

Delegate

Delegate, lambda expression ed eventi

Anno 2018-19

Indice generale

1	Introduzione.....	4
1.1	Gestire una lista di piloti: filtrare i piloti.....	4
1.1.1	Implementare il metodo di filtro nella classe.....	5
2	Definizione e uso dei <i>delegate</i>.....	7
2.1	Definizione di un <i>delegate</i>	7
2.1.1	Terminologia.....	7
2.1.2	Dichiarazione e uso (esecuzione) di variabili <i>delegate</i>	7
2.2	Passare delegate come argomenti.....	8
3	Esempio d'uso dei delegate.....	9
3.1	Usare un delegate per stabilire il criterio di filtro.....	9
3.2	Chiamare il metodo passando una condizione di filtro.....	10
3.3	Conclusioni.....	10
4	Metodi anonimi: lambda expression.....	11
4.1	Lambda expression.....	11
4.1.1	Sintassi delle lambda expressions.....	12
4.2	Utilizzare una <i>lambda expression</i> per filtrare i piloti.....	12
5	<i>Delegate</i> predefiniti.....	13
5.1	Delegate predefiniti: azioni e funzioni.....	13
5.2	Uso di un <i>delegate</i> predefinito per il criterio di filtro dei piloti.....	14
5.3	Esempi di <i>delegate</i> predefiniti e di metodi corrispondenti.....	14
5.4	Altri delegate predefiniti.....	15
6	Uso di <i>delegate</i> generici (in classi generiche).....	16
6.1	Filtrare una lista generica.....	16
6.2	Conclusioni.....	17
7	Eventi (programmazione <i>event-driven</i>).....	18
7.1	Definizione di un evento.....	18
7.1.1	Terminologia.....	18
7.2	Eventi di sola notifica ed eventi con parametro.....	19

7.3	Implementare gli eventi mediante i <i>delegate</i> predefiniti.....	19
8	Programmazione <i>event-driven</i>: lista osservabile.....	20
8.1	Definizione dell'evento.....	20
8.1.1	Parametro dell'evento.....	20
8.1.2	Scegliere il delegate.....	20
8.1.3	Dichiarazione dell'evento (campo delegate).....	21
8.2	Sollevare l'evento.....	21
8.2.1	Verificare che l'evento sia stato "sottoscritto"	21
8.2.2	Usare un metodo helper per sollevare l'evento.....	22
8.3	Sottoscrivere all'evento.....	22
8.3.1	Operatore di sottoscrizione all'evento.....	23
8.4	Conclusioni.....	23

1 Introduzione

In questo tutorial affronto i *delegate*, i quali estendono il modello di programmazione procedurale. È sui *delegate* che si basano due modelli di programmazione molto comuni: *funzionale* ed *event-driven*.

Prenderò spunto da una funzione già affrontata, l'implementazione di una *lista*, e mostrerò come, attraverso l'uso di *delegate*, sia possibile potenziarla e renderla fruibile in una più ampia gamma di scenari. Contestualmente introdurrò le *lambda expression*.

Successivamente generalizzerò la soluzione attraverso l'uso dei *generics* e dunque di *delegate* generici.

Infine mostrerò la funzione dei *delegate* nel modello di programmazione *event-driven*.

1.1 Gestire una lista di piloti: filtrare i piloti

Partendo dall'ipotesi che non esista il tipo generico `List<>`, considera la classe `ListaPiloti`, che gestisce una collezione di record `Pilota`. Segue l'interfaccia pubblica della classe:

```
public class Pilota
{
    public string Nominativo;
    public string Moto;
    public string Vittorie;
    public int Punti;
}

public class ListaPiloti
{
    public ListaPiloti(int capacity = 10) {...}

    public int Count {...}
    public Pilota this[int index] {...}

    public void Clear() {...}
    public void Add(Pilota value) {...}
    public void RemoveAt(int index) {...}
}
```

`ListaPiloti` fornisce soltanto le operazioni fondamentali, dunque non definisce funzioni molto comuni, come l'ordinamento e il filtro (piloti che soddisfano una determinata condizione). Si tratta di operazioni per le quali sono possibili molteplici implementazioni:

- Ordinare i piloti in base alla classifica, oppure in ordine alfabetico, oppure in base al numero di vittorie ottenute.
- Ottenere i piloti di una moto, oppure quelli con almeno una vittoria, etc.

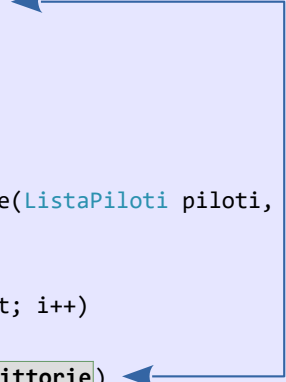
Per brevità, di seguito considero soltanto la funzione di filtro, proponendo dei metodi che filtrano i piloti in base alla moto e al numero di vittorie:

```

static ListaPiloti PilotiPerMoto(ListaPiloti piloti, string moto)
{
    var ris = new ListaPiloti();
    for (int i = 0; i < piloti.Count; i++)
    {
        if (piloti[i].Moto == moto)
            ris.Add(piloti[i]);
    }
    return ris;
}

static ListaPiloti PilotiConVittorie(ListaPiloti piloti, int vittorie)
{
    var ris = new ListaPiloti();
    for (int i = 0; i < piloti.Count; i++)
    {
        if (piloti[i].Vittorie >= vittorie)
            ris.Add(piloti[i]);
    }
    return ris;
}

```



Ho evidenziato le parti che differenziano i due metodi, e cioè le condizioni utilizzate per filtrare. Come si vede, per il resto sono identici.

1.1.1 Implementare il metodo di filtro nella classe

Sarebbe molto conveniente implementare questa funzione nella classe, ma non è possibile, perché il criterio di filtro dipende dai requisiti dell'applicazione e non può essere stabilito una volta per tutte. Diventerebbe possibile, però, se `ListaPiloti` potesse delegare al codice esterno il criterio da applicare, limitandosi a definire il procedimento generale.

In sostanza, sarebbe utile poter scrivere un metodo come il seguente:

```

public class ListaPiloti
{
    ...
    public ListaPiloti PilotiPer(<condizione>)
    {
        var ris = new ListaPiloti();
        for (int i = 0; i < Count; i++)
        {
            if (<condizione>) // l'elemento deve essere incluso?
                ris.Add(items[i]);
        }
        return ris;
    }
}

```

Perché il metodo precedente serva lo scopo, nel codice applicativo dovrebbe essere possibile invocarlo specificando la condizione di filtro:

```

class Program
{
    static void Main(string[] args)
    {
        ListaPiloti piloti = new ListaPiloti();
        //... carica piloti
        var pilotiHonda = piloti.PilotiPer(moto == "Honda"); // solo i piloti Honda
        ...
        var pilotiVittoria = piloti.PilotiPer(vittorie >= 1); // solo i piloti vittoriosi
        ...
    }
}

```

In sostanza, si tratta di poter passare come argomento non un dato, *ma il codice da eseguire*, in modo che possa essere utilizzato all'interno del metodo chiamato.

Si tratta di una soluzione resa possibile dai *delegate*.

2 Definizione e uso dei *delegate*

Un *delegate* è un:

tipo che consente di dichiarare variabili che referenziano metodi

Un *delegate* è per i metodi ciò che un tipo qualsiasi è per i dati: consente di dichiarare variabili; assegnare metodi a variabili; eseguire i metodi associati alle variabili.

2.1 Definizione di un *delegate*

Come ci deve essere compatibilità tra il tipo di una variabile e il tipo dei valori che memorizza, così una variabile *delegate* deve essere compatibile con il metodo che referencia. Un *delegate* fa sempre riferimento a un tipo di metodo, più precisamente all'*intestazione* del metodo:

- Il tipo restituito (o `void`).
- La lista dei parametri (di ogni parametro è importante soltanto il tipo).

Segue la dichiarazione di un *delegate* compatibile con tutti i metodi `void` che dichiarano un parametro `string` (il nome del parametro è irrilevante):

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

`ElaboraStringa` introduce un nuovo tipo, che consente di dichiarare variabili in grado di referenziare metodi `void` con un parametro `string`.

2.1.1 Terminologia

Con il termine *delegate*, in corsivo, intendo un tipo, mediante il quale è possibile dichiarare variabili. Per queste ultime, nel testo, userò il termine *delegate* scritto normale.

2.1.2 Dichiarazione e uso (esecuzione) di variabili *delegate*

La dichiarazione di una variabile *delegate* assume la forma di qualsiasi altra dichiarazione. Il codice seguente dichiara e inizializza una variabile locale di tipo `ElaboraStringa`:

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

    static void Visualizza(string s) //è compatibile con il delegate ElaboraStringa
    {
        Console.WriteLine(s);
    }
}
```

Nota bene:

- Il delegate `ElaboraStringa` corrisponde all'instanziazione del metodo `Visualizza()`.
- Nell'assegnazione, `Visualizza` è senza parentesi: l'istruzione non esegue il metodo, lo assegna `elabora`. Successivamente, è possibile eseguirlo attraverso la variabile:

Eseguire la variabile *delegate* significa utilizzarla come se corrispondesse a un metodo:

```
...
static void Main(string [] args)
{
    ElaboraStringa elabora = Visualizza;
    ...
    elabora("hello!");           // esegue Visualizza()
}
```

In conclusione: la variabile `elabora` consente di eseguire il metodo `Visualizza()`, o qualsiasi altro metodo compatibile, senza nominarlo direttamente. È questo meccanismo di esecuzione indiretta a dare ai *delegate* la loro potenza espressiva.

2.2 Passare delegate come argomenti

La possibilità di usare parametri *delegate* mostra la loro vera potenzialità. Considera 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);           //-> l'operazione dipende dal metodo assegnato a "elabora"
    }
}
```

`ProcessaDati()` non esegue un'operazione specifica sugli elementi del vettore, si limita a passarli al delegate `elabora`. L'operazione effettivamente eseguita la decide chi esegue il metodo:

```
static void Main(string [] args)
{
    string [] nomi = { "Einstein" , "Fermi" , "Dirac" };
    ProcessaDati(nomi, Visualizza); //-> visualizza ogni elemento del vettore
}
```

Naturalmente è possibile passare come argomento qualsiasi metodo compatibile con il *delegate* `ElaboraStringa` e, in questo modo, stabilire il procedimento da applicare agli elementi del vettore.

3 Esempio d'uso dei delegate

Considera nuovamente `ListaPiloti` e l'idea di implementare la funzione di filtro nella classe, però in modo che sia il codice applicativo a stabilire i criteri da utilizzare.

3.1 Usare un delegate per stabilire il criterio di filtro

Riporto nuovamente lo pseudo-codice del metodo `PilotiPer()`, che richiede la condizione da applicare per stabilire se includere o meno un pilota nel risultato.

```
public class ListaPiloti
{
    ...
    public ListaPiloti PilotiPer(<condizione>)
    {
        var ris = new ListaPiloti();
        for (int i = 0; i < Count; i++)
        {
            if (<condizione>)           //ad esempio: items[i].Moto == "Honda"
                ris.Add(items[i]);     //oppure    : items[i].Vittorie > 0
        }
        return ris;
    }
}
```

Occorre innanzitutto definire un *delegate* che rappresenti la condizione. Questa si applica a un pilota e, come qualsiasi condizione, restituisce `true` o `false`:

```
public delegate bool FiltroPilota (Pilota p);
```

Nota bene: il nome del *delegate* non è importante (al di là del significato che veicola); ciò che conta è il tipo di operazione che esprime, compatibile con un metodo che riceve un pilota e restituisce un valore booleano.

Adesso è possibile completare il metodo `PilotiPer()`, definendo un parametro `FiltroPilota`:

```
public ListaPiloti PilotiPer(FiltroPilota filtro)
{
    var ris = new ListaPiloti();
    for (int i = 0; i < Count; i++)
    {
        if (filtro(items[i]) //se filtro() restituisce true, include il pilota
            ris.Add(items[i]);
        }
        return ris;
    }
}
```

3.2 Chiamare il metodo passando una condizione di filtro

Per ottenere un elenco di piloti sulla base di una determinata condizione è sufficiente incapsularla in un metodo da passare a `PilotiPer()`. Di seguito definisco due metodi di filtro, per ottenere i piloti della "Honda" e della "Yamaha":

```
static void Main(string[] args)
{
    var piloti = new ListaPiloti();
    var pilotiHonda = piloti.PilotiPer(PilotiHonda);
    //->"Marquez, Marc", "Pedrosa, Dani"

    var pilotiYamaha = piloti.PilotiPer(PilotiYamaha);
    //->"Rossi, Valentino", "Lorenzo, Jorge"
}

static bool PilotiHonda(Pilota p) => p.Moto == "Honda";

static bool PilotiYamaha(Pilota p) => p.Moto == "Yamaha";
```

Nota bene: ho definito i metodi usando la sintassi *expression body*.

3.3 Conclusioni

L'uso del delegate `FiltroPilota` rende possibile "disaccoppiare" il codice che filtra i piloti (metodo `PilotiPer()`) da quello che stabilisce la condizione da usare. Si tratta di una conquista importante, poiché consente di rendere una funzione "programmabile", e dunque adattabile alle esigenze del codice applicativo che la usa.

4 Metodi anonimi: lambda expression

Considera il seguente esempio, nel quale l'utente inserisce una casa motociclistica e il programma risponde visualizzando l'elenco dei piloti che vi corrono:

```
class Program
{
    static string moto;
    static void Main(string[] args)
    {
        var piloti = new ListaPiloti();
        //... carica piloti

        moto = Console.ReadLine();
        var pilotiMoto = piloti.PilotiPer(PilotiMoto);
        Visualizza(PilotiMoto);
        ...
    }

    static bool PilotiMoto(Pilota p) => p.Moto == moto;

    static void Visualizza(ListaPiloti piloti) {...}
}
```

Si nota un problema: occorre utilizzare una variabile globale per memorizzare il valore inserito dall'utente, altrimenti la moto non sarebbe accessibile al metodo che incapsula il criterio di filtro, `PilotiMoto()`.

Si tratta di un esempio specifico di un problema più generale: la necessità di implementare un metodo quando, in molti casi, occorre delegare l'esecuzione di una semplice istruzione.

Le *lambda expression* risolvono elegantemente questo problema.

4.1 Lambda expression

Una *lambda expression*, o metodo anonimo, equivale a un metodo privo degli elementi non rilevanti alla sua esecuzione, compreso il nome.

Considera il metodo `PilotiHonda()`, nel quale evidenzio le parti rilevanti: tipo restituito, tipo e nome del parametro, corpo del metodo:

```
static bool PilotiHonda(Pilota p) => p.Moto == "Honda";
```

In realtà, se si considera che il metodo viene assegnato a una variabile *delegate* (con la quale deve essere compatibile), si possono omettere altri elementi, poiché sono deducibili dal linguaggio. Ciò che resta, compresa la *fat arrow*, è una *lambda expression*:

```
static bool PilotiHonda(Pilota p) => p.Moto == "Honda";
```

che si legge come: dato un pilota `p` restituisce il valore della condizione `p.Moto == "Honda"`.

4.1.1 Sintassi delle *lambda expressions*

Come detto, una *lambda expression* è un metodo al quale il linguaggio consente di eliminare tutto ciò che è irrilevante alla sua esecuzione, compreso ciò che può essere dedotto dal compilatore:

- Il nome.
- il tipo dei parametri (viene dedotto dal tipo *delegate* della variabile utilizzata).
- la coppia di parentesi intorno della lista parametri, ma soltanto se c'è un solo parametro.
(Se non ci sono parametri, occorre una coppia di parentesi vuota.)
- Le parentesi graffe se il corpo del metodo è composto da una sola istruzione.
- La parola `return`.

(Gli ultimi due punti valgono anche con i normali metodi, mediante la sintassi *expression body*.)

Ovviamente, una *lambda expression* non può essere eseguita direttamente; deve essere assegnata a una variabile *delegate*.

4.2 Utilizzare una *lambda expression* per filtrare i piloti

Scompare il metodo `PilotiMoto()` e dunque anche la necessità di usare una variabile globale:

```
class Program
{
    static string moto;
    static void Main(string[] args)
    {
        var piloti = new ListaPiloti();
        //... carica piloti

        var moto = Console.ReadLine();
        var pilotiMoto = piloti.PilotiPer(p => p.Moto == moto);
        Visualizza(PilotiMoto);
        ...
    }

    static bool PilotiMoto(Pilota p) => p.Moto == moto;

    static void Visualizza(ListaPiloti piloti) {...}
}
```

Dietro le quinte il linguaggio produce un codice del tutto equivalente a quello di inizio capitolo:

- Definisce un metodo equivalente alla *lambda expression*.
- *Cattura* la variabile locale `moto`, e cioè la dichiara globale, rendendola però accessibile soltanto nel metodo che la definisce e in quello corrispondente alla *lambda expression*.
- Passa il metodo suddetto come argomento a `PilotiPer()`.

5 Delegate predefiniti

In C# l'appartenenza a un determinato tipo implica regole formali e stringenti sulle operazioni ammissibili sulle variabili. Implica anche che due tipi strutturalmente identici non sono uguali.

Ad esempio, considera i seguenti *record*, strutturalmente identici:

```
class Persona
{
    public string Nome;
    public string Cognome;
}
```

```
class Dipendente
{
    public string Nome;
    public string Cognome;
}
```

Il seguente codice è formalmente scorretto:

```
Persona p = new Dipendente { Nome = "Filippo", Cognome = "Rossi" };
```

perché non si può assegnare un oggetto di tipo `Dipendente` a una variabile di tipo `Persona`, anche se i due tipi sono sostanzialmente uguali.

Questo problema esiste anche per i *delegate* e, se non gestito correttamente, comporterebbe una proliferazione di *delegate* funzionalmente identici, ma formalmente incompatibili.

Ad esempio, i seguenti *delegate* rappresentano la stessa funzione:

```
public delegate bool FiltroPilota (Pilota p);
public delegate bool CondizionePilota (Pilota p);
public delegate bool Criterio(Pilota p);
public delegate bool Condizione(Pilota p);
```

ma, per il linguaggio, sono quattro tipi distinti.

Per questo motivo sono stati definiti dei delegate generici che coprono tutte le funzioni richieste nella maggior parte dei programmi ed evitano al programmatore di dover scrivere i propri *delegate*. Sono ampiamente usati negli *oggetti* predefiniti del .NET Framework; su di essi si fonda un sotto linguaggio chiamato LINQ.

5.1 Delegate predefiniti: azioni e funzioni

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

- **Action method**, o metodi che non ritornano un valore (metodi `void`): `Action`, `Action<>`, `EventHandler`, `EventHandler<>`.
- **Function method**, o metodi che ritornano un valore: `Func<>`.

Questi *delegate* rispondono a qualunque necessità ed evitano di dover scrivere i propri *delegate*. I *delegate*, `EventHandler` e `EventHandler<>` sono usati prevalentemente nella definizione degli eventi. (Vedi 7)

5.2 Uso di un *delegate* predefinito per il criterio di filtro dei piloti

Il *delegate*:

```
public delegate bool FiltroPilota (Pilota p);
```

corrisponde a un metodo che riceve un pilota e restituisce un `bool`. Ebbene, la stessa funzione può essere espressa dal *delegate*: `Func<Pilota, bool>`.

La classe `ListaPiloti` può essere così modificata:

```
public delegate bool FiltroPilota (Pilota p);    //non più necessario

public class ListaPiloti
{
    ...
    public ListaPiloti PilotiPer(Func<Pilota, bool> filtro)
    {
        var ris = new ListaPiloti();
        for (int i = 0; i < Count; i++)
        {
            if (filtro(items[i]))
                ris.Add(items[i]);
        }
        return ris;
    }
}
```

Il codice applicativo, invece, non richiede alcuna modifica.

5.3 Esempi di *delegate* predefiniti e di metodi corrispondenti

A sinistra il *delegate*, al centro un metodo corrispondente, a destra una *lambda expression* corrispondente:

Delegate	Metodo	Lambda expression
<code>Action</code>	<code>void M() { }</code>	<code>() => Write("Rossi");</code>
<code>Action<int></code>	<code>void M(int a) { }</code>	<code>i => Write(i);</code>
<code>Action<string, int></code>	<code>void M(string s, int i) { }</code>	<code>(s, i) => Write(\$"{s}{i}")</code>
<code>Func<int></code>	<code>int M() { }</code>	<code>() => Console.Read()</code>
<code>Func<int, string></code>	<code>string M(int i) { }</code>	<code>i => i.ToString()</code>

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

5.4 Altri delegate predefiniti

`Action<>` e `Func<>` sono stati introdotti nella versione 3.5 del .NET. Esistono altri *delegate*, implementati nella versione precedente del framework, ma ancora usati in alcune classi, come `List<>` ad esempio. Tra questi ci sono `Comparison<>` e `Predicate<>`.

Ad esempio, è possibile riscrivere il metodo `PilotiPer()` utilizzando `Predicate<>` al posto di `Func<>`. Infatti: `Predicate<Pilota>` equivale a `Func<Pilota, bool>`:

```
public ListaPiloti PilotiPer(Predicate<Pilota> filtro)
{
    var ris = new ListaPiloti();
    for (int i = 0; i < Count; i++)
    {
        if (filtro(items[i]))
            ris.Add(items[i]);
    }
    return ris;
}
```

6 Uso di *delegate* generici (in classi generiche)

La possibilità di definire *delegate* generici è fondamentale ed è, tra le altre cose, alla base della programmazione funzionale (LINQ). Il loro uso in classi come `List<T>` consente di implementare funzioni come ordinamento, filtro, etc, che altrimenti dovrebbero essere gestite nel codice applicativo.

6.1 Filtrare una lista generica

Considera la lista generica `Lista<T>`, della quale fornisco l'interfaccia pubblica, e l'idea di aggiungere un metodo di filtro analogo a quello di `ListaPiloti`:

```
public class Lista<T>
{
    public Lista(int capacity = 10) {...}

    public int Count {...}
    public T this[int index] {...}

    public void Clear() {...}
    public void Add(T value) {...}
    public void RemoveAt(int index) {...}
}
```

Il codice da scrivere è praticamente identico, con la differenza che occorre utilizzare un *delegate* generico, definendone uno proprio, oppure utilizzandone uno appropriato tra quelli predefiniti: `Predicate<T>` o `Func<T, bool>`.

Di seguito mostro come definire e utilizzare un mio *delegate*:

```
public delegate bool Filtro<T>(T t); // in realtà non serve definirlo
...                                // conviene usare un delegate predefinito
public class Lista<T>
{
    ...
    public Lista<T> FindAll(Filtro<T> filter) // oppure: Func<T, bool> filter
    {                                         // oppure: Predicate<T> filter
        var ris = new Lista<T>();
        for (int i = 0; i < Count; i++)
        {
            if (filter(items[i]))
                ris.Add(items[i]);
        }
        return ris;
    }
}
```

Si usa `Lista<T>` esattamente come lista piloti, con la notevole differenza che, essendo generica, può essere usata per collezionare oggetti di qualsiasi tipo:


```

static void Main(string[] args)
{
    var piloti = new Lista<Pilota>();
    //... carica piloti

    var moto = Console.ReadLine();    //->ipotizza: "Honda"
    var pilotiHonda = piloti.FindAll(p => p.Moto == moto);
    //->"Marquez, Marc", "Pedrosa, Dani"

    var numeri = new Lista<int>();
    //... carica numeri: 1, 4, -3, -2
    var negativi = numeri.FindAll(n => n < 0);
    //-> -3, -2
}

```

Nota bene: il linguaggio, mediante *type inference*, è in grado di capire che la variabile `p` nella prima *lambda expression* è di tipo `Pilota`, mentre la variabile `n` nella seconda è di tipo `int`.

6.2 Conclusioni

Appare evidente che qualsiasi struttura dati – *lista, coda, dizionario, pila*, etc - deve essere definita in modo generico, altrimenti sarebbe necessario fornire una versione per ogni tipo di dato da gestire. In questo senso i *delegati* generici sono fondamentali; in loro assenza non sarebbe possibile dotare classi simili di funzioni comuni e ampiamente utilizzate nella programmazione.¹

1 Affermazione corretta soltanto alla luce delle nostre attuali conoscenze del linguaggio. In realtà esistono delle alternative.

7 Eventi (programmazione event-driven)

Gli eventi sono variabili *delegate* usate per notificare che “è accaduto”, “sta accadendo” o “sta per accadere” qualcosa. Possono avere varie funzioni; ad esempio:

- Si è verificato un evento esterno: un tasto premuto (`KeyPress`); un click del mouse su un bottone (`Click`). Un processo è stato completato (`DownloadFileCompleted`).
- C'è stato un cambiamento in un oggetto: è stato selezionato un elemento in un *listbox* (`SelectedIndexChanged`); è cambiato il testo di un *textbox* (`TextChanged`); una collezione è stata modificata (`CollectionChanged`).
- C'è stato un avanzamento in un processo: (`DownloadProgressChanged`).
- Sta per essere eseguita un'operazione: un *form* sta per essere chiuso (`FormClosing`). Etc.

Queste e altre funzioni rientrano nel modello di *programmazione event-driven* (guidato dagli eventi), il quale, mediante l'uso di *delegate*, semplifica la comunicazione tra i componenti di un'applicazione.

7.1 Definire di un evento

La definizione di un evento ricalca quella di qualsiasi variabile *delegate*, eccetto l'uso della parola chiave `event`:

```
public event <tipo-delegate> <variabile>
```

`event` istruisce il linguaggio a fornire un “trattamento speciale” alla variabile. In pratica impedisce che sia usata (eseguita) dal codice esterno².

7.1.1 Terminologia

Nella programmazione *event-driven* esistono alcuni concetti e termini convenzionali che occorre ricordare.

- L'azione di eseguire un evento (eseguire la variabile *delegate*) viene chiamata **sollevare l'evento**.
- Il componente che *solleva l'evento* è chiamato **sender** (mittente dell'evento).
- I componenti che associano un metodo all'evento sono chiamati **subscribers** (sottoscrittori all'evento).
- Il metodo (o i metodi) sottoscritto all'evento si chiama **event handler** (gestore d'evento).

Gestire un evento, dunque, significa associare un metodo alla variabile *delegate*, in modo che sia eseguito quando viene sollevato l'evento.

² In realtà c'è più di questo.

7.2 Eventi di sola notifica ed eventi con parametro

Gli eventi si possono suddividere in due categorie:

- **eventi di sola notifica**: si limitano a notificare che è accaduto (o sta per accadere) qualcosa, senza fornire informazioni aggiuntive.

`Click` notifica che l'utente ha cliccato sul bottone. `FormClosed` che il *form* è stato chiuso. `TextChanged` che il contenuto del *textbox* è stato cambiato.

- **eventi con parametri**: alla notifica aggiungono i dati necessari a gestire l'evento.

`KeyPress` fornisce il codice del tasto premuto. `MouseMove` la posizione del mouse. Etc.

In ogni caso, il *delegate* dell'evento dovrebbe essere un'*action* che definisce perlomeno il parametro *sender*, che referencia il mittente. (Eccetto se l'evento è sollevato da un metodo statico, poiché in quel caso non esiste un oggetto mittente.)

Benché sia possibile definire dei propri *delegate*, è opportuno impiegare i *delegate* predefiniti, in modo da adottare un modello di programmazione consistente e condiviso dalla comunità dei programmatori.

Sono normalmente utilizzati i *delegate* `EventHandler`, `EventHandler<>` e `Action<>`. I primi sono largamente adottati dai controlli delle interfacce grafiche nelle applicazioni WinForms, WPF e Xamarin.

8 Programmazione event-driven: lista osservabile

La programmazione *event-driven* trova la sua naturale applicazione nelle interfacce grafiche, ma è molto utile anche nella realizzazione di altri tipi di componenti. Di seguito fornisco un esempio estendendo il funzionamento della classe generica `Lista<>`, rendendola una *lista osservabile*.

Si dice *osservabile* un oggetto che notifica i cambiamenti del proprio stato. Una *lista osservabile* notifica quando vengono aggiunti/modificati/rimossi elementi, sollevando un evento ogni qual volta viene eseguito `Add()`, `RemoveAt()`, `Clear()` e il *set accessor* dell'indicizzatore.

8.1 Definire l'evento ListaChanged

Occorre innanzitutto stabilire se notificare soltanto l'avvenuto cambiamento nella lista, oppure fornire anche informazioni sulla modifica effettuata. Perché l'evento sia realmente utile è opportuna la seconda scelta; si tratta dunque di implementare un evento con parametri.

8.1.1 Parametro dell'evento: ListaChangedEventArgs

Un cambiamento nella lista implica un'azione – aggiunta, modifica, etc – e l'elemento coinvolto dal cambiamento. Definisco pertanto i tipi necessari per fornire le suddette informazioni:

```
public class ListaChangedEventArgs
{
    public Action Action; // natura del cambiamento
    public int Index;     // indice dell'elemento coinvolto (-1 se l'azione è Reset)
}

public enum Action
{
    Reset, // tutti gli elementi sono stati coinvolti (metodo Clear())
    Add,   // aggiunto un elemento: Add()
    Remove, // rimosso un elemento: RemoveAt()
    Set    // modificato un elemento: set accessor dell'indicizzatore
}
```

`ListaChangedEventArgs` rappresenta il parametro dell'evento. Il suffisso `EventArgs` rispetta una convenzione largamente adottata e suggerisce che si tratta di un tipo utilizzato da un evento.

8.1.2 Scegliere il delegate

Esistono varie possibilità, compresa quella di definire un proprio *delegate*. In generale, comunque, è prassi comune usare il *delegate* generico `EventHandler<>`, così definito:

```
public delegate void EventHandler<T>(object sender, T e); // T il tipo del parametro evento
```

Il parametro `sender` identifica il mittente, e dunque l'oggetto, che solleva l'evento. Di norma, il codice applicativo usa questo parametro se lo stesso metodo è associato agli eventi di più oggetti e dunque deve stabilire chi, tra questi, ha sollevato l'evento.

8.1.3 Dichiarazione dell'evento (campo delegate)

L'evento è una variabile di tipo `EventHandler<>`. È opportuno scegliere un nome che suggerisca la funzione dell'evento; la desinenza verbale dovrebbe indicare se si tratti di un evento relativo a un processo già eseguito, in corso di esecuzione o ancora da eseguire.

```
public class ListaOsservabile<T>
{
    public event EventHandler<ListaChangedEventArgs> ListaChanged;
    ...
}
```

8.2 Sollevare l'evento

L'evento viene eseguito (sollevato) come qualsiasi variabile *delegate*. Ad esempio, ecco come potrebbe essere riscritto il metodo `RemoveAt()`:

```
public class ListaOsservabile<T>
{
    public event EventHandler<ListaChangedEventArgs> ListaChanged;
    ...
    public void RemoveAt(int index)
    {
        ValidateIndex(index);
        Array.Copy(items, index + 1, items, index, Count - index - 1);
        Count--;

        var e = new ListaChangedEventArgs { Action = Action.Remove, Index = index };
        ListaChanged(this, e); // this (l'oggetto) è il mittente dell'evento
    }
}
```

Ma c'è un problema: il codice funziona se `ListaChanged` riferenzia un metodo, ma questo dipende dal fatto che, nel codice esterno, sia stato effettivamente associato un metodo alla variabile. Non è un presupposto vincolante, perché il codice esterno potrebbe usare la lista senza alcun bisogno di gestire l'evento.

Si tratta di una questione generale: *non si può presupporre che un evento sia stato sottoscritto!*

(Pensa alle decine di eventi implementati dai controlli di un'interfaccia grafica e al fatto che, normalmente, per ogni controllo sono gestiti uno, al massimo due eventi.)

8.2.1 Verificare che l'evento sia stato "sottoscritto"

È sufficiente verificare che la variabile sia diversa da `null`. È possibile farlo in un'unica istruzione:

```
ListaChanged(this, e); // produce un'eccezione se l'evento non è sottoscritto
ListaChanged?.Invoke(this, e); // esegue l'evento soltanto se diverso da null
```

8.2.2 Usare un metodo helper per sollevare l'evento

Poiché il precedente codice – crea il parametro e solleva l'evento – deve essere ripetuto più volte, è utile incapsularlo in un metodo (di seguito mostro come utilizzarlo in `Add()` e `RemoveAt()`):

```
public class ListaOsservabile<T>
{
    public event EventHandler<ListaChangedEventArgs> ListaChanged;
    ...
    public void Add(T value)
    {
        if (Count == items.Length)
        {
            var newCapacity = items.Length == 0 ? 10 : items.Length * 2;
            Array.Resize(ref items, newCapacity);
        }
        items[Count++] = value;
        Raise(Action.Add, Count - 1); // Count-1 è l'indice dell'elemento aggiunto
    }

    public void RemoveAt(int index)
    {
        ValidateIndex(index);
        Array.Copy(items, index + 1, items, index, Count - index - 1);
        Count--;
        Raise(Action.Remove, index);
    }
    ...

    private void Raise(Action action, int index)
    {
        var e = new ListaChangedEventArgs { Action = action, Index = index };
        ListaChanged?.Invoke(this, e);
    }
}
```

8.3 Sottoscrivere all'evento: gestore di evento

Di seguito sottoscrivo un gestore d'evento a una lista di nominativi e quindi eseguo alcune operazioni sulla lista:

```
static void Main(string[] args)
{
    var nomi = new ListaOsservabile<string>();
    nomi.ListaChanged += Nomi_ListaChanged; // sottoscrive Nomi_ListaChanged all'evento

    nomi.Add("Filippo"); // Action->Add Index->0
    nomi.Add("Sara"); // Action->Add Index->1
    nomi.RemoveAt(0); // Action->Remove Index->0
    nomi[0] = "Sonia"; // Action->Set Index->0
    nomi.Clear(); // Action->Reset Index->-1
}
```

```

...
}

private static void Nomi_ListaChanged(object sender, ListaChangedEventArgs e)
{
    Console.WriteLine($"Azione: {e.Action}\tIndice: {e.Index}");
}

```

Nota bene: è prassi nominare il gestore di evento rispettando la convenzione:

```
<mittente>_<nome evento>.
```

8.3.1 Operatore di sottoscrizione all'evento

Considera l'istruzione di sottoscrizione dell'evento:

```
nomi.ListaChanged += Nomi_ListaChanged;
```

`+=` è l'unico operatore ammesso con le variabili evento. Questo operatore, utilizzabile anche con "normali" variabili *delegate*, consente di sottoscrivere più metodi allo stesso evento. Si tratta di un aspetto importante della programmazione *event-driven*, perché consente a più componenti della applicazione di reagire allo stesso evento.

8.4 Conclusioni

Il modello di programmazione *event-driven* consente di far comunicare tra loro i componenti di un'applicazione riducendo, o addirittura azzerando, il livello di accoppiamento tra di essi. In molti scenari semplifica lo sviluppo del programma.

Ad esempio, considera l'idea di un'applicazione grafica che debba gestire una lista di immagini, da visualizzare in una *gallery*. L'applicazione deve eseguire diverse operazioni sulla lista – aggiungere immagini, eliminarle, svuotare la lista – aggiornando immediatamente la *gallery*. In assenza di eventi è necessario, dopo ogni `Add()`, `RemoveAt()` e `Clear()`, eseguire l'aggiornamento della *gallery*, con inevitabile duplicazione del codice di aggiornamento. D'altra parte, usando `ListaOsservabile<>` è sufficiente collocare l'aggiornamento della *gallery* nel gestore dell'evento `ListaChanged` per garantire che sia sempre sincronizzata con lo stato della lista.