a, int b) |
| 16 | { |
| 17 | return (a * b); |
| 18 | } |
| 19 | } |
| 20 | |
| 21 | public class Test |
| 22 | { |
| 23 | public static void Main() |
| 24 | { |
| 25 | Operazioni o1 = new Operazioni(); |
| 26 | Console.WriteLine(o1.somma(2,3)); |
| 27 | OperaSuInteri Osi = new OperaSuInteri(o1.somma); |
| 28 | Console.WriteLine(Osi(2,3)); |
| 29 | } |
| 30 | } |

In questo esempio non vi è nulla di complicato: la riga 3 definisce il nostro delegato. Una istanza di questo viene creata alla riga 27 e, in particolare, in essa si può apprezzare il richiamo al metodo “somma” contenuto nella classe Operazioni o meglio nella istanza o1 della classe Operazioni. Il metodo coincide perfettamente in termini di firma con la definizione del delegato il quale, come si vede alla riga 28, può essere usato in sua vece. Le righe 26 e 28 infatti producono lo stesso risultato. Esiste tuttavia una notazione più semplice e compatta, trasformiamo la classe Test come segue:

```
public class Test
{
    public static void Main()
    {
        Operazioni o1 = new Operazioni();
        Console.WriteLine(o1.somma(2,3));
        OperaSuInteri Osi = o1.somma;
        Console.WriteLine(Osi(2,3));
        Osi = o1.sottrai;
        Console.WriteLine(Osi(2,3));
    }
}
```

Anche questo codice funziona perfettamente, è più elastico anche se nasconde un po’ la natura dei delegati. Che i delegati siano delle istanze di classe può essere anche ulteriormente comprovato dal seguente codice che può essere inserito nell’esempio precedente:

```
public class Test
{
    public static void Main()
    {
        Operazioni o1 = new Operazioni();
        OperaSuInteri[] ard = { new OperaSuInteri(o1.somma),
                               new OperaSuInteri(o1.sottrai) };
        Console.WriteLine(ard[0](4,2));
        Console.WriteLine(ard[1](4,2));
    }
}
```

```

    }
}

```

In pratica abbiamo usato due istanze del nostro delegato come elementi dell'array `ard`. Questo codice funziona ed è perfettamente legale e comprensibile abbracciando il punto di vista secondo cui i delegati sono istanze di classe. Non ci sono limitazioni in questo senso e vedremo come sia possibile anche passare un delegato come parametro ad un metodo. A proposito della reale natura di un delegato otiamo come esistano due interessanti proprietà nella classe `System.MulticastDelegate` e precisamente **Target** e **Method**. Aggiungiamo in coda al nostro piccolo programma di esempio le seguenti righe di codice:

```

Console.WriteLine(Osi.Target);
Console.WriteLine(Osi.Method);

```

ne ricaveremo il seguente output:

Operazioni

Int32 sottrai(Int32, Int32)

Questo in quanto `Target` restituisce l'istanza della classe alla quale il delegato è "connesso" mentre `Method` rappresenta in dettaglio il metodo di quella classe che il delegato maschera. In pratica sono le proprietà che effettuano il "puntamento" alla classe ed al relativo metodo richiesti. Ma in maniera del tutto "safe". Nulla cambia a livello pratico se viene "puntato" un metodo statico, come nel prossimo esempio che potrete esercitarvi a snellire, in effetti è creato solo per scopi didattici:

| C# | Esempio 6.1 |
|----|---|
| 1 | <code>using System;</code> |
| 2 | |
| 3 | <code>public delegate string Saluti();</code> |
| 4 | |
| 5 | <code>public class Greetings</code> |
| 6 | <code>{</code> |
| 7 | <code> public static string ciao()</code> |
| 8 | <code> {</code> |
| 9 | <code> return("ciao");</code> |
| 10 | <code> }</code> |
| 11 | <code> public static string hello()</code> |
| 12 | <code> {</code> |
| 13 | <code> return("hello");</code> |
| 14 | <code> }</code> |
| 15 | <code> public static string hola()</code> |
| 16 | <code> {</code> |
| 17 | <code> return("hola");</code> |
| 18 | <code> }</code> |
| 19 | <code> public static string nulla()</code> |
| 20 | <code> {</code> |
| 21 | <code> return ("non conosco altre lingue");</code> |
| 22 | <code> }</code> |
| 23 | <code>}</code> |
| 24 | |
| 25 | <code>public class Test</code> |
| 26 | <code>{</code> |
| 27 | <code> public static void Main()</code> |
| 28 | <code> {</code> |
| 29 | <code> Console.WriteLine("1: italiano");</code> |
| 30 | <code> Console.WriteLine("2: inglese");</code> |
| 31 | <code> Console.WriteLine("3: spagnolo");</code> |
| 32 | <code> Console.Write("Scegli la lingua: ");</code> |
| 33 | <code> string scelta = Console.ReadLine();</code> |
| 34 | <code> switch (scelta)</code> |
| 35 | <code> {</code> |
| 36 | <code> case "1":</code> |
| 37 | <code> Saluti s0 = new Saluti(Greetings.ciao);</code> |
| 38 | <code> Console.WriteLine(s0());</code> |
| 39 | <code> break;</code> |
| 40 | <code> case "2":</code> |
| 41 | <code> Saluti s1 = new Saluti(Greetings.hello);</code> |

```

42         Console.WriteLine(s1());
43         break;
44     case "3":
45         Saluti s2 = new Saluti(Greetings.hola);
46         Console.WriteLine(s2());
47         break;
48     default:
49         Saluti s3 = new Saluti(Greetings.null);
50         Console.WriteLine(s3());
51         break;
52     }
53 }
54 }

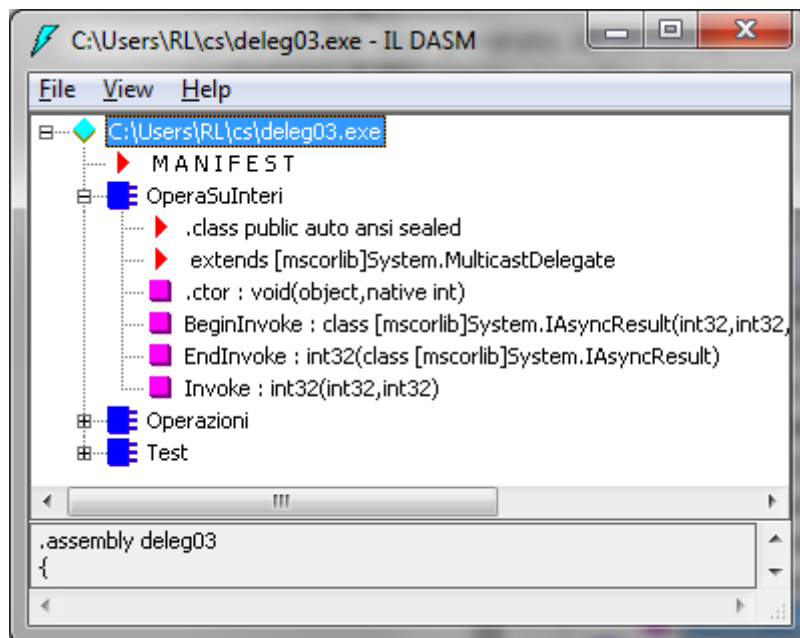
```

Un po' più lungo del nostro solito ma non c'è nulla di complicato. Ovviamente l'uso dei delegati in questo caso è una semplice forzatura a scopo illustrativo del tutto non necessaria. A livello interno l'unica reale differenza tra il richiamo ad un metodo statico e ad una istanza di metodo sta nel contenuto del membro `Target`, visto in precedenza, che nel caso di metodi statici risulta **null**.

Possiamo quindi riassumere schematicamente ed analiticamente cosa accade quando creiamo e richiamiamo un delegato:

1- Definiamo il nostro delegate. Il compilatore crea una classe derivando da **System.MulticastDelegate** che a sua volta discende da **System.Delegate**

2- Tra i vari membri del delegate ve ne sono alcuni estremamente indicativi. Per meglio analizzarli ricorriamo ad un utilissimo tool che accompagna il framework .Net e precisamente il disassemblatore che corrisponde al file **ildasm.exe** richiamabile anche da linea di comando. Questo strumento permette di aprire un file .exe generato da un compilatore .Net e ci fa dare uno sguardo alla sua struttura interna. Ad esempio analizziamo il file creato a partire dal sorgente dell'esempio 12.1. Ecco cosa ricaviamo relativamente all'analisi del delegato:



In particolare sono interessanti 3 membri e precisamente **BeginInvoke**, **EndInvoke** ed **Invoke**. Come si vede dalla loro specifica i primi due sono legati all'esecuzione asincrona del delegato (avremo modo di riparlare insieme ai thread) e in particolare **BeginInvoke** si porta dietro i parametri del metodo ed **EndInvoke** espone il tipo di ritorno mentre **Invoke** (che come si può notare reca con sé i riferimenti al tipo di ritorno ed ai parametri) viene adoperato per lanciare ogni metodo indicato nel delegato. **Invoke** non è peraltro richiamabile direttamente da C# via codice.

3- nei campi **target** e **method** ricordiamo che ci sono gli agganci alla classe ed al metodo che il delegato rappresenta.

C# - Capitolo 12 - I delegati e gli eventi

Per inciso, qualora non lo si sia notato, si evidenzia, al punto 2, la natura dei delegati di poter lavorare in modo sincrono ed asincrono.

Tuttavia le caratteristiche dei delegati non finiscono qua: è possibile infatti ora introdurre il concetto di “**catena dei delegati**”. In buona sostanza questo significa che è possibile costruire una sequenza di delegati di modo che una volta che sia chiamato quello di testa possano essere sequenzialmente richiamati quelli che seguono. Il fulcro di questa possibilità è costituito dal metodo **Combine** della classe **Delegate** che dispone di due overload:

```
Delegate.Combine(Delegate[])
```

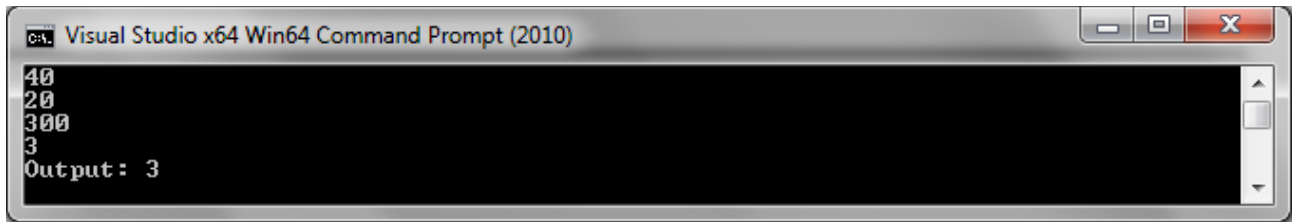
```
Delegate.Combine(Delegate, Delegate)
```

La prima chiamiamola variante agisce combinando i delegati presenti in un array mentre la seconda concatena due delegati. Vediamo un esempio e commentiamolo:

```
C#    Esempio 12.3
1    using System;
2
3    public delegate int Operatore (int x, int y);
4
5    public class Operazioni
6    {
7        public int somma(int x1, int x2)
8        {
9            int result = (x1 + x2);
10           Console.WriteLine(result);
11           return result;
12        }
13
14        public int differenza(int x1, int x2)
15        {
16            int result = (x1 - x2);
17            Console.WriteLine(result);
18            return result;
19        }
20
21        public int moltiplicazione(int x1, int x2)
22        {
23            int result = (x1 * x2);
24            Console.WriteLine(result);
25            return result;
26        }
27
28        public int divisione(int x1, int x2)
29        {
30            int result = (x1 / x2);
31            Console.WriteLine(result);
32            return result;
33        }
34    }
35
36    class Program
37    {
38        public static void Main()
39        {
40            Operazioni op1 = new Operazioni();
41            Operatore dop1 = new Operatore(op1.somma);
42            Operatore dop2 = new Operatore(op1.differenza);
43            Operatore dop3 = new Operatore(op1.moltiplicazione);
44            Operatore dop4 = new Operatore(op1.divisione);
45            Operatore[] listadel = new Operatore[] {dop1, dop2, dop3, dop4};
46            Operatore catena = (Operatore)Delegate.Combine(listadel);
47            int output = catena(30, 10);
48            Console.WriteLine("Output: {0}",output);
49        }
50    }
51 }
```

C# - Capitolo 12 - I delegati e gli eventi

La riga 45 e la 46 eseguono quanto descritto, la prima definisce un array di delegati e quella successiva esegue la concatenazione degli stessi. In particolare è interessante osservare alla riga 40 che è necessario effettuare una operazione di cast di modo che il compilatore sappia come agire nel creare la combinazione. Inoltre, osserviamo l'output:



```
Visual Studio x64 Win64 Command Prompt (2010)
40
20
300
3
Output: 3
```

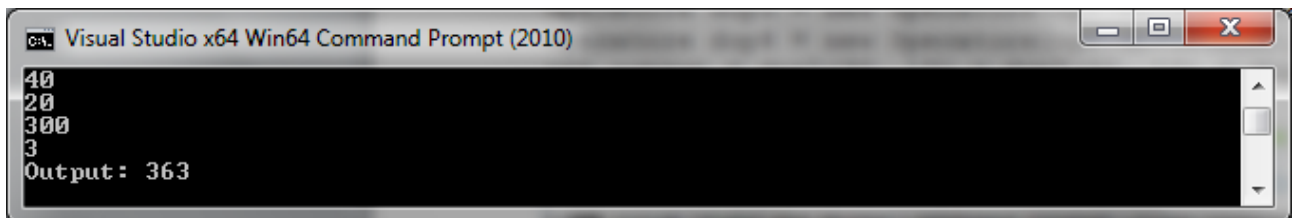
Per giustificare il quale bisogna considerare che:

1. I delegati vengono eseguiti ma l'output finale esposto corrisponde solo all'ultimo. Eventuali risultati intermedi vengono persi nell'ambito della combinazione per cui se necessario è bene prevedere output espliciti anche nei delegati precedenti l'ultimo, come fatto alle righe 10, 17, 24, 31.
2. Eventuali eccezioni sollevate durante l'esecuzione di uno dei delegate termina la catena e inizia la consueta gestione dell'eccezione stessa.
3. Ovviamente eventuali parametri ref comuni vengono modificati man mano che un delegato agisce su di essi. Quindi attenzione.

Se invece modificassimo le righe dalla 40 alla 46 ad esempio come segue:

```
Operazioni op1 = new Operazioni();
Operatore dop1 = new Operatore(op1.somma);
Operatore dop2 = new Operatore(op1.differenza);
Operatore dop3 = new Operatore(op1.moltiplicazione);
Operatore dop4 = new Operatore(op1.divisione);
int output = dop1(30, 10) + dop2(30, 10) + dop3(30, 10) + dop4(30, 10);
Console.WriteLine("Output: {0}", output);
```

otterremo ovviamente:



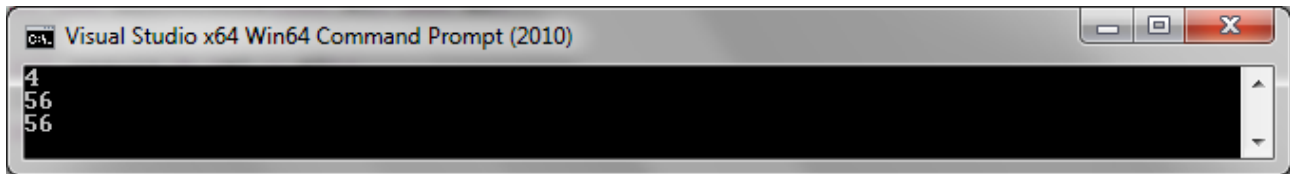
```
Visual Studio x64 Win64 Command Prompt (2010)
40
20
300
3
Output: 363
```

Una interessante proprietà di System.Delegate è quella di poter agire sulla lista dei delegati presenti in un array nell'ambito di una catena. Stiamo parlando di [GetInvocationList](#) che risulta utile nei vari casi in cui si voglia operare in un certo senso "internamente" alla lista dei delegati. L'uso di [GetInvocationList](#) è il seguente, sempre lavorando sull'esempio 12.3

```
Delegate[] oper = catena.GetInvocationList\(\);
```

in questo modo otteniamo evidentemente un array di nome oper contenente al suo interno dei delegati, genericamente tali come si evidenzia dalla parola sottolineata Delegate con cui inizia l'istruzione. Vi lascio alla semplice analisi del codice seguente, che potrete inserire nell'esempio 12.3 a partire dalla riga 45, ed al relativo output:

```
Operatore[] listadel = new Operatore[] {dop1, dop2, dop3, dop4};
Operatore catena = (Operatore)Delegate.Combine(listadel);
Delegate[] oper = catena.GetInvocationList\(\);
Console.WriteLine(oper.Length);
Console.WriteLine(\(\(\(Operatore\)oper\[2\]\)\(8, 7\)\));
```



Quando vedrete del codice C# in cui ci siano di mezzo i delegati, cioè molto spesso, troverete una notazione particolare costituita dai simboli `-=` e `+=`. Questi costituiscono il modo sicuramente più comune di aggiungere e togliere metodi da una lista. In pratica se `del01` è il nostro delegato si può scrivere:

`del01 += metodo` o anche:

`del01 = del01 + metodo` ed evidentemente

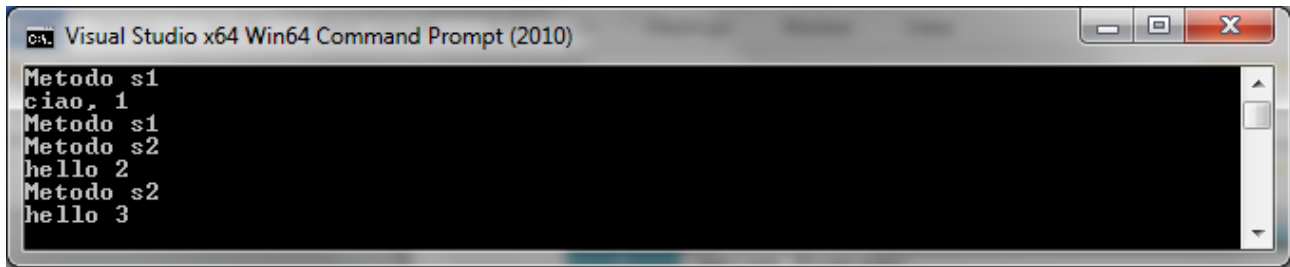
`del01 -= metodo` o anche

`del01 = del01 - metodo`

al fine di aggiungere un metodo alla lista di `del01`. Ovviamente il simbolo `-=` è utilizzato per togliere un metodo dalla lista stessa. In pratica siamo in presenza di una rappresentazione di **combine** forse più consona ai nostri occhi, ovvero espressa tramite una formulazione a cui siamo più abituati. Va specificato che, dal punto di vista dell'uso di memoria, un delegato è una entità **immutabile** per cui utilizzare su di essi gli operatori appena visti equivale in realtà a crearne uno nuovo attribuendolo poi alla variabile di riferimento. L'esempio che segue, nella sua banalità ci spiega alcune cose interessanti, peraltro già accennate:

| C# | Esempio 6.1 |
|----|--|
| 1 | <code>using System;</code> |
| 2 | |
| 3 | <code>public delegate string Saluta (string s);</code> |
| 4 | |
| 5 | <code>public class Foo</code> |
| 6 | <code>{</code> |
| 7 | <code> public string s1 (string s)</code> |
| 8 | <code> {</code> |
| 9 | <code> string temp = "ciao, " + s;</code> |
| 10 | <code> Console.WriteLine("Metodo s1");</code> |
| 11 | <code> return temp;</code> |
| 12 | <code> }</code> |
| 13 | <code> public string s2 (string s)</code> |
| 14 | <code> {</code> |
| 15 | <code> string temp = "hello " + s;</code> |
| 16 | <code> Console.WriteLine("Metodo s2");</code> |
| 17 | <code> return temp;</code> |
| 18 | <code> }</code> |
| 19 | <code>}</code> |
| 20 | |
| 21 | <code>class Program</code> |
| 22 | <code>{</code> |
| 23 | <code> public static void Main()</code> |
| 24 | <code> {</code> |
| 25 | <code> Foo f1 = new Foo();</code> |
| 26 | <code> Saluta sal = new Saluta(f1.s1);</code> |
| 27 | <code> Console.WriteLine(sal("1"));</code> |
| 28 | <code> sal += f1.s2;</code> |
| 29 | <code> Console.WriteLine(sal("2"));</code> |
| 30 | <code> sal -= f1.s1;</code> |
| 31 | <code> Console.WriteLine(sal("3"));</code> |
| 32 | <code> }</code> |
| 33 | <code>}</code> |

Con il seguente output:



```

C:\> Visual Studio x64 Win64 Command Prompt (2010)
Metodo s1
ciao, 1
Metodo s1
Metodo s2
hello 2
Metodo s2
hello 3

```

Come si può vedere la riga 27 espone l'output del primo richiamo delegato che a sua volta punta al metodo s1 della classe Foo per cui vengono espresse le prime due righe dell'output. La riga 28 aggiunge il metodo s2 della classe Foo ed è da annotare quanto detto in precedenza quando abbiamo commentato **combine** in quanto il metodo s1 viene sicuramente eseguito, come dimostra la terza riga dell'output ma l'output finale è quello dell'ultimo metodo, in pratica la stringa restituita da s1 viene "persa". La riga 30 elimina il metodo s1 che quindi scompare dall'esecuzione. Per quanto inutile ai fini pratici l'esempio mostra il meccanismo in modo abbastanza elementare e, credo, comprensibile. L'equivalente di **--** in forma per così dire verbosa e un po' più macchinosa a parer mio è **Remove** ed esiste anche **RemoveAll** che svuota completamente la lista dei metodi di un delegato. La cosa si basa sulla seguente sintassi:

```

public static Delegate Remove(Delegate sorgente, Delegate valore)
public static Delegate RemoveAll(Delegate sorgente, Delegate valore)

```

Remove e RemoveAll eliminano dalla lista dei metodi di un delegato sorgente l'ultimo metodo o tutti i metodi rispettivamente che si trovano nella lista di un altro delegato qui indicato come "valore" (nel senso che contiene il valore o i valori da eliminare, non mi è venuta in mente una definizione migliore). Esempio:

| C# | Esempio 6.1 |
|----|---|
| 1 | using System; |
| 2 | |
| 3 | public delegate string Saluta (string s); |
| 4 | |
| 5 | public class Foo |
| 6 | { |
| 7 | public string s1 (string s) |
| 8 | { |
| 9 | string temp = "ciao, " + s; |
| 10 | Console.WriteLine("Metodo s1"); |
| 11 | return temp; |
| 12 | } |
| 13 | public string s2 (string s) |
| 14 | { |
| 15 | string temp = "hello " + s; |
| 16 | return temp; |
| 17 | } |
| 18 | } |
| 19 | |
| 20 | class Program |
| 21 | { |
| 22 | public static void Main() |
| 23 | { |
| 24 | Foo f1 = new Foo(); |
| 25 | Saluta sal = new Saluta(f1.s1); |
| 26 | Saluta sal2 = new Saluta(f1.s2); |
| 27 | Console.WriteLine(sal("1")); |
| 28 | sal += f1.s2; |
| 29 | Console.WriteLine(sal("2")); |
| 30 | sal = (Saluta)Delegate.RemoveAll(sal, sal2); |
| 31 | Console.WriteLine(sal("3")); |
| 32 | } |
| 33 | } |

C# - Capitolo 12 - I delegati e gli eventi

L'istruzione critica si presenta ovviamente alla riga 30 (si poteva anche usare `Remove`). Da notare che è necessario un assegnamento ed un cast in quanto come detto un delegato è immutabile quindi bisogna in un certo senso ricrearlo. Scrivere semplicemente alla riga 30:

```
Delegate.RemoveAll(sal, sal32);
```

Compila ma, in pratica non fa nulla, come vi sarà facile constatare.

Interessante è la possibilità di lavorare con i tipi generici in congiunzione con i delegati:

public delegate void generico<T> (T param);

l'uso è abbastanza intuitivo:

| C# | Esempio 6.1 |
|----|---|
| 1 | using System; |
| 2 | public delegate void Generico<T> (T param); |
| 3 | |
| 4 | public class Test |
| 5 | { |
| 6 | public static void scrivi (string s) |
| 7 | { |
| 8 | Console.WriteLine(s); |
| 9 | } |
| 10 | public static void conta (int i) |
| 11 | { |
| 12 | Console.WriteLine(i); |
| 13 | } |
| 14 | } |
| 15 | class Program |
| 16 | { |
| 17 | static void Main() |
| 18 | { |
| 19 | Generico<string> g1 = Test.scrivi; |
| 20 | Generico<int> g2 = Test.conta; |
| 21 | } |
| 22 | } |

Le righe evidenziate nel frammento di codice qui sopra mettono in evidenza come richiamare il generico delegato ed i relativi metodi i quali presentano una parametrizzazione diversa l'uno dall'altro. Non si tratta dell'uso più comune che si fa dei delegati, in generale, ma non è comunque infrequente incontrare questo utilizzo, almeno in base alle mie esperienze.

Per quanto sia ovvio i delegati possono far riferimento anche a metodi con parametri di tipo **ref** oppure **out**. Riprendiamo a titolo di esempio un programma già visto nel capitolo 6 e precisamente l'esempio 6.3 modificandolo per consentire l'uso di un delegato:

| C# | Esempio 6.1 |
|----|---|
| 1 | using System; |
| 2 | |
| 3 | public delegate void del (int x, int y, ref int somma); |
| 4 | |
| 5 | class Program |
| 6 | { |
| 7 | public static void Somma(int x, int y, ref int somma) |
| 8 | { |
| 9 | somma = x + y; |
| 10 | } |
| 11 | public static void Main() |
| 12 | { |
| 13 | Console.Write("Inserisci un numero: "); |
| 14 | int a = int.Parse(Console.ReadLine()); |
| 15 | Console.Write("Inserisci un numero: "); |
| 16 | int b = int.Parse(Console.ReadLine()); |

```

17     int totale = 0;
18     del deleg = new del(Somma);
19     deleg(a, b, ref totale);
20     Console.WriteLine("Totale: " + totale);
21 }
22 }

```

Ovviamente qui il delegato è di troppo ma in se stesso il banale codice presentato testimonia l'uso semplice e coerente con quanto visto in precedenza.

Torneremo sui delegati quando parleremo dei thread.

EVENTI

Se è vero che tramite i delegati possiamo instaurare una comunicazione tra oggetti tramite i loro metodi va anche aggiunto che, in questo caso, la codifica si presenta a volte pesante. C# presenta un altro meccanismo che permette ad una classe di comunicare direttamente ai propri client un qualche tipo di messaggio prestabilito. Si tratta, come è evidente, di un meccanismo molto importante ed interessante con vaste potenzialità. Tutto ciò si basa su un concetto che unisce lo scatenamento di un dato evento, appunto, e la comunicazione di questo tutte quelle entità che hanno sottoscritto quello stesso evento. L'entità che genera l'evento si chiama **autore** mentre **sottoscrittore** è chi è interessato all'evento. Nella letteratura in lingua inglese si parla di **sender** e **receiver** rispettivamente. Ovviamente per ogni evento ci può essere più di un sottoscrittore. Esempio classico è quello della pressione di un pulsante su di un form e delle reazioni che si generano in quei controlli che sono interessati da questo fatto. A questo proposito bisogna ricordare che c'è ampia libertà sia per quanto riguarda la natura degli autori che dei sottoscrittori. La cosa certa è che negli ultimi, qualora abbiano sottoscritto un evento, esiste un "qualche cosa" che gestisce la notifica dell'evento stesso. Si parla quindi di "event handler" riferendoci ad una particolare entità che dovrà gestire l'evento. Anche in questo caso esiste una keyword molto semplice ed esplicativa: **event**.

L'esempio tipico che si fa in quasi tutti i testi per presentare gli eventi è quello della classica pressione di un bottone in ambiente grafico di interfaccia utente. Anche io seguo quella strada del tutto intuitiva:

| C# | Esempio 6.1 |
|----|--|
| 1 | using System; |
| 2 | using System.Windows.Forms; |
| 3 | partial class Form1 |
| 4 | { |
| 5 | private void InitializeComponent() |
| 6 | { |
| 7 | this.button1 = new System.Windows.Forms.Button(); |
| 8 | this.button1.Name = "button1"; |
| 9 | this.button1.Text = "button1"; |
| 10 | this.button1.Click += new System.EventHandler(this.button1_Click); |
| 11 | this.Controls.Add(this.button1); |
| 12 | this.Name = "Form1"; |
| 13 | this.Text = "Ciao!"; |
| 14 | } |
| 15 | private System.Windows.Forms.Button button1; |
| 16 | } |
| 17 | |
| 18 | static class Program |
| 19 | { |
| 20 | static void Main() |
| 21 | { |
| 22 | Application.Run(new Form1()); |
| 23 | } |
| 24 | } |
| 25 | |
| 26 | public partial class Form1 : Form |
| 27 | { |
| 28 | public Form1() |
| 29 | { |
| 30 | InitializeComponent(); |
| 31 | } |
| 32 | |

```

33     private void button1_Click(object sender, EventArgs e)
34     {
35         this.BackColor = System.Drawing.SystemColors.Highlight;
36     }
37 }

```

Questo programma una volta eseguito ci presenta una semplice finestra con un bottone in alto a sinistra. La pressione del bottone causa il cambiamento dello sfondo del form che lo contiene. Non è necessario al momento che vi sforziate di comprendere tutto il codice. Le parti salienti sono evidenziate alla riga 10 e alla 33. In particolare proprio alla 33 viene definita una funzione che altro non fa che definire il nuovo colore di fondo del form, ovvero come detto della finestrella che compare eseguendo il programma. Questa funzione si chiama `button1_Click` e viene richiamata alla riga 10. Questa ha un significato molto preciso: il form si iscrive all'ascolto dell'evento corrispondente alla pressione del bottone. Per fare questa operazione viene richiamato un delegato la cui signature deve corrispondere a quella della funzione che sarà richiamata, come è ovvio. Le strade da seguire sono due: si può creare un delegato custom oppure, come in questo caso, si può utilizzare quello fornito di default dal framework. Il nome del delegato fornitoci dal sistema è **EventHandler**. Esso costruisce il ponte tra l'evento e la funzione destinato a gestire tale evento. Come detto **EventHandler** è il delegato fornito dal framework qualora non siano richiesti informazioni custom relativamente all'evento. Prevede due parametri: l'oggetto sorgente dell'evento e i dati relativi all'evento stesso che sono contenuti in una variabile di tipo **EventArgs**. Quest'ultima non è che una classe che costituisce la base per gli eventi ed entra. Avrete notato la mancanza della parola `Event` nel codice precedente. Il fatto è che un bottone, nell'ambito del framework ha già predefinito in se stesso l'evento `click` richiamato a sinistra del simbolo `+=` alla riga 10. Quindi, riassumendo, cosa troviamo nell'esempio precedente?

- Un form che contiene un bottone
- Ogni bottone si porta dietro, come evento naturale, il classico `Click`
- Il form sottoscrive l'evento
- L'evento è tipizzato tramite il delegato `EventHandler` definito nell'ambito del framework
- La funzione che viene puntata dal delegato altro non fa che cambiare il colore di fondo del form.

In particolare, per amore di precisione, `Click` è uno dei tanti eventi pubblici di un bottone, ce ne sono tantissimi, provate semplicemente ad esempio nel programma di sopra a sostituire alla riga 10 `this.Button1.Click` con `this.Button1.MouseEnter` ed eseguire nuovamente.

Tutto questo risulta molto facile ed intuitivo con gli eventi già precotti a livello di framework. Ma ovviamente noi vogliamo crearci e gestirci i nostri eventi. L'esempio che segue ci mostra come fare, tenendo presente che nel mondo a riga di comando per programmi così semplici l'uso degli eventi è quasi sempre una forzatura. Non è quindi molto significativo ma è comunque il più semplice possibile o quasi, anche in rete troverete codice simile a questo, non credo sia possibile scriverne usando un numero minore di righe.

```

C#   Esempio 6.1
1   using System;
2
3   public class Evento
4   {
5       public delegate void UrloEventHandler(string str);
6       public static event UrloEventHandler Urlo;
7
8       public Evento()
9       {
10          if (Urlo != null) Urlo("AAAAAAAAAAHHHHH!");
11      }
12  }
13
14  public class Program
15  {
16      private static void Urlo(string s1)
17      {
18          Console.WriteLine("Ora caccio un urlo: " + s1);
19      }
20
21      public static void Main(string[] args)
22      {

```

```

23         Evento.Urllo += new Evento.UrlloEventHandler(Program.Urllo);
24         new Evento();
25     }
26 }

```

La riga 3 definisce una classe all'interno della quale sono definiti un delegato, riga 5, che costituisce il prototipo, in questo caso creato da noi, per il nostro evento mentre alla riga 6 troviamo l'evento vero e proprio. La riga 10 ci presenta invece l'esecuzione dell'evento. Da notare che bisogna prevedere anche il caso in cui nessuno sottoscriva l'evento cosa che avviene prendendo in considerazione il valore null, sempre alla riga 10. Se questo non avvenisse, ovvero se per esempio la riga 23 risultasse commentata il codice alla riga 24 originerebbe una eccezione di tipo `System.NullReferenceException`. La classe principale `Program`, in cui è presente anche l'entry point del programma, prevede un metodo "Urllo" definito alla riga 16 che sarà passato parametricamente all'evento in fase di sottoscrizione allo stesso (riga 23). Potete cambiare il comportamento del metodo o scriverne un altro da passare all'evento purchè ovviamente sia rispettata la signature. Come detto un evento può essere sottoscritto da più entità all'interno del programma. Come prova per la conferma della tecnica di sottoscrizione ad un evento e della disiscrizione provate a modificare il codice dell'esempio precedente, relativamente alla classe `Program`, nell'ambito del `Main`, nei due modi seguenti:

```

public static void Main(string[] args)
{
    Evento.Urllo += new Evento.UrlloEventHandler(Program.Urllo);
    new Evento();
    new Evento();
}

```

e

```

public static void Main(string[] args)
{
    Evento.Urllo += new Evento.UrlloEventHandler(Program.Urllo);
    new Evento();
    Evento.Urllo -= new Evento.UrlloEventHandler(Program.Urllo);
    new Evento();
}

```

la differenza vi risulterà evidente. Per sottoscrivere un altro evento basterà sempre ricorrere all'operatore `+=`, non vi è sovrapposizione di eventi, nel senso che una successiva sottoscrizione non annulla le precedenti, come d'altronde è ovvio.

Non abbiamo ancora utilizzato la classe `EventArgs`. Essa come detto entra in gioco quando è necessario creare eventi che prevedano un passaggio di dati. Il seguente esempio (che probabilmente troverete in forma simile anche altrove) costruisce un evento partendo da `EventArgs` ed utilizzando un delegato custom.

| C# | Esempio 6.1 |
|----|--|
| 1 | using System; |
| 2 | |
| 3 | public class Evento: EventArgs |
| 4 | { |
| 5 | public Evento(string ruolo, int stipendio) |
| 6 | { |
| 7 | this.ruolo = ruolo; |
| 8 | this.stipendio = stipendio; |
| 9 | } |
| 10 | public string ruolo; |
| 11 | public int stipendio; |
| 12 | } |
| 13 | |
| 14 | public class Personale |
| 15 | { |
| 16 | public delegate void gestore(object sender, Evento e); |
| 17 | public event gestore evento; |
| 18 | public void Indagine (string ruolo, int stipendio) |
| 19 | { |
| 20 | Evento ev = new Evento(ruolo, stipendio); |

```
21         evento(this, ev);
22     }
23 }
24
25 class Controlla
26 {
27     public Controlla(Personale p)
28     {
29         p.evento += new Personale.gestore(conteggia);
30     }
31
32     void conteggia(object sender, Evento e)
33     {
34         if (e.stipendio < 1000) Console.WriteLine("Mi pare poco");
35         if (e.stipendio == 1000) Console.WriteLine("Mi pare ok");
36         if (e.stipendio > 1000) Console.WriteLine("Mi pare troppo");
37     }
38 }
39 class Program
40 {
41     public static void Main()
42     {
43         Personale p = new Personale();
44         Controlla c = new Controlla(p);
45         p.Indagine("Operaio", 1000);
46         p.Indagine("dirigente", 3000);
47     }
48 }
```

La riga 3 è quella critica, nella quale viene generato una classe che eredita da EventArgs. Ad essa, tramite il suo costruttore, passiamo due parametri. La classe Personale crea l'evento che viene poi manipolato tramite la classe Controlla che, alla riga 29, si iscrive all'ascolto dell'evento..

Facciamo un passo avanti. Quanto visto finora costituisce la versione "classica" e "purista" degli eventi. A partire da .Net 2.0 tuttavia è possibile modificare un po' l'approccio. In questo senso mentre fino alla versione 1.1 era necessario fare riferimento esplicito ad un delegato per definire un evento, a partire appunto dalla versione 2.0 si è potuto usare il modello generico EventHandler<T> (se volete potete vederlo come un modello generico del delegato generico interno a .Net). Questo ci permette evidentemente una maggior libertà in sede di definizione degli eventi in quanto ci permette di non dover definire, in certe condizioni, un delegato custom per ogni evento.

So che non è facile, soprattutto nel secondo e nel terzo esempio, capire la logica degli eventi ma, col tempo e soprattutto in ambiti più impegnativi la cosa risulterà di più facile comprensione.