Delegate

Delegate, Lambda expression ed eventi

Progetto di Informatica classe 4^{a}

Ambiente: .NET 4.0/C# 4.0

Anno 2015

Indice generale

1 Introduzione ai *delegate*....4

1.1 Cosa è un *delegate*....4

1.2 Definizione e uso di un *delegate*....4

1.2.1 Dichiarazione e uso di una variabile (istanza) delegate....4

1.3 Referenziare più metodi contemporaneamente: "*multidelegate*"....5

1.3.1 Eliminare l'associazione tra variabile delegate e metodi....6

1.4 Uso di delegate come parametri di metodi....6

1.4.1 Passare come argomento il nome del metodo da eseguire....7

2 Esempio d'uso dei delegate....8

2.1 Presentazione del programma: "Top riders MotoGP"....8

2.1.1 Interfaccia utente....8

2.1.2 Visualizzazione dei piloti....9

2.2 Implementare l'ordinamento....9

2.2.1 Definire i criteri di ordinamento....10

2.2.2 Definire un delegate che rappresenti il criterio di ordinamento....10

2.3 Utilizzare il criterio scelto dall'utente....11

2.4 Conclusioni....11

3 Metodi anonimi....12

3.1 Definizione e uso di un metodo anonimo....12

3.2 Uso dei metodi anonimi in "Top riders MotoGP"....13

3.3 Conclusioni....13

4 Lambda expression....14

4.1 Definizione di una *lambda expression*....14

4.1.1 Scorciatoie sintattiche nelle lambda expressions....14

4.2 "Cattura" della variabili locali....15

4.3 Uso delle *lambda expressions* in "Top riders MotoGP"....16

5 *Delegate* predefiniti....17

5.1 Azioni e funzioni....18

5.2 Altri delegate predefiniti....18

5.3 Uso dei *delegate* predefiniti in "Top riders MotoGP"....18

6 Eventi....20

6.1 Esempio di un evento....20

6.1.1 Informazioni associate al delegate EventHandler....20
6.2 Eventi con parametro....21
6.2.1 Evento KeyPress....21
6.3 “Sollevare” l'evento....21
6.4 Implementare gli eventi....22
6.4.1 Implementazione evento CopiaCompletata....22
6.4.2 Implementazione evento: FileCopiato....23
6.5 Sottoscrizione degli eventi: uso di metodi....24
6.6 Sottoscrizione degli eventi: uso di lambda expression....24

1 Introduzione ai *delegate*

L'obiettivo del tutorial è quello di introdurre alla programmazione funzionale ed ***event-driven***.

1.1 Cosa è un *delegate*

Un *delegate* è un:

tipo che rappresenta un riferimento a un metodo.

Si può associare una variabile *delegate* a qualsiasi metodo compatibile; dopodiché è possibile eseguire il metodo attraverso la variabile.

1.2 Definizione e uso di un *delegate*

Un *delegate* fa sempre riferimento a un determinato tipo di metodo. Qui, per "tipo di metodo", ci si riferisce alla sua **firma** (*signature*), e cioè:

- il tipo restituito (o **void**);
- lista dei parametri (di ogni parametro è importante soltanto il tipo).

Di seguito dichiaro un *delegate* compatibile con tutti i metodi **void** con un parametro stringa (il nome del parametro è irrilevante):

```
delegate void ElaboraStringa (string s);
```

La definizione può essere collocata fuori o dentro una classe.

D'ora in avanti userò il termine *delegate* (in corsivo) per riferirmi al tipo, e il termine delegate (normale) per parlare di una variabile (istanza) di un tipo *delegate*.

1.2.1 Dichiarazione e uso di una variabile (istanza) delegate

La dichiarazione di una variabile assume la forma di qualsiasi altra dichiarazione:

```
class Program
{
    delegate void ElaboraStringa (string s);   //definisce il delegate
    static void Main(string [] args)
    {
        ElaboraStringa elabora = Visualizza; // dichiara e inizializza una variabile
        ...
    }

    static void Visualizza(string s)
    {
         Console.WriteLine(s);
    }
}
```

L'istruzione evidenziata non produce l'esecuzione del metodo `Visualizza()`, ma soltanto la sua associazione a `elabora`. Dopo la dichiarazione è possibile eseguire il metodo attraverso la variabile:

```
delegate void ElaboraStringa (string s);  //definisce il delegate
static void Main(string [] args)
{
    ElaboraStringa elabora = Visualizza;
    elabora("hello!");                  // esegue Visualizza() attraverso elabora
}
```

Ma il delegate può referenziare anche il metodo `Scrivi()`, poiché è anch'esso compatibile con il tipo `ElaboraStringa`:

```
static void Main(string [] args)
{
    ElaboraStringa elabora = Visualizza;
    elabora("hello!" );  //-> scrive "Hello !" sullo schermo
    elabora = Scrivi;
    elabora("Ciao!" );   //-> scrive "Ciao!" sul file "Log.txt"
}

static void Scrivi(string testo)
{
    var sw = new StreamWriter("Log.txt" , true );
    sw.WriteLine(testo);
    sw.Close();
}

static void Visualizza(string s)
{
    Console .WriteLine(s);
}
```

1.3 Referenziare più metodi contemporaneamente: "*multidelegate*"

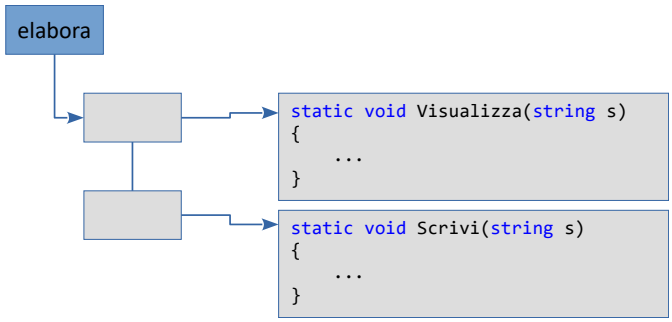
Un delegate può referenziare più metodi contemporaneamente; si parla di ***multidelegate***. L'esecuzione multipla di metodi è normalmente utilizzata nella programmazione *event-driven*.

Si consideri la seguente modifica al codice precedente:

```
 static void Main(string [] args)
{
    ElaboraStringa elabora = Visualizza;   // referenzia Visualizza
    elabora += Scrivi;                    // referenzia anche Scrivi
    elabora("Ciao!" );                    // esegue Visualizza() e Scrivi()
}
```

L'uso dell'operatore `+=` aggiunge il metodo `Scrivi()` all'elenco dei metodi referenziati dal delegate. `elabora` esegue entrambi i metodi, in ordine di associazione.

È possibile schematizzare questo stato di cose nel seguente modo:



Il delegate memorizza una collezione di riferimenti, ognuno dei quali punta a un metodo. La collezione produce una *catena di esecuzione* che rispetta l'ordine dei riferimenti nella collezione.

1.3.1 Eliminare l'associazione tra variabile delegate e metodi

Lo si fa semplicemente assegnando `null` al delegate; dopodiché, il tentativo di eseguirlo produce una `NullReferenceException`.

La situazione cambia se si desidera annullare l'associazione di un metodo senza interferire con gli altri metodi referenziati dal delegate. In questo caso si usa l'operatore `-=`:

```
static void Main(string [] args)
{
    ElaboraStringa elabora = Visualizza;
    elabora += Scrivi;
    elabora("Ciao!" );   // esegue Visualizza() e Scrivi()
    elabora -= Scrivi;   // elimina il riferimento a Scrivi
    elabora("Ciao!" );   // esegue soltanto Visualizza()
}
```

1.4 Uso di delegate come parametri di metodi

Gli esempi precedenti mostrano come associare metodi e delegate, ma non ne dimostrano l'utilità (perché dovremmo eseguire un metodo attraverso un delegate quando possiamo farlo direttamente?). La possibilità di usare delegate come parametri mostra la loro vera potenzialità.

Si consideri il seguente metodo, la cui funzione è elaborare un elenco di stringhe:

```
static void ProcessaDati(string[] elenco , ElaboraStringa elabora)
{
    foreach (var s in elenco)
    {
        elabora(s);   // -> dipende da quale metodo referenzia il delegate "elabora"
    }
}
```

Il metodo non definisce alcuna logica di elaborazione delle stringe memorizzate in `elenco`, ma si limita a dichiarare un delegate e a usarlo. L'operazione effettivamente eseguita *la decide chi esegue il metodo*:

```
static void Main(string [] args)
{
    string [] nomi = { "Einstein" , "Fermi" , "Dirac" };
```

```
    ElaboraStringa elabora = Visualizza;
    ProcessaDati(nomi, elabora); //-> visualizza i nomi sullo schermo
    elabora = Scrivi;
    ProcessaDati(nomi, elabora); //-> scrive i nomi sul file "Log.txt"
}
```

1.4.1 Passare come argomento il nome del metodo da eseguire

Il codice precedente è prolisso; non c'è alcun bisogno di utilizzare una variabile per stabilire il metodo da eseguire, basta specificarlo direttamente come argomento:

```
static void Main(string [] args)
{
    string [] nomi = { "Einstein" , "Fermi" , "Dirac" };
    ProcessaDati(nomi, Visualizza); //-> visualizza i nomi sullo schermo
    ProcessaDati(nomi, Scrivi);     //-> scrive i nomi sul file "Log.txt"
}
```

2 Esempio d'uso dei delegate

Di seguito utilizzo una semplice applicazione per mostrare i *delegate* in uno scenario significativo. Si tratta di un programma che visualizza un elenco ordinato di oggetti. Il programma sarà utilizzato anche nei capitoli successivi, quando introdurrò i *metodi anonimi* e le *lambda expressions*.

2.1 Presentazione del programma: “Top riders MotoGP”

Il programma visualizza l'elenco dei piloti del campionato MotoGP (i primi sei del campionato 2014-2015). Ogni pilota è definito dal tipo `Rider`:

```
public class Rider
{
    public string FirstName { get ; set ; }
    public string LastName { get ; set ; }
    public string Fullname { get { return LastName + ", " + FirstName; } }
    public string Team { get ; set ; }
    public int Points { get ; set ; }
}
```

I piloti sono gestiti da un oggetto di tipo `RidersRepository`, che restituisce l'elenco mediante il metodo `GetRiders()`:

```
public class RidersRepository
{
    public List<Rider> GetRiders()
    {
        var riders = new List<Rider>();
        riders.Add(new Rider { FirstName = "Valentino" , LastName = "Rossi" ,
                               Points = 325, Team = "Yamaha" });
        riders.Add(...);  // Jorge Lorenzo
        riders.Add(... ); // Dani Pedrosa
        riders.Add(...);  // Marc Marquez
        riders.Add(...);  // Smith Bradley
        riders.Add(...);  // Andrea Iannone
        return riders;
    }
}
```

2.1.1 Interfaccia utente

Attraverso dei *radiobutton* l'utente può selezionare il criterio di ordinamento, mentre cliccando sul bottone ottiene l'elenco dei piloti su un listbox. All'inizio i piloti sono visualizzati nell'ordine in sono stati inseriti nell'elenco.

2.1.2 Visualizzazione dei piloti

Segue il codice di visualizzazione dei piloti, che non tiene conto di alcun ordinamento:

```
RidersRepository repo = new RidersRepository();

private void btnUpdateRidersView_Click(object sender, EventArgs e)
{
    var riders = repo.GetRiders();
    // TODO: ordina i piloti
    lstRiders.Items.Clear();
    foreach (var rider in riders)
    {
        var text = FormatRider(rider);
        lstRiders.Items.Add(text);
    }
}

private string FormatRider(Rider r)
{
    return string.Format("{0,-18} {1,-8} {2}" , r.Fullname, r.Team, r.Points);
}
```

2.2 Implementare l'ordinamento

Poiché esistono tre criteri di ordinamento, abbiamo due opzioni:

1. realizzare tre metodi di ordinamento;
2. realizzare un metodo di ordinamento in grado di utilizzare il criterio selezionato.

La prima opzione implica una duplicazione di codice; scegliamo dunque la seconda. Per il momento implementiamo un metodo di ordinamento basato sul nome:

```
private void SortRiders(List<Rider> riders)
{
    for (int i = 0; i <riders.Count - 1; i++)
    {
        for (int j = i + 1; j <riders.Count; j++)
        {
            if (string.Compare(riders[i].Fullname, riders[j].Fullname) > 0 )
            {
                var tmp = riders[i];
                riders[i] = riders[j];
                riders[j] = tmp;
            }
        }
    }
}
```

> **Cofronto tra due stringhe: Compare()**
>
> Il metodo **string.Compare()** confronta i due argomenti e restituisce:
>
> - un intero negativo se il primo argomento è minore del secondo;
> - zero se sono uguali;
> - un intero positivo se il primo argomento è maggiore del secondo.

Esaminando `SortRiders()` si capisce che a caratterizzare i tre criteri di ordinamento è il processo di confronto dei piloti. Se vogliamo adattare l'ordinamento alla scelta dell'utente è necessario che il criterio di ordinamento *sia stabilito fuori dal metodo*.

2.2.1 Definire i criteri di ordinamento

Un criterio di ordinamento è un procedimento che, dati due piloti, restituisce -1, 0, +1 in base al fatto che il primo sia minore, uguale o maggiore del secondo.

Implementiamo tre metodi perché tre sono i criteri:

```
private int NameCompare(Rider r1, Rider r2)
{
    return r1.Fullname.CompareTo(r2.Fullname);
}

private int TeamCompare(Rider r1, Rider r2)
{
    return r1.Team.CompareTo(r2.Team);
}

private int PointsCompare(Rider r1, Rider r2)
{
    return r2.Points.CompareTo(r1.Points);
}
```

Nota bene: viene usato il metodo `CompareTo()`. Questo si comporta come `Compare()`, ma è disponibile anche per i valori numerici. (In generale è disponibile per tutti i tipi comparabili.)

2.2.2 Definire un delegate che rappresenti il criterio di ordinamento

Per utilizzare un criterio di ordinamento esterno, il metodo `SortRiders()` deve definire un parametro *delegate* appropriato. È sufficiente rispettare la firma dei tre metodi:

```
delegate int RiderComparer(Rider r1, Rider r2);
```

Ora è possibile modificare il metodo per renderlo in grado di utilizzare qualsiasi criterio:

```
private void SortRiders(List<Rider> riders, RiderComparer comparer)
{
    for (int i = 0; i < riders.Count - 1; i++)
    {
        for (int j = i + 1; j < riders.Count; j++)
        {
            if (comparer(riders[i], riders[j]) > 0)
            {
                var tmp = riders[i];
```

```
                riders[i] = riders[j];
                riders[j] = tmp;
            }
        }
    }
}
```

Ad esempio, dopo questa modifica è possibile scrivere codice come il seguente:

```
SortRiders(riders, TeamCompare);  // ordina il base al Team
```

2.3 Utilizzare il criterio scelto dall'utente

Si tratta di verificare lo stato dei tre *radiobutton* per stabilire il criterio di ordinamento da passare al metodo `SortRiders()`. L'approccio più semplice consiste nell'assegnare il metodo scelto a una variabile *delegate*, che sarà utilizzata nell'ordinamento. Scrivo pertanto un metodo che ritorna il criterio di ordinamento in base alla scelta dell'utente:

```
private RiderComparer GetComparer()
{
    if (rboTeamOrder.Checked)
        return TeamCompare;        //ordinamento per team

    if (rboPointsOrder.Checked)
        return PointsCompare;      //ordinamento per punteggio

    return NameCompare;            //ordinamento per nome
}
```

Quindi aggiorno il metodo di visualizzazione allo scopo di utilizzare il criterio selezionato.

```
private void btnUpdateRidersView_Click(object sender, EventArgs e)
{
    var riders = repo.GetRiders();
    var comparer = GetComparer();
    SortRiders(riders, comparer);
    lstRiders.Items.Clear();
    foreach (var rider in riders)
    {
        var text = FormatRider(rider);
        lstRiders.Items.Add(text);
    }
}
```

2.4 Conclusioni

L'uso di un *delegate* consente di "disaccoppiare" il codice che ordina i dati da quello che stabilisce il criterio di ordinamento: *un'eventuale modifica del secondo non influenza il primo!* Si tratta di una conquista importante; infatti, in futuro potremmo decidere di gestire altre informazioni (n° vittorie, n° podi, etc) e quindi aggiungere altri criteri di ordinamento. In questo caso non dovremo intervenire sul metodo di ordinamento, poiché resta valido qualunque sia il criterio utilizzato.

3 Metodi anonimi

L'esempio precedente mostra l'utilità dei *delegate*, ma ne evidenzia anche la "spigolosità". Siamo infatti costretti a scrivere un metodo di confronto anche quando il codice da eseguire è estremamente semplice:

```
private int NameCompare(Rider r1, Rider r2)
{
    return r1.Fullname.CompareTo(r2.Fullname);
}

private int TeamCompare(Rider r1, Rider r2)
{
    return r1.Team.CompareTo(r2.Team);
}

private int PointsCompare(Rider r1, Rider r2)
{
    return r2.Points.CompareTo(r1.Points);
}
```

In queste situazioni i **metodi anonimi** (*anonymous methods*) forniscono una scorciatoia.

3.1 Definizione e uso di un metodo anonimo

Un metodo anonimo (o metodo *inline*) è un metodo privo di nome[1], ma contenente una lista parametri e un corpo. La definizione ricalca quella di un metodo normale:

```
TipoDelegate delegate = delegate(lista parametri)
                        {
                            corpo del metodo
                        }
```

Nota bene:

- La parola chiave `delegate` sostituisce il nome del metodo.
- Il metodo anonimo deve essere assegnato a una variabile *delegate*[2].

Un metodo anonimo viene definito direttamente nel codice (e dunque dentro un altro metodo):

```
static void Main(string[] args)
{
    ElaboraStringa elabora = delegate(string s)
    {
        Console.WriteLine(s);
    };

    elabora("hello!");   // esegue il metodo anonimo
}
```

1 Per lo stesso motivo, alcuni usano chiamare i metodi normali: *named method*.
2 Questo non è strettamente vero.

3.2 Uso dei metodi anonimi in “Top riders MotoGP”

Occorre modificare il metodo `GetComparer()`. La versione originale restituisce uno dei metodi di confronto; la nuova versione restituirà un metodo anonimo:

```
private RiderComparer GetComparer()
{
    if (rboTeamOrder.Checked)
        return delegate(Rider r1, Rider r2)
        {
            return r1.Team.CompareTo(r2.Team);
        };

    if (rboPointsOrder.Checked)
        return delegate(Rider r1, Rider r2)
        {
            return r2.Points.CompareTo(r1.Points);
        };

    return delegate(Rider r1, Rider r2)
    {
        return r1.Fullname.CompareTo(r2.Fullname);
    };
}
```

Dopo questa modifica, i tre metodi “normali” possono essere eliminati.

3.3 Conclusioni

L'uso di metodi anonimi evita la necessità di definire dei normali metodi anche quando il procedimento da eseguire è molto semplice. Un secondo vantaggio è dato dalla possibilità di collegare visivamente il delegate al codice che dovrà eseguire; ciò rende esplicita la funzione del delegate.

4 Lambda expression

Una caratteristica negativa dei metodi anonimi è la sintassi poco amichevole. Le *lambda expressions* risolvono il problema, poiché forniscono delle scorciatoie sintattiche che rendono il codice semplice e leggibile.

(D'ora in avanti userò l'acronimo LE per *lambda expression*.)

4.1 Definizione di una *lambda expression*

Nella sua forma completa una LE non è molto diversa da un metodo anonimo:

Metodo anonimo

```
ElaboraStringa ma = delegate(string s)
{
    Console.WriteLine(s);
};
```

Lambda expression

```
ElaboraStringa lambda = (string s) =>
{
    Console.WriteLine(s);
};
```

Nota bene: scompare la parola `delegate` e, a sinistra della lista parametri, compare la *fat arrow* `=>`.

4.1.1 Scorciatoie sintattiche nelle lambda expressions

Il linguaggio consente di eliminare tutto ciò che può essere dedotto dal compilatore:

- il tipo dei parametri (*type inference*);
- la coppia di parentesi intorno della lista parametri se c'è un solo parametro;
- le parentesi graffe se il corpo è composto da una sola istruzione;
- la parola **return**.

Su queste basi, ecco come può essere scritta la LE precedente:

```
ElaboraStringa lambda = s => Console.WriteLine(s);
```

La LE può essere tradotta così: *data una stringa qualsiasi, scrivila sullo schermo*.

Seguono alcuni esempi; a sinistra il metodo anonimo, a destra la LE corrispondente (viene omesso il tipo *delegate*, che può essere dedotto):

Metodo anonimo

```
ma = delegate(string s)
{
    return s.ToUpper();
};
```

Lambda expression

```
lambda = s => s.ToUpper();
```

Metodo anonimo

```
ma = delegate(int a, int b)
{
    return a + b;
};
```

Lambda expression

```
lambda = (a,b) => a+b;
```

```
ma = delegate(string s)
{
    var sw = new StreamWriter(...);
    sw.WriteLine(s);
    sw.Close();
};
```

```
lambda = s =>
{
    var sw = new StreamWriter(...);
    sw.WriteLine(s);
    sw.Close();
};
```

4.2 “Cattura” della variabili locali

Una caratteristica importante delle LE (che hanno anche i metodi anonimi) è la facoltà di “catturare” le variabili locali e utilizzarle come se fossero globali.

Ad esempio, consideriamo nuovamente il metodo `ProcessaDati()`:

```
static void ProcessaDati(string[] elenco, ElaboraStringa elabora)
{
    foreach (var s in elenco)
    {
        elabora(s);
    }
}
```

Supponiamo di utilizzarlo per concatenare le stringhe dell'elenco, usando il carattere `|` per separare una stringa dalla successiva. Se volessimo usare un metodo dovremmo scrivere il seguente codice:

```
static void Main(string[] args)
{
    string[] nomi = { "Einstein", "Fermi", "Dirac" };
    ProcessaDati(nomi, Concatena);
    Console.WriteLine(testo.Trim('|')); // -> Einstein|Fermi|Dirac
}

static string testo = "";         // è necessario utilizzare una variabile globale!
static void Concatena(string s)
{
    testo = testo + s + "|";
}
```

È necessario utilizzare una variabile globale, poiché una locale non “sopravvivrebbe” tra una chiamata e la successiva del metodo. L'uso di una LE evita questa necessità:

```
static void Main(string[] args)
{
    string[] nomi = { "Einstein", "Fermi", "Dirac" };

    string testo = "";    // variabile locale
    ProcessaDati(nomi, s => testo = testo + s + "|");
    Console.WriteLine(testo.Trim('|'));   // -> Einstein|Fermi|Dirac
}
```

La variabile `testo` viene “catturata” e cioè temporaneamente promossa a variabile globale, che come tale mantiene il proprio valore tra una chiamata e l'altra della LE.

4.3 Uso delle *lambda expressions* in “Top riders MotoGP”

L'uso di LE semplifica il codice del metodo `GetComparer()`, che risulta molto più compatto e leggibile rispetto alla versione che usa i metodi anonimi:

```
private RiderComparer GetComparer()
{
    if (rboTeamOrder.Checked)
        return (r1, r2) => r1.Team.CompareTo(r2.Team);

    if (rboPointsOrder.Checked)
        return (r1, r2) => r2.Points.CompareTo(r1.Points);

    return (r1, r2) => r1.Fullname.CompareTo(r2.Fullname);
}
```

5 *Delegate* predefiniti

Le *lambda expressions* sono usate ovunque in .NET; esiste addirittura un sotto linguaggio, LINQ, che si fonda su di esse. Ciò è reso possibile dall'uso di *delegate* predefiniti. Consideriamo l'esempio utilizzato nel primo capitolo, nel quale è stato definito il *delegate* `ElaboraStringa`:

```
delegate void ElaboraStringa(string s);
static void Main(string[] args)
{
    string[] nomi = { "Einstein", "Fermi", "Dirac" };
    ProcessaDati(nomi, s => Console.WriteLine(s));
    Console.WriteLine(testo.Trim('|'));
}
static void ProcessaDati(string[] elenco, ElaboraStringa elabora)
{
    foreach (var s in elenco)
    {
        elabora(s);
    }
}
```

Ebbene, non c'è alcun bisogno di definire un nuovo *delegate*, poiché esiste già `Action<string>` che svolge la stessa funzione. Il codice si può dunque riscrivere nel seguente modo:

```
delegate void ElaboraStringa(string s);     // inutile!
static void Main(string[] args)
{
    string[] nomi = { "Einstein", "Fermi", "Dirac" };
    ProcessaDati(nomi, s => Console.WriteLine(s));
    Console.WriteLine(testo.Trim('|'));
}
static void ProcessaDati(string[] elenco, Action<string> elabora)
{
    foreach (var s in elenco)
    {
        elabora(s);
    }
}
```

Discorso analogo vale per il delegate:

```
delegate int RiderComparer(Rider r1, Rider r2);
```

Esiste già il delegate: `Func<Rider, Rider, int>`, che ha lo stessa funzione. (Come vedremo più avanti, ne esiste anche un altro.)

5.1 Azioni e funzioni

I metodi, e dunque anche i *delegate*, si possono suddividere in due grandi categorie:

- **Action method**: metodi che non ritornano un valore.
- **Function method**: metodi che ritornano un valore

All'interno di queste categorie ciò che distingue un metodo dagli altri è la lista dei parametri e il tipo del valore ritornato. I *delegate* `Action`, `Action<>` e `Func<>` rispondono a qualunque necessità ed evitano di dover scrivere i propri *delegate*.

Seguono alcuni esempi. A sinistra il delegate, al centro il metodo corrispondente, a destra un esempio di *lambda expression* corrispondente:

Delegate	Metodo	Lambda expression
`Action`	`void Met() { }`	`() => { };`
`Action<int>`	`void Met(int a) { }`	`i => sw.WriteLine(i)`
`Action<string, int>`	`void Met(string s, int i) { }`	`(s, i) => Scrivi(s,i)`
`Func<int>`	`int Met() { }`	`() => Console.Read()`
`Func<int, string>`	`string Met(int i) { }`	`i => i.ToString()`

Nota bene: nel *delegate* `Func<>` il secondo tipo specificato (o l'unico) corrisponde al tipo del valore restituito dal metodo (e dalla LE).

5.2 Altri delegate predefiniti

`Action<>` e `Func<>` sono stati definiti nella versione 3.5 del .NET. Esistono altri *delegate*, implementati nella versione precedente, ma ancora usati da alcune classi, come array e `List<>`. Tra questi ci sono `Comparison<>` e `Predicate<>`, utilizzati nei metodi di filtro e ordinamento delle collezioni.

5.3 Uso dei *delegate* predefiniti in "Top riders MotoGP"

Si tratta innanzitutto di scegliere il delegate predefinito corretto per sostituire:

```
delegate int RiderComparer(Rider r1, Rider r2);
```

Esistono due opzioni: `Comparison<Rider>` e `Func<Rider, Rider, int>`. Scelgo la prima e modifico il codice come segue:

```
private Comparison<Rider> GetComparer()
{
    ... il resto del codice è inalterato
}
```

Non c'è bisogno di modificare il codice di `GetComparer()`, poiché è perfettamente compatibile con il nuovo *delegate*.

`Comparison<>` è inoltre utilizzato dal metodo di ordinamento `Sort()` di `List<>`. Posso dunque eliminare anche il metodo di ordinamento e utilizzare quello predefinito.

Ecco il nuovo codice di visualizzazione:

```
private void btnUpdateRidersView_Click(object sender, EventArgs e)
{
    var riders = repo.GetRiders();
    var comparer = GetComparer();
    SortRiders(riders, comparer);  // eliminato!
    riders.Sort(comparer);
    lstRiders.Items.Clear();
    foreach (var rider in riders)
    {
        var text = FormatRider(rider);
        lstRiders.Items.Add(text);
    }
}
```

6 Eventi

Gli eventi sono campi *delegate* pubblici che un oggetto utilizza per notificare che "è accaduto, o sta accadendo, qualcosa"[3]:

- L'oggetto che *solleva l'evento* (esegue il delegate) è chiamato ***sender***.
- Gli oggetti che associano un metodo all'evento sono chiamati ***subscribers***.
- Il metodo (o i metodi) associato all'evento si chiama ***event handler***.

Gli eventi sono largamente impiegati dai controlli delle interfacce grafiche, ma trovano impiego anche in molte altre classi.

6.1 Esempio di un evento

Consideriamo il controllo **Button**; questo definisce un campo pubblico che rappresenta l'evento **Click**[4]:

```
public event EventHandler Click;
```

EventHandler è un *delegate* così definito:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Sottoscriversi all'evento significa associare un metodo alla variabile **Click**; il metodo deve essere compatibile con **EventHandler**.

Supponiamo che il bottone si chiami `btnLoad` e che si voglia associare il metodo `LoadUsers`:

```
private void Form1_Load(object sender, EventArgs e)
{
    btnLoad.Click += LoadUsers;
}

private void LoadUsers(object sender, EventArgs e)
{
    // qui viene gestito l'evento
}
```

Nota bene: si usa `+=` e non `=`; infatti, possono esserci più metodi associati all'evento.

6.1.1 Informazioni associate al delegate EventHandler

Il delegate **EventHandler**:

```
public delegate void EventHandler(object sender, EventArgs e);
```

definisce due parametri:

- **sender** è un riferimento all'oggetto mittente;
- **e** contiene le informazioni sull'evento, in questo caso nessuna.

3 È possibile implementare anche degli eventi statici, o di classe.
4 Le cose sono in realtà più complicate di così, ma il concetto resta valido.

L'unico significativo è il parametro **sender**. Infatti è possibile associare lo stesso *event handler* a eventi prodotti da oggetti diversi: attraverso **sender** è possibile stabilire l'identità del mittente.

Gli eventi di tipo **EventHandler** sono chiamati "eventi di sola notifica", poiché notificano ciò che è accaduto (il click sul bottone, ad esempio), senza fornire informazioni aggiuntive.

6.2 Eventi con parametro

Molti controlli definiscono eventi che forniscono informazioni attraverso il parametro **e**. A questo scopo è utilizzato il *delegate* generico **EventHandler<>**.

6.2.1 Evento KeyPress

Consideriamo l'evento **KeyPress**, sollevato da molti controlli (ad esempio i textbox) quando l'utente preme un tasto. Segue la dichiarazione dell'evento:

```
public event EventHandler<KeyPressEventArgs> KeyPress;
```

e la definizione della classe **KeyPressEventArgs**:

```
public class KeyPressEventArgs : EventArgs
{
    public KeyPressEventArgs(char keyChar);
    public bool Handled { get; set; }
    public char KeyChar { get; set; }
}
```

Nota bene: la classe deriva da **EventArgs** ed è specificata come *argomento di tipo* nel *delegate* dell'evento.

6.3 "Sollevare" l'evento

Si tratta di eseguire il delegate. Nel caso di **KeyPress**, l'esecuzione avviene come segue:

```
var e = new KeyPressEventArgs(<testo premuto>);  // crea parametro dell'evento
if (KeyPress != null)
    KeyPress(this, e);
```

Mentre, nel caso dell'evento **Click**:

```
if (Click != null)
    Click(this, EventArgs.Empty);
```

Nota bene:

- Il codice viene eseguito nella classe che definisce l'evento.
- Viene verificato il delegate, poiché potrebbe essere **null**. (Nessuno si è sottoscritto all'evento.)[5]
- Il primo argomento, **this**, rappresenta il riferimento all'oggetto mittente. Nell'**event handler**, questo sarà rappresentato dal parametro **sender**.

5 Quella fornita è un'implementazione standard, ma senz'altro migliorabile. Resta comunque valida per spiegare il concetto.

6.4 Implementare gli eventi

La classe specificata di seguito fornisce un metodo per copiare i file da una cartella ad un'altra. Per ogni file copiato viene sollevato l'evento **FileCopiato**, che specifica il nome completo del file. Alla fine del processo di copia viene sollevato l'evento di sola notifica **CopiaCompletata**.

Segue il codice di base, senza eventi:

```
public class CopiaFile
{
    // TODO: definisci eventi
    public void Copia(string sorgente, string destinazione)
    {
        string[] listaFile = Directory.GetFiles(sorgente);
        foreach (var file in listaFile)
        {
            string fileDest = Path.Combine(destinazione, Path.GetFileName(file));
            File.Copy(file, fileDest, true);
            // TODO: solleva evento FileCopiato
        }
        // TODO: solleva evento CopiaCompletata
    }
}

// TODO: definisci classe parametro per l'evento FileCopiato
```

6.4.1 Implementazione evento CopiaCompletata

L'implementazione implica due passi: definizione del delegate e sua esecuzione.

```
public class CopiaFile
{
    public event EventHandler CopiaCompletata;

    public void Copia(string sorgente, string destinazione)
    {
        string[] listaFile = Directory.GetFiles(sorgente);
        foreach (var file in listaFile)
        {
            string fileDest = Path.Combine(destinazione, Path.GetFileName(file));
            File.Copy(file, fileDest, true);
            // TODO: solleva evento FileCopiato
        }
        OnCopiaCompletata();
    }
    private void OnCopiaCompletata()
    {
        if (CopiaCompletata != null)
            CopiaCompletata(this, EventArgs.Empty);
    }
}
```

L'esecuzione del delegate è stata incapsulata in un metodo per ragioni pratiche e di leggibilità del codice. (Si tratta di un pattern comune.)

6.4.2 Implementazione evento: FileCopiato

Questo evento necessita di un passo in più: il tipo che memorizza il parametro dell'evento, e cioè il percorso del file copiato.

```
public class FileCopiatoEventArgs: EventArgs
{
    public string NomeFile { get; set; }
}
```

Nota bene: la classe deriva da **EventArgs**.

```
public class CopiaFile
{
    public event EventHandler CopiaCompletata;
    public event EventHandler<FileCopiatoEventArgs> FileCopiato;

    public void Copia(string sorgente, string destinazione)
    {
        string[] listaFile = Directory.GetFiles(sorgente);
        foreach (var file in listaFile)
        {
            string fileDest = Path.Combine(destinazione, Path.GetFileName(file));
            File.Copy(file, fileDest, true);
            var e = new FileCopiatoEventArgs { NomeFile = fileDest };
            OnFileCopiato(e);
        }
        OnCopiaCompletata();
    }

    private void OnFileCopiato(FileCopiatoEventArgs e)
    {
        if (FileCopiato != null)
            FileCopiato(this, e);
    }

    private void OnCopiaCompletata()
    {
        if (CopiaCompletata != null)
            CopiaCompletata(this, EventArgs.Empty);
    }
}
```

Nota bene: il metodo `OnFileCopiato()` riceve come argomento il parametro dell'evento, creato nel metodo `Copia()`, e lo passa al delegate `FileCopiato`.

6.5 Sottoscrizione degli eventi: uso di metodi

Valgono gli stessi meccanismi utilizzati per gli eventi dell'interfaccia utente:

```
static void Main(string[] args)
{
    CopiaFile copiaFile = new CopiaFile();
    copiaFile.CopiaCompletata += copiaFile_CopiaCompletata;
    copiaFile.FileCopiato += copiaFile_FileCopiato;
    copiaFile.Copia(".", "TEMP");
    ....
}

static void copiaFile_FileCopiato(object sender, FileCopiatoEventArgs e)
{
    Console.WriteLine(e.NomeFile);
}

static void copiaFile_CopiaCompletata(object sender, EventArgs e)
{
    Console.WriteLine("Copia completata");
}
```

Nota bene: per gli *event handler* ho usato i nomi prodotti da Visual Studio, che rispettano il pattern: **<oggetto>_<evento>**.

6.6 Sottoscrizione degli eventi: uso di lambda expression

Poiché gli eventi sono dei delegate è legittimo utilizzare delle *lambda expressions*. Ciò è vantaggioso quando l'*event handler* è composto da poche istruzioni, o addirittura una soltanto:

```
static void Main(string[] args)
{
    CopiaFile copiaFile = new CopiaFile();
    copiaFile.FileCopiato += (s, e) => Console.WriteLine(e.NomeFile);
    copiaFile.CopiaCompletata += (s, e) => Console.WriteLine("Copia completata");
    copiaFile.Copia(".", "TEMP");
    ...
}
// Gli event-handler non sono più necessari!
```

Nota bene: il *type inference* evita la necessità di definire il tipo dei parametri `s` ed `e`. Il linguaggio (e l'intellisense di Visual Studio) riconosce che il parametro `e` della prima LE è di tipo **FileCopiatoEventArgs**.