Ingegneria del Software T

Delegati ed Eventi

Delegati

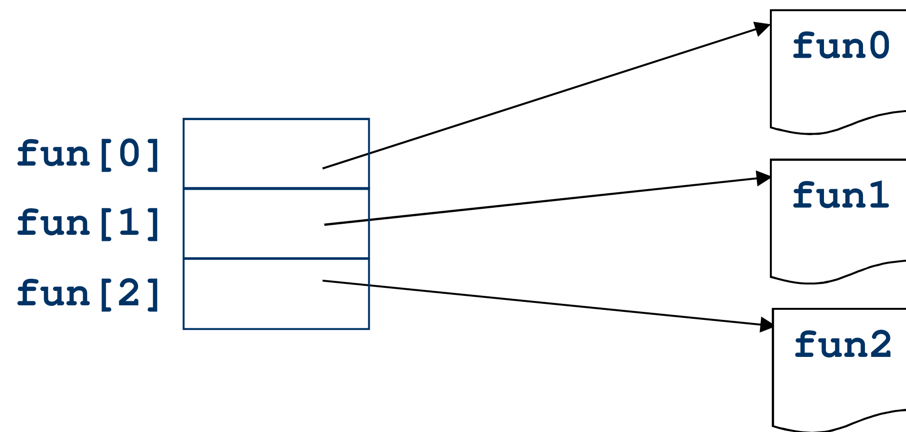
- Sono oggetti che possono contenere **il riferimento** (*type safe*) **a un metodo**, tramite il quale il metodo può essere invocato
- **Oggetti funzione** (*functor*) oggetti che si comportano come una funzione (metodo)
- Simili ai puntatori a funzione del C/C++, ma *object-oriented* e molto più potenti
- Utilizzo standard: funzionalità di ***callback***
  - **Elaborazione asincrona**
  - **Elaborazione cooperativa** (il chiamato fornisce una parte del servizio, il chiamante fornisce la parte rimanente – es. qsort in C)
  - **Gestione degli eventi** (chi è interessato a un certo evento si registra presso il generatore dell’evento, specificando il metodo che gestirà l’evento)

PUNTATORI A FUNZIONI

```
int funX(char c);
int funY(char c);
int (*g)(char c) = NULL;
...
g = cond1 ? funX : funY;
oppure:  g = cond1 ? &funX : &funY;
...
… g('H') …  ≡  … (*g)('H') …
```

C/C++: ARRAY DI PUNTATORI A FUNZIONI

```
void fun0(char *s);
void fun1(char *s);
void fun2(char *s);
void (*fun[])(char *s) =
   { fun0, fun1, fun2 };
...
fun[m]("stringa di caratteri");  ≡
   (*fun[m])("stringa di caratteri");
```



Delegati

- **Dichiarazione** di un nuovo **tipo** di **delegato**
che può contenere il riferimento a un metodo che ha un unico argomento intero e restituisce un intero:

```
delegate int Azione(int param);
```

- **Definizione** di un **delegato**:

```
Azione azione;
```

- **Inizializzazione** di un delegato:

```
azione = new Azione(nomeMetodoStatico);
azione = new Azione(obj.nomeMetodo);
azione = nomeMetodoStatico; // C# 2.0
azione = obj.nomeMetodo;     // C# 2.0
```

- **Invocazione del metodo** referenziato dal delegato:

```
int k1 = azione(10);
```

Esempio3.1

Delegati
Multicasting

- È possibile assegnare al delegato una **lista di metodi**
  All'atto della chiamata del delegato, i metodi vengono chiamati
  – **automaticamente** e
  – **in sequenza**

- Per **aggiungere un metodo** alla lista: **+=**

```
Azione azione = new Azione(Fun1);
… azione(10) …  // Fun1(10)
azione += new Azione(Fun2);
… azione(10) …  // Fun1(10), Fun2(10)
```

- Per **togliere un metodo** dalla lista: **–=**

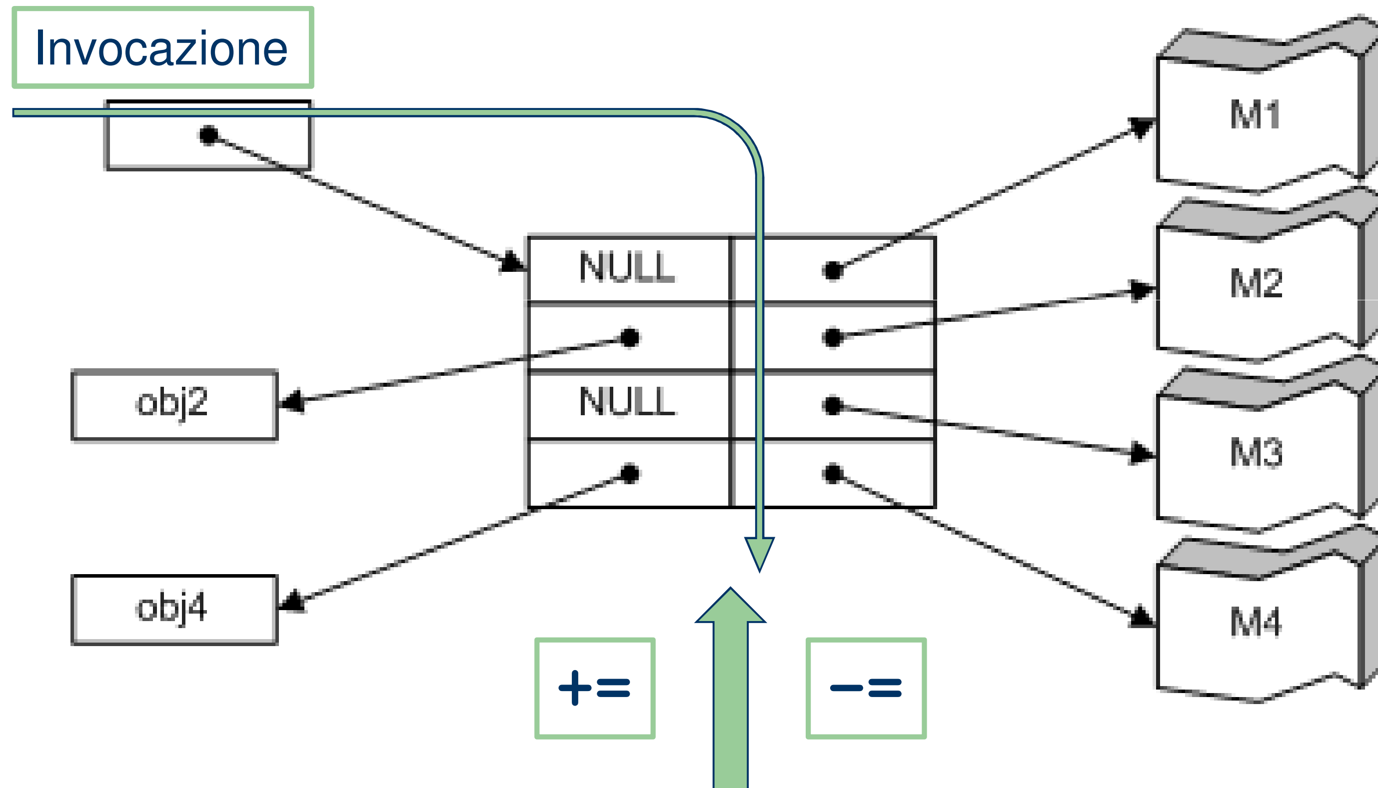
```
azione -= new Azione(Fun1);
… azione(10) …  // Fun2(10)
```

Esempio3.2

Delegati

- A delegate instance encapsulates one or more methods (with a particular set of arguments and return type), each of which is referred to as a **callable entity**
  - for **static methods**, a callable entity consists of just a method
  - for **instance methods**, a callable entity consists of an instance and a method on that instance
- A delegate
  - enforces only a single **method signature** (not a name)
  - does not know or care about the class of the object that it references
- This makes delegates suited for **anonymous invocation**

Delegati

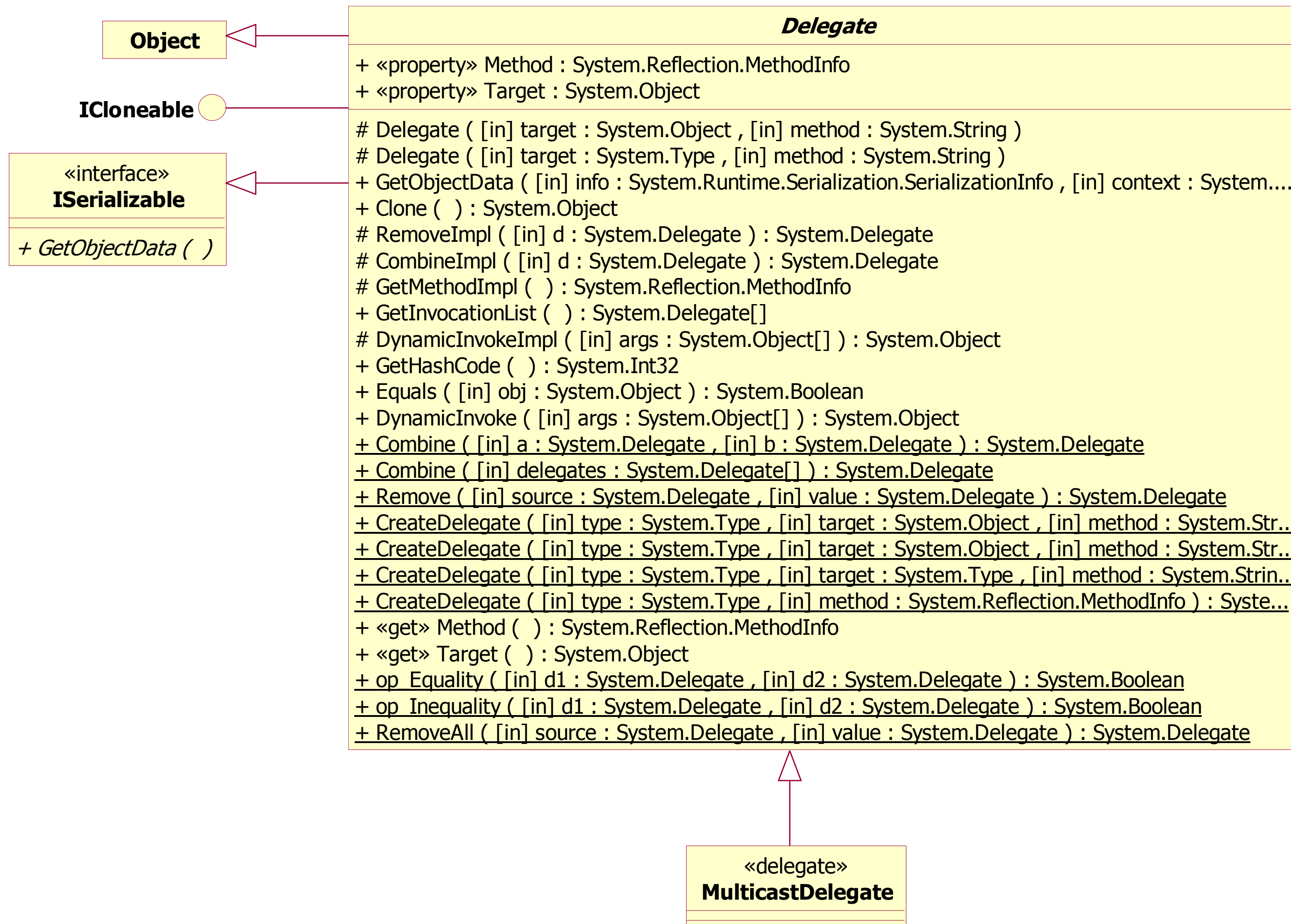


Delegati

- Invocation of a delegate instance whose invocation list contains multiple entries proceeds by invoking each of the methods on the invocation list, **synchronously, in order**
- Each method so called is passed the same set of arguments as was given to the delegate instance
- If such a delegate invocation includes **reference parameters**
  - each method invocation will occur with a reference to the same variable
  - changes to that variable by one method in the invocation list will be visible to methods further down the invocation list
- If the delegate invocation includes **output parameters** or a **return value**
  - their final value will come from the invocation of the last delegate in the list

Esempio3.3

Delegati



Delegati

- In C#, la dichiarazione di un nuovo tipo di delegato definisce automaticamente una nuova classe derivata dalla classe **`System.MulticastDelegate`**

```
System.Object
  System.Delegate
    System.MulticastDelegate
      Azione
```

- Pertanto, sulle istanze di **`Azione`** è possibile invocare i metodi definiti a livello di classi di sistema

Esempio3.4,5,6

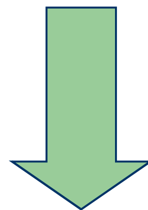
Delegati
Esempio Boss-Worker

- È necessario modellare un'interazione tra due componenti
  - un **Worker** che effettua un'attività (o lavoro)
  - un **Boss** che controlla l'attività dei suoi Worker
- Ogni Worker deve notificare al proprio Boss:
  - quando il lavoro inizia
  - quando il lavoro è in esecuzione
  - quando il lavoro finisce
- Soluzioni possibili:
  - ✓ ***class-based*** *callback relationship*
  - ✓ ***interface-based*** *callback relationship*
  - ✓ ***pattern Observer*** (lista di notifiche)
  - 4. ***delegate-based*** *callback relationship*
  - 5. ***event-based*** *callback relationship*

A delegate-based callback relationship

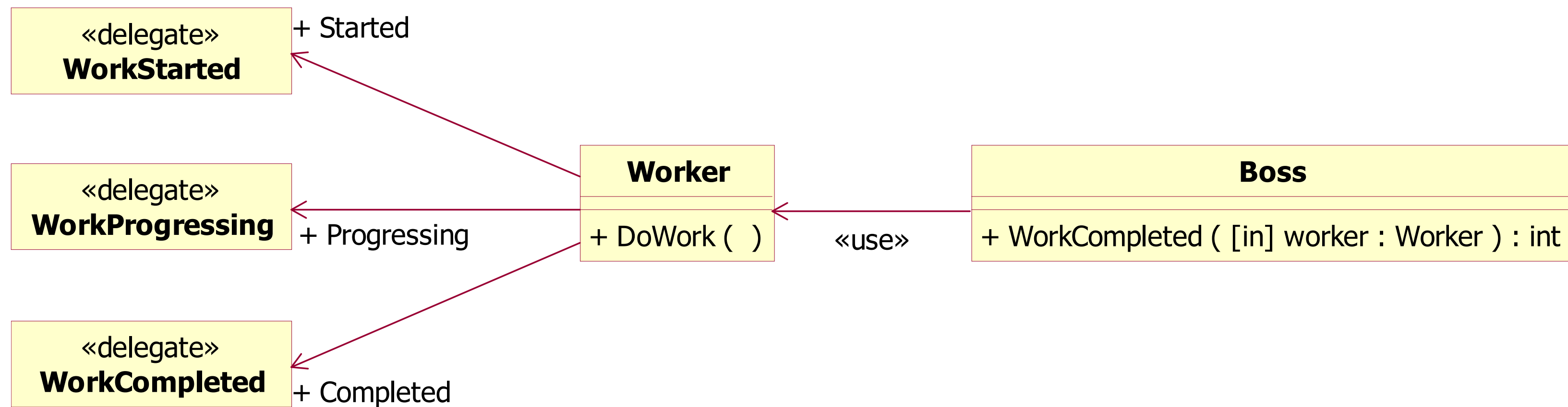
- Un delegato è un'entità *type-safe* che si pone tra 1 *caller* e 0+ *call target* e che agisce come un'interfaccia con un solo metodo

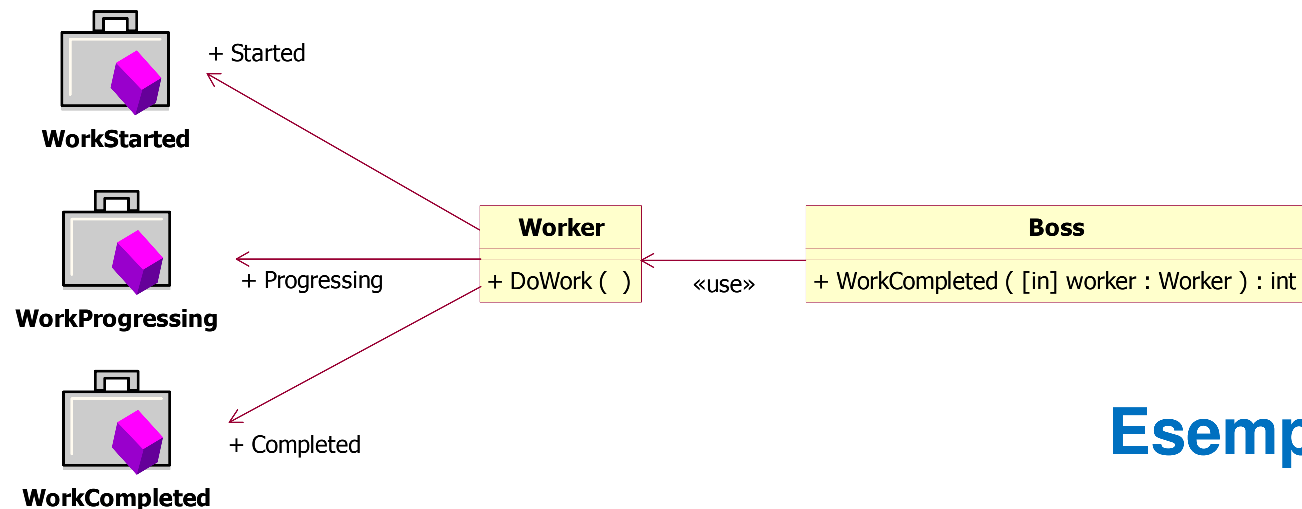
```
interface IWorkerEvents
{
   void WorkStarted(Worker worker);
   void WorkProgressing(Worker worker);
   int WorkCompleted(Worker worker);
}
```



```
delegate void WorkStarted(Worker worker);
delegate void WorkProgressing(Worker worker);
delegate int WorkCompleted(Worker worker);
```

A delegate-based callback relationship





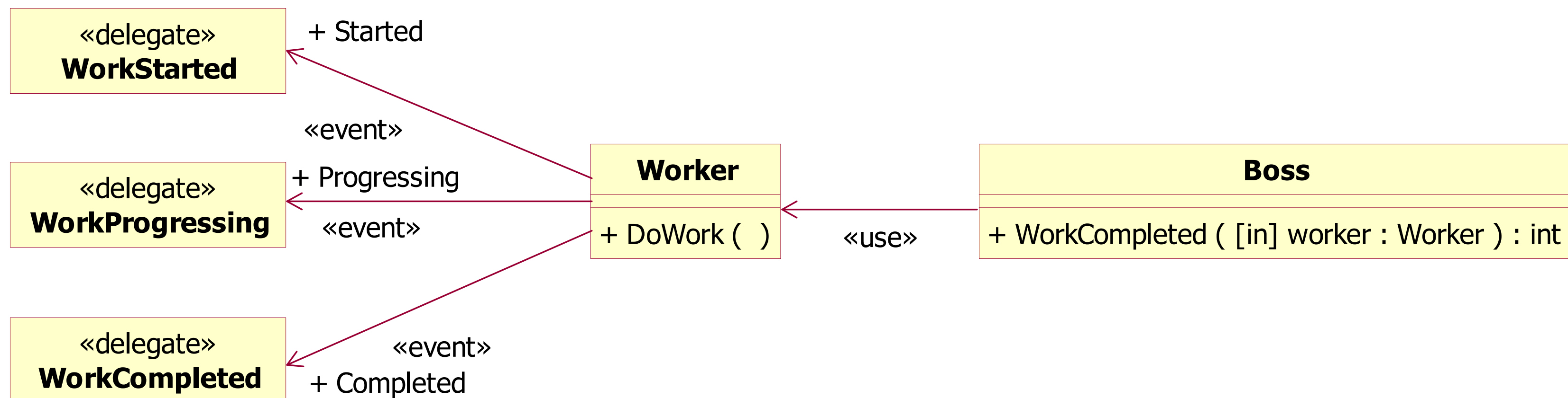
Esempio

Dai delegati agli eventi

- Using public fields for registration offers too much access
  - Client can overwrite previously registered target(s)
    ```
    peter.Started = WorkStarted;
    ```
  - Client can invoke target(s)
    ```
    peter.Completed(peter);
    ```
- Public registration methods coupled with private delegate field is better, but tedious if done manually
- `event` modifier automates support for
  - **public [un]registration** and
  - **private implementation**

An event-based callback relationship

```
class Worker
{
   public event WorkStarted Started;
   public event WorkProgressing Progressing;
   public event WorkCompleted Completed;
   …
}
```



Esempio

Customizing event registration

- User-defined event registration handlers may be provided
  - One benefit of writing your own registration methods is control
  - Alternative property-like syntax supports user-defined registration handlers
  - Allows you to make registration conditional or otherwise customized
  - Client-side access syntax not affected
  - You must provide storage for registered clients

Customizing event registration

```
class Worker
{ …
   public event WorkProgressing Progressing
   {
     add
     {
       if(DateTime.Now.Hour < 12)
       { _progressing += value; }
       else
       { throw new InvalidOperationException
            ("Must register before noon."); }
     }
     remove
     { _progressing -= value; }
   }
   private WorkProgressing _progressing;
   …
}
```

Eventi

- **Evento:** "*Fatto o avvenimento determinante nei confronti di una situazione oggettiva o soggettiva*"
- In programmazione, un evento può essere scatenato
  - dall'interazione con l'utente (click del mouse, …)
  - dalla logica del programma
- ***Event sender*** – l'oggetto (o la classe) che scatena (*raises* o *triggers*) l'evento (sorgente dell'evento)
- ***Event receiver*** – l'oggetto (o la classe) per il quale l'evento è determinante e che quindi desidera essere notificato quando l'evento si verifica (cliente)
- ***Event handler*** – il metodo (dell'*event receiver*) che viene eseguito all'atto della notifica

Eventi

- Quando si verifica l'evento,
  il *sender* invia un messaggio di notifica a tutti i *receiver*
  in pratica, invoca gli *event handler* di tutti i *receiver*
- In genere, il *sender* NON conosce né i *receiver*, né gli *handler*
- Il meccanismo che viene utilizzato per collegare *sender* e *receiver/handler* è il **delegato** (che permette **invocazioni anonime**)

Dichiarazione di un evento

- Un evento incapsula un delegato
  è quindi **necessario dichiarare un tipo di delegato prima di poter dichiarare un evento**
- By convention, event delegates in the .NET Framework have two parameters
  - the **source** that raised the event and
  - the **data** for the event
- Many events, including some user-interface events such as mouse clicks, do not generate event data
- In such situations, the event delegate provided in the class library for the no-data event, **`System.EventHandler`**, is adequate
- Custom event delegates are needed only when an event generates event data

Dichiarazione di un evento

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

```
System.Object
  System.Delegate
    System.MulticastDelegate
      System.EventHandler
```

- La classe **`System.EventArgs`** viene utilizzata quando un evento non deve passare informazioni aggiuntive ai propri gestori
- Se i gestori dell'evento hanno bisogno di informazioni aggiuntive, è necessario derivare una classe dalla classe **`EventArgs`** e aggiungere i dati necessari

Dichiarazione di un evento

```
public event EventHandler Changed;
```

- In pratica, **`Changed`** è un delegato, ma la *keyword* **`event`** ne limita
  - la visibilità e
  - le possibilità di utilizzo

- Una volta dichiarato, l'evento può essere trattato come un delegato di tipo speciale
  in particolare, può:
  - essere **`null`** se nessun cliente si è registrato
  - essere associato a uno o più metodi da invocare

Invocazione di un evento

- Per scatenare un evento è opportuno definire un metodo protetto virtuale **`On`NomeEvento** e invocare sempre quello

```
public event EventHandler Changed;

protected virtual void OnChanged()
{
  if(Changed != null)
    Changed(this, EventArgs.Empty);
}
…
OnChanged();
…
```

- **<u>Limitazione rispetto ai delegati</u>**
L'invocazione dell'evento può avvenire solo all'interno della classe nella quale l'evento è stato dichiarato (benché l'evento sia stato dichiarato **`public`**)

Utilizzo di un evento

- Al di fuori della classe in cui l'evento è stato dichiarato, un evento viene visto come un **delegato con accessi molto limitati**
- Le sole operazioni effettuabili dal cliente sono:
  - **agganciarsi a un evento**: aggiungere un nuovo delegato all'evento mediante l'operatore **+=**
  - **sganciarsi da un evento**: rimuovere un delegato dall'evento mediante l'operatore **−=**

Agganciarsi a un evento

Per iniziare a ricevere le notifiche di un evento, il cliente deve:

- **Definire il metodo** (*event handler*) **che dovrà essere invocato** all'atto della notifica dell'evento (con la stessa *signature* dell'evento):

```
void ListChanged(object sender, EventArgs e)
{ … }
```

- **Creare un delegato** dello stesso tipo dell'evento, farlo riferire al metodo e **aggiungerlo** alla lista dei delegati associati **all'evento**:

```
List.Changed += new EventHandler(ListChanged);
```

```
List.Changed += ListChanged;  //  C# 2.0
```

Sganciarsi da un evento

Per smettere di ricevere le notifiche di un evento, il cliente deve:

- **Rimuovere il delegato** dalla lista dei delegati associati all'evento:

```
List.Changed -= new EventHandler(ListChanged);

List.Changed -= ListChanged;  //  C# 2.0
```

Eventi

- Since += and −= are the only operations that are permitted on an event outside the type that declares the event, external code
  - can add and remove handlers for an event, but
  - cannot in any other way obtain or modify the underlying list of event handlers
- Events provide a generally useful way for objects to signal state changes that may be useful to clients of that object
- Events are an important building block **for creating classes that can be reused in a large number of different programs**

Esempio MVC