Laboratorio di Ingegneria del Software L-A

Delegati ed Eventi

Delegati

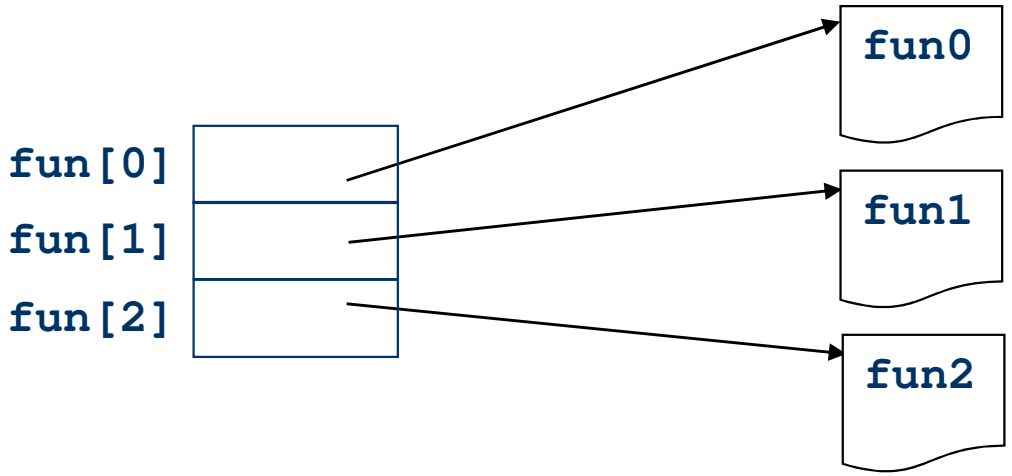
- Sono oggetti che possono contenere
  il riferimento (*type safe*) **a un metodo**,
  tramite il quale il metodo può essere invocato
- **Oggetti funzione** (*functor*)
  oggetti che si comportano come una funzione (metodo)
- Simili ai puntatori a funzione del C/C++,
  ma *object-oriented* e molto più 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)

C/C++
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 Action(int param);
```

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

```
Action action;
```

- **Inizializzazione** di un delegato:

```
action = new Action(nomeMetodoStatico);
…
action = new Action(obj.nomeMetodo);
```

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

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

Delegati
Esempio

```
delegate int Action(int param);

class Class1
{
  static void Main(string[] args)
  {
    Action action;
    action = new Action(Raddoppia);
    Console.WriteLine("Risultato: {0}", action(10));
    action = new Action(Dimezza);
    Console.WriteLine("Risultato: {0}", action(10));
  }
  static int Raddoppia(int x)
  { return x * 2; }
  static int Dimezza(int x)
  { return x / 2; }
}
```

```
Risultato: 20
Risultato: 5
```

Delegati
Esempio

```
delegate int Action(int param);
class Class1
{
  static void Main(string[] args)
  {
    Go(new Action(Raddoppia));
    Go(new Action(Dimezza));
  }
  static void Go(Action action)
  {
    Console.WriteLine("Risultato: {0}", action(10));
  }
  static int Raddoppia(int x)
  { return x * 2; }
  static int Dimezza(int x)
  { return x / 2; }
}
```

```
Risultato: 20
Risultato: 5
```

Delegati
Esempio

- Un delegato può contenere anche il **riferimento a un metodo NON statico** – in questo caso mantiene un riferimento anche all'oggetto su cui invocare il metodo

```
delegate int Action(int param);

class Class1
{
  private int _y;
  public Class1(int y)
  { _y = y; }
  public int Moltiplica(int x)
  { return x * _y; }
  public int Dividi(int x)
  { return x / _y; }
  …
}
```

Delegati
Esempio

```
static void Main(string[] args)
{
  Action action;
  Class1 c = new Class1(5);
  action = new Action(Raddoppia);
  Console.WriteLine("Risultato: {0}", action(10));
  action = new Action(Dimezza);
  Console.WriteLine("Risultato: {0}", action(10));
  action = new Action(c.Moltiplica);
  Console.WriteLine("Risultato: {0}", action(10));
  action = new Action(c.Dividi);
  Console.WriteLine("Risultato: {0}", action(10));
}
```

```
Risultato: 20
Risultato: 5
Risultato: 50
Risultato: 2
```

Delegati

- Esempio2.Step1
  - Delegati anonimi
  - Lambda expressions
  - Reflector
- Esempio2.Step2

Delegati
Multicasting

- Possibilità di creare una **lista di metodi** che vengono chiamati
  - **automaticamente** e
  - **in sequenza**

  all'atto della chiamata del delegato

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

```
Action action = new Action(Fun1);
… action(10) …  // Fun1(10)
action += new Action(Fun2);
… action(10) …  // Fun1(10), Fun2(10)
```

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

```
action -= new Action(Fun1);
… action(10) …  // Fun2(10)
```

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

Delegati
Esempio *multicasting*

```
delegate string Action(ref string param);
class Class1
{
  static string ToUpper(ref string str)
  {
    str = str.ToUpper(); return str;
  }
  static string TrimEnd(ref string str)
  {
    str = str.TrimEnd(); return str;
  }
  static string TrimStart(ref string str)
  {
    str = str.TrimStart(); return str;
  }
  …
```

Delegati
Esempio *multicasting*

```
static void Main(string[] args)
   {
     string s1 = "  abcdefghijk  ";
     Action action =
          new Action(ToUpper) +
          new Action(TrimStart) +
          new Action(TrimEnd);
     Console.WriteLine("s1a: \"" + action(ref s1) + "\"");
     Console.WriteLine("s1b: \"" + s1 + "\"");
   }
}
```

```
s1a: "ABCDEFGHIJK"
s1b: "ABCDEFGHIJK"
```

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
- An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references
- Any object will perfectly do; all that matters is that the method's argument types and return type match the delegate's
- This makes delegates suited for **anonymous invocation**

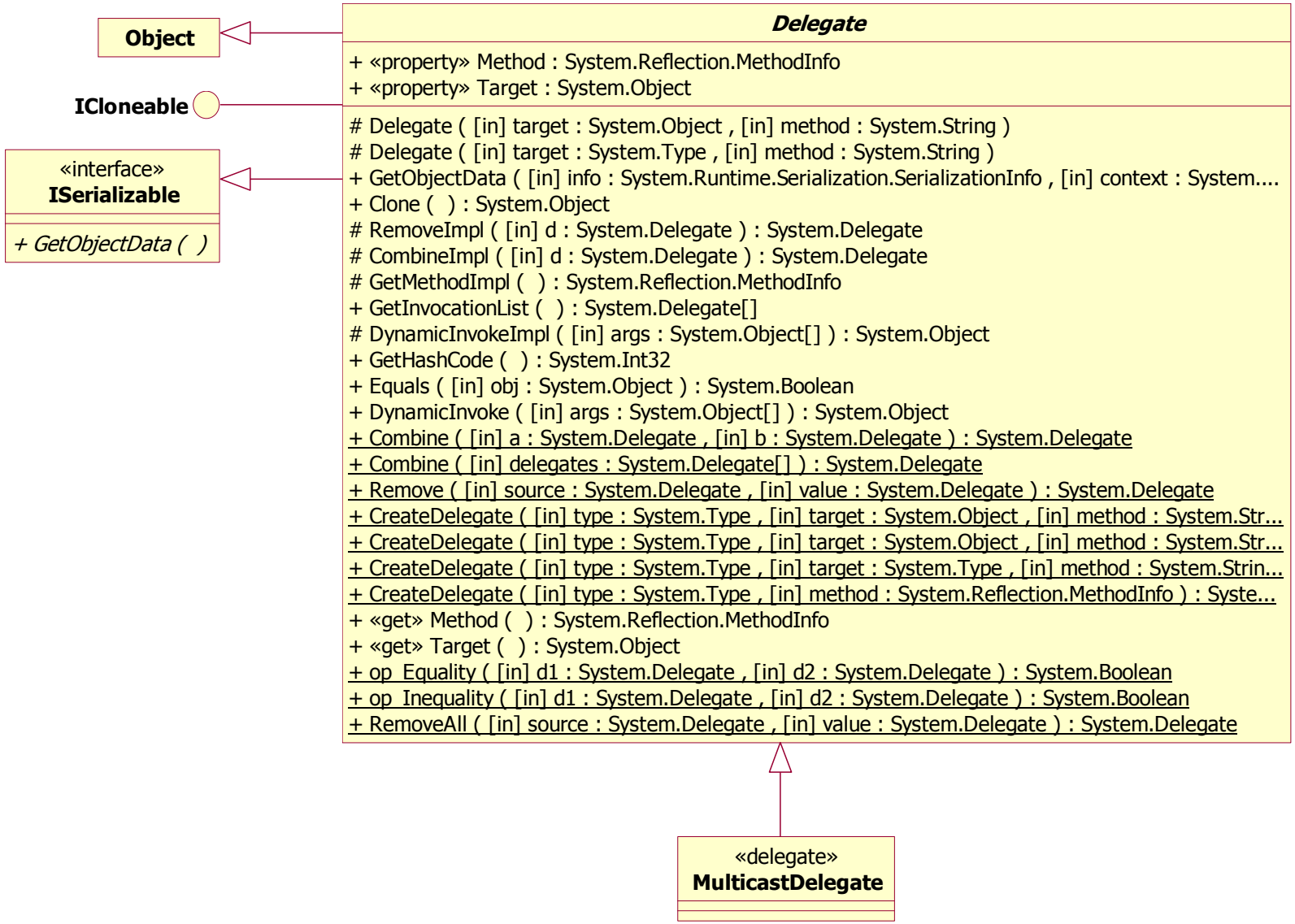
Delegati

- 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
      Action
```

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

Delegati


Object
ICloneable
«interface»
ISerializable
+ GetObjectData ( )
Delegate
+ «property» Method : System.Reflection.MethodInfo
+ «property» Target : System.Object
# Delegate ([in] target : System.Object , [in] method : System.String)
# Delegate ([in] target : System.Type , [in] method : System.String)
+ GetObjectData ([in] info : System.Runtime.Serialization.SerializationInfo , [in] context : System....
+ Clone ( ) : System.Object
# RemoveImpl ([in] d : System.Delegate) : System.Delegate
# CombineImpl ([in] d : System.Delegate) : System.Delegate
# GetMethodImpl ( ) : System.Reflection.MethodInfo
+ GetInvocationList ( ) : System.Delegate[]
# DynamicInvokeImpl ([in] args : System.Object[]) : System.Object
+ GetHashCode ( ) : System.Int32
+ Equals ([in] obj : System.Object) : System.Boolean
+ DynamicInvoke ([in] args : System.Object[]) : System.Object
+ Combine ([in] a : System.Delegate , [in] b : System.Delegate) : System.Delegate
+ Combine ([in] delegates : System.Delegate[]) : System.Delegate
+ Remove ([in] source : System.Delegate , [in] value : System.Delegate) : System.Delegate
+ CreateDelegate ([in] type : System.Type , [in] target : System.Object , [in] method : System.Str...
+ CreateDelegate ([in] type : System.Type , [in] target : System.Object , [in] method : System.Str...
+ CreateDelegate ([in] type : System.Type , [in] target : System.Type , [in] method : System.Strin...
+ CreateDelegate ([in] type : System.Type , [in] method : System.Reflection.MethodInfo) : Syste...
+ «get» Method ( ) : System.Reflection.MethodInfo
+ «get» Target ( ) : System.Object
+ op_Equality ([in] d1 : System.Delegate , [in] d2 : System.Delegate) : System.Boolean
+ op_Inequality ([in] d1 : System.Delegate , [in] d2 : System.Delegate) : System.Boolean
+ RemoveAll ([in] source : System.Delegate , [in] value : System.Delegate) : System.Delegate
«delegate»
MulticastDelegate

Delegati
Esempio

```
Action action =
   new Action(ToUpper) +
   new Action(TrimStart) +
   new Action(TrimEnd);
…
foreach (Action a in action.GetInvocationList())
   Console.WriteLine(a.Method.Name);
```

```
ToUpper
TrimStart
TrimEnd
```

```
foreach (Action a in action.GetInvocationList())
   Console.WriteLine(a.Method.ToString());
```

```
System.String ToUpper(System.String ByRef)
System.String TrimStart(System.String ByRef)
System.String TrimEnd(System.String ByRef)
```

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*
  - ***delegate-based*** *callback relationship*
  - **eventi**

A class-based callback relationship: caller

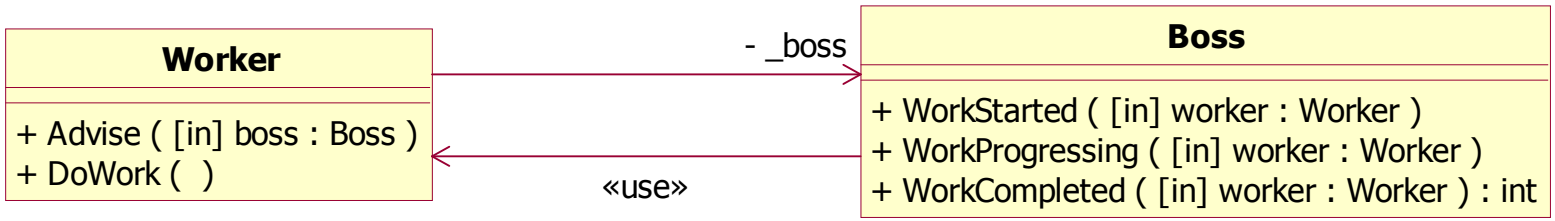
```
class Worker {
   private Boss _boss;
   public void Advise(Boss boss)
   { _boss = boss; }
   public void DoWork()
   {
     Console.WriteLine("Worker: work started");
     if(_boss != null) _boss.WorkStarted(this);
     Console.WriteLine("Worker: work progressing");
     if(_boss != null) _boss.WorkProgressing(this);
     Console.WriteLine("Worker: work completed");
     if(_boss != null)
       {
       int grade = _boss.WorkCompleted(this);
       Console.WriteLine("Worker grade = {0}",grade);
       }
   }
}
```

A class-based callback relationship: target

```
class Boss
{
   public void WorkStarted(Worker worker)
   {
     // Boss doesn't care
   }
   public void WorkProgressing(Worker worker)
   {
     // Boss doesn't care
   }
   public int WorkCompleted(Worker worker)
   {
     Console.WriteLine("It's about time!");
     return 2; // Out of 10
   }
}
```

A class-based callback relationship: registration

```
class Universe
{
   static void Main()
   {
      Worker peter = new Worker();
      Boss boss = new Boss();
      peter.Advise(boss);
      peter.DoWork();
   }
}
```



Interface-based callback relationship

- Interface-based designs are incrementally better than class-based designs for modeling bi-directional relationships
  - More flexible than class-based design
  - Does not constrain implementer's choice of base type
  - As with class-based approach, requires callee to conform to/change type hierarchy

- An interface used for callbacks:

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

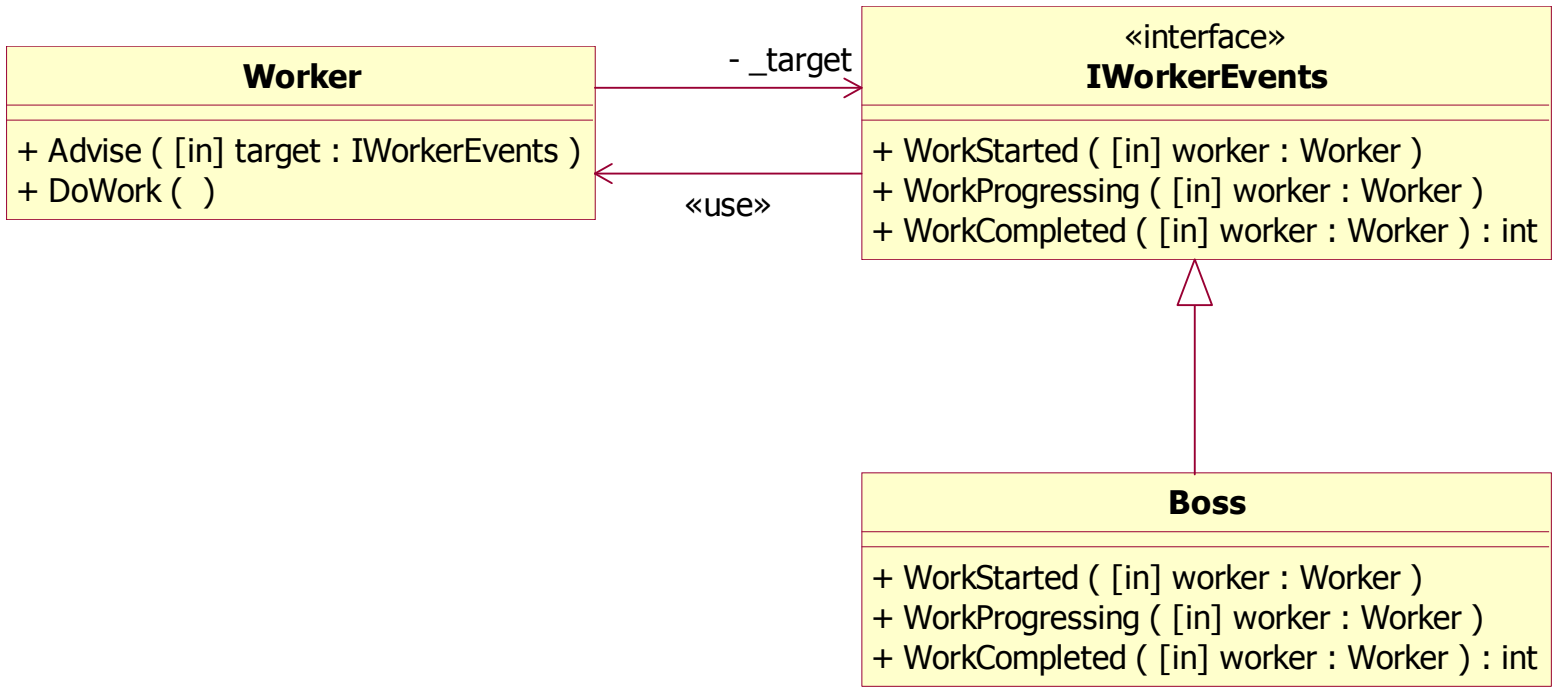
An interface-based callback relationship: caller

```
class Worker {
   private IWorkerEvents _target;
   public void Advise(IWorkerEvents target)
   { _target = target; }
   public void DoWork()
   {
     Console.WriteLine("Worker: work started");
     if(_target != null) _target.WorkStarted(this);
     Console.WriteLine("Worker: work progressing");
     if(_target != null) _target.WorkProgressing(this);
     Console.WriteLine("Worker: work completed");
     if(_target != null)
       {
       int grade = _target.WorkCompleted(this);
       Console.WriteLine("Worker grade = {0}",grade);
       }
   }
}
```

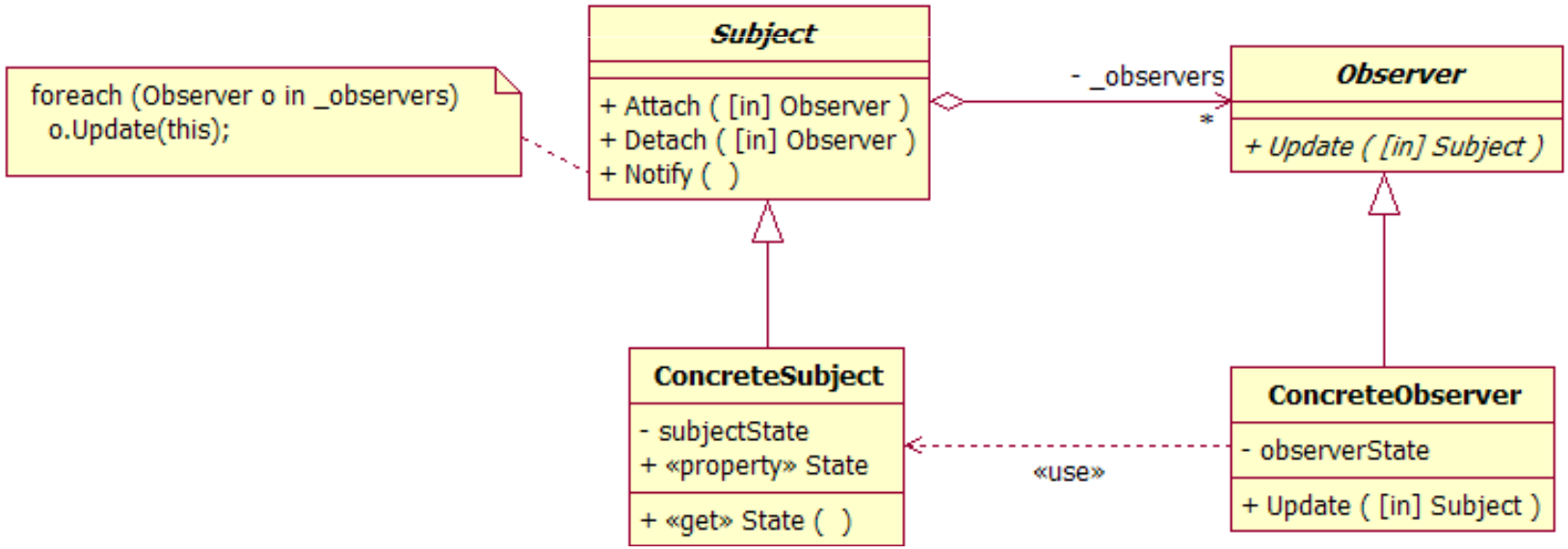
An interface-based callback relationship: target

```
class Boss : IWorkerEvents
{
   public void WorkStarted(Worker worker)
   {
     // Boss doesn't care
   }
   public void WorkProgressing(Worker worker)
   {
     // Boss doesn't care
   }
   public int WorkCompleted(Worker worker)
   {
     Console.WriteLine("It's about time!");
     return 4; // Out of 10
   }
}
```

An interface-based callback relationship



Pattern OBSERVER

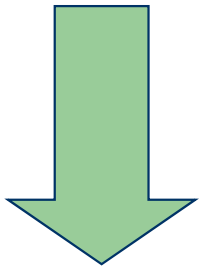


A delegate-based callback relationship

- Un delegato è un'entità *type-safe* riconosciuta e gestita dal CLR che si pone tra 1 *caller* e 0+ *call target*
  - Do not require type compatibility like classes/interfaces
  - Enforce only a single method signature (not a name)
  - Act like a tiny interface with one method
  - Facilitate component integration without source code access
  - Support multiple call targets
  - Support asynchronous method invocation

A delegate used for callbacks

```
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);
```

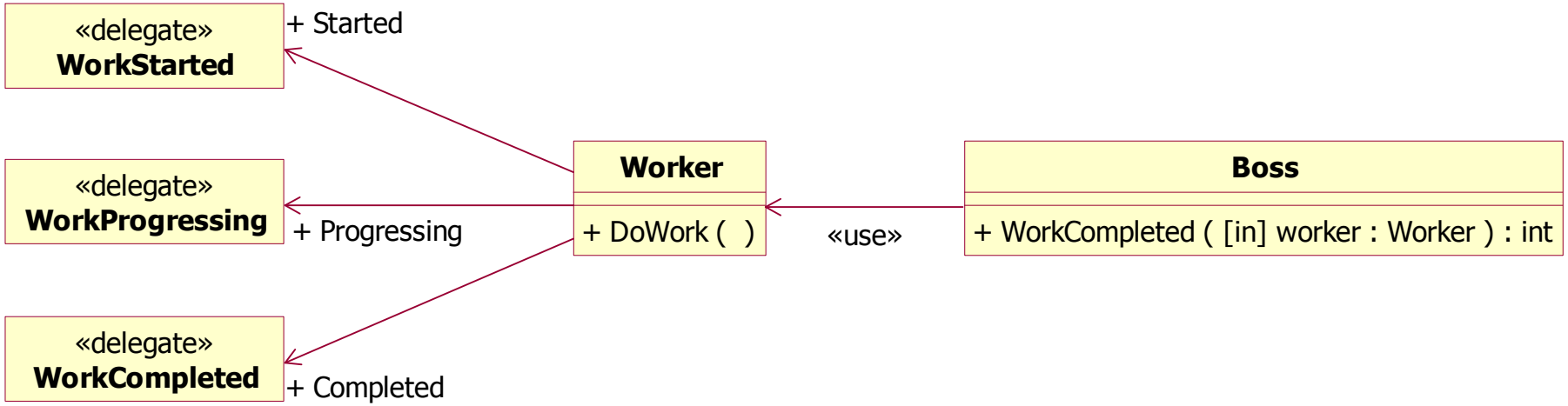
A delegate-based callback relationship: caller

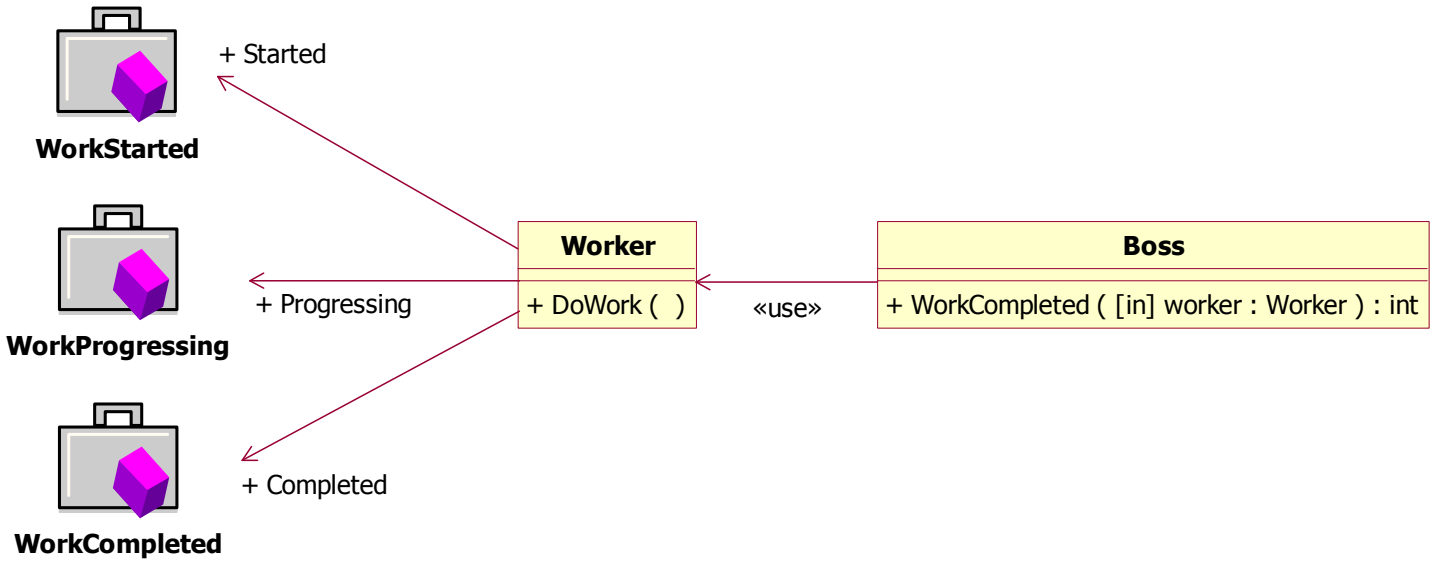
```
class Worker {
   public WorkStarted Started;
   public WorkProgressing Progressing;
   public WorkCompleted Completed;
   public void DoWork()
   {
     Console.WriteLine("Worker: work started");
     if(Started != null) Started(this);
     Console.WriteLine("Worker: work progressing");
     if(Progressing != null) Progressing(this);
     Console.WriteLine("Worker: work completed");
     if(Completed != null)
       {
       int grade = Completed(this);
       Console.WriteLine("Worker grade = {0}",grade);
       }
   }
}
```

A delegate-based callback relationship: target

```
class Boss
{
   public int WorkCompleted(Worker worker)
   {
     Console.WriteLine("Better...");
     return 6; // Out of 10
   }
}
```

A delegate-based callback relationship

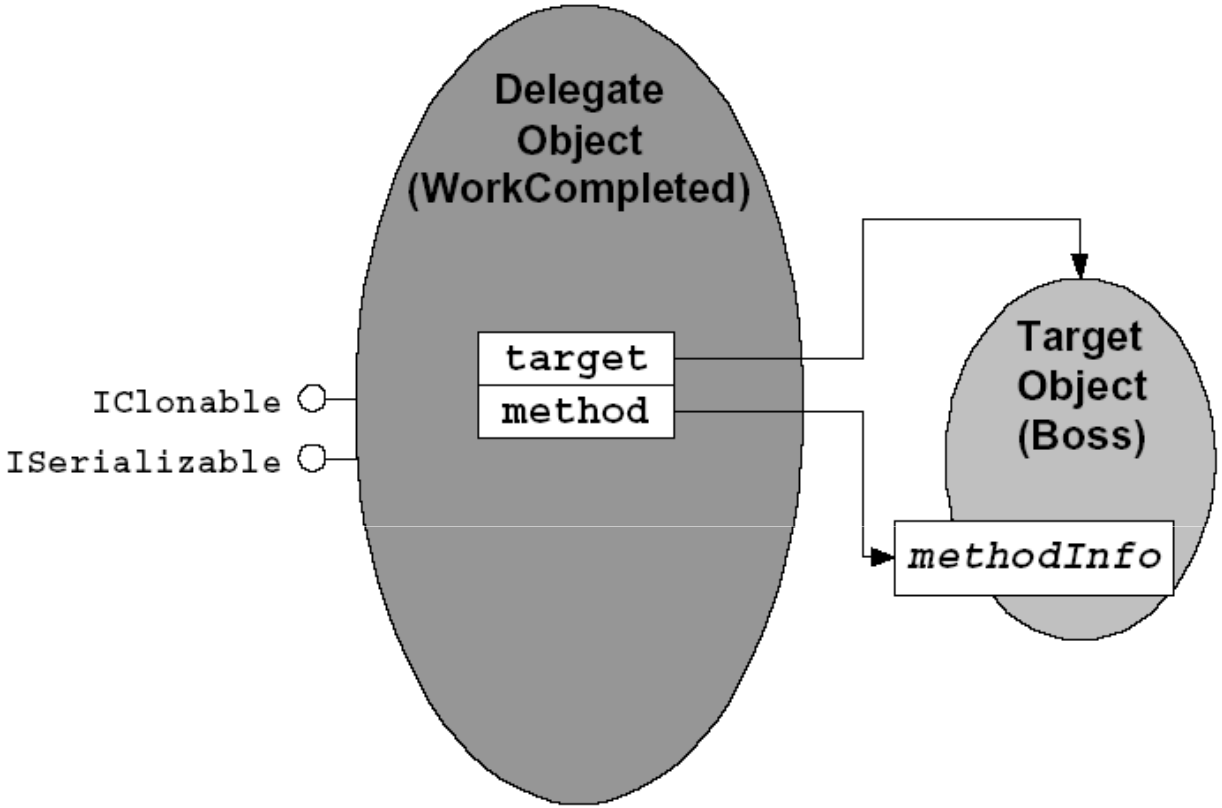




A delegate-based callback relationship: registration

```
class Universe
{
   static void Main()
   {
     Worker peter = new Worker();
     Boss boss = new Boss();
     peter.Completed =
        new WorkCompleted(boss.WorkCompleted);
     peter.DoWork();
   }
}
```

A delegate-based callback relationship



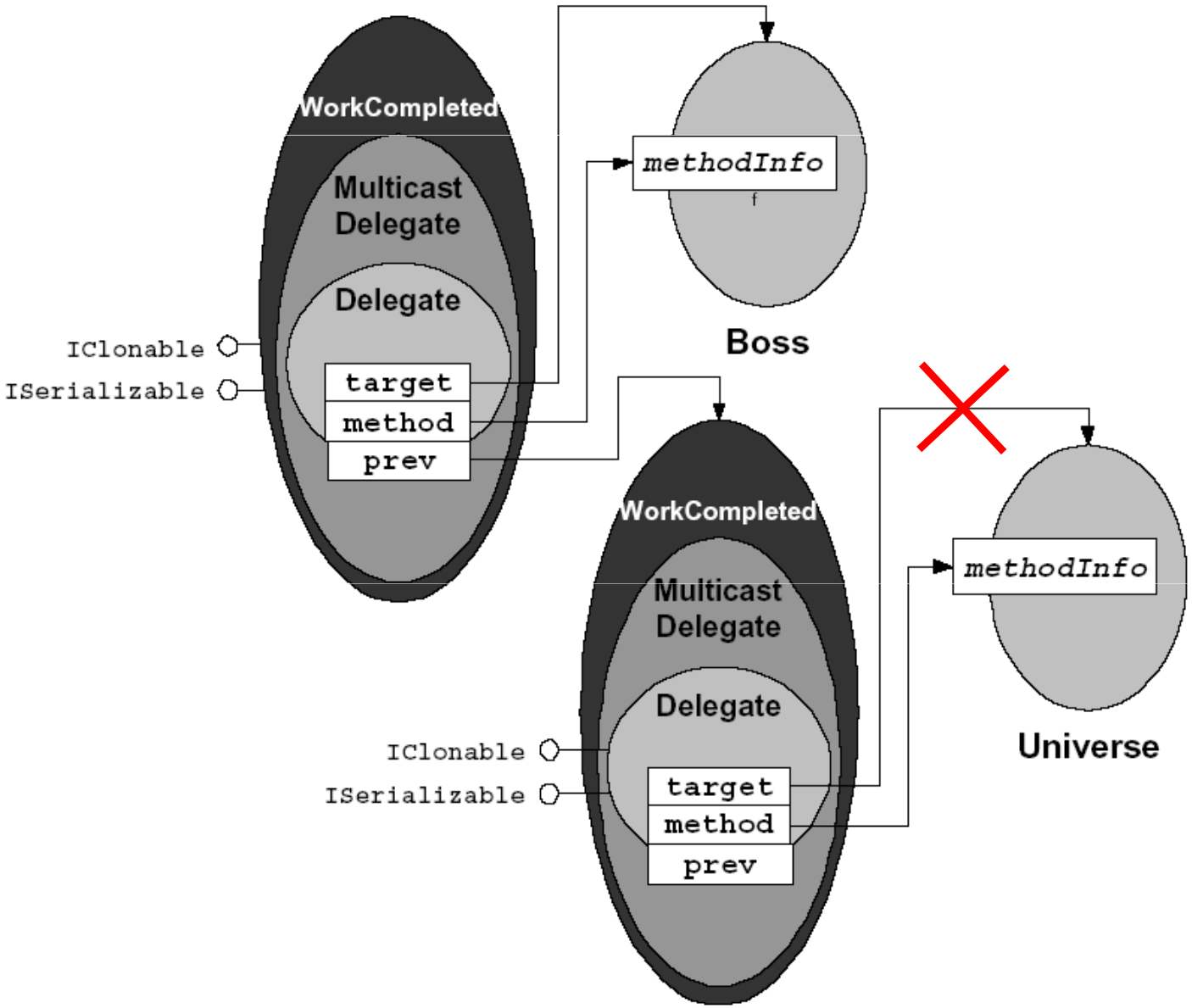
Multiple target registration

```
class Universe
{
   static void WorkerStartedWork(Worker worker)
   {
     Console.WriteLine
        ("Universe notices working starting");
   }
   static int WorkerDoneWorking(Worker worker)
   {
     Console.WriteLine
        ("Universe notices is worker done");
     return 7;
   }
   static void Main()
   {
     …
   }
}
```

Multiple target registration

```
Worker peter = new Worker();
Boss boss = new Boss();

//  Istruzioni possibili ma da evitare:
peter.Completed = new WorkCompleted(boss.WorkCompleted);
peter.Started = new WorkStarted(WorkerStartedWork);
peter.Progressing = null;
peter.Completed(peter);

//  Preferred registration syntax:
peter.Completed += new WorkCompleted(WorkerDoneWorking);

peter.DoWork();
```

A multicast delegate



Iterating through registered targets

```
class Worker
{
   public void DoWork()
   {
      // Start and progress with work...
      Console.WriteLine("Worker: work completed");
      if(Completed != null)
      {
         foreach (WorkCompleted wc in Completed.GetInvocationList())
         {
            int grade = wc(this);
            Console.WriteLine("Worker grade = {0}",grade);
         }
      }
   }
   …
}
```

Dai delegati agli eventi

- Using public fields for registration offers too much access
  - Client can overwrite previously registered target(s)
  - Client can invoke target(s)
- 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

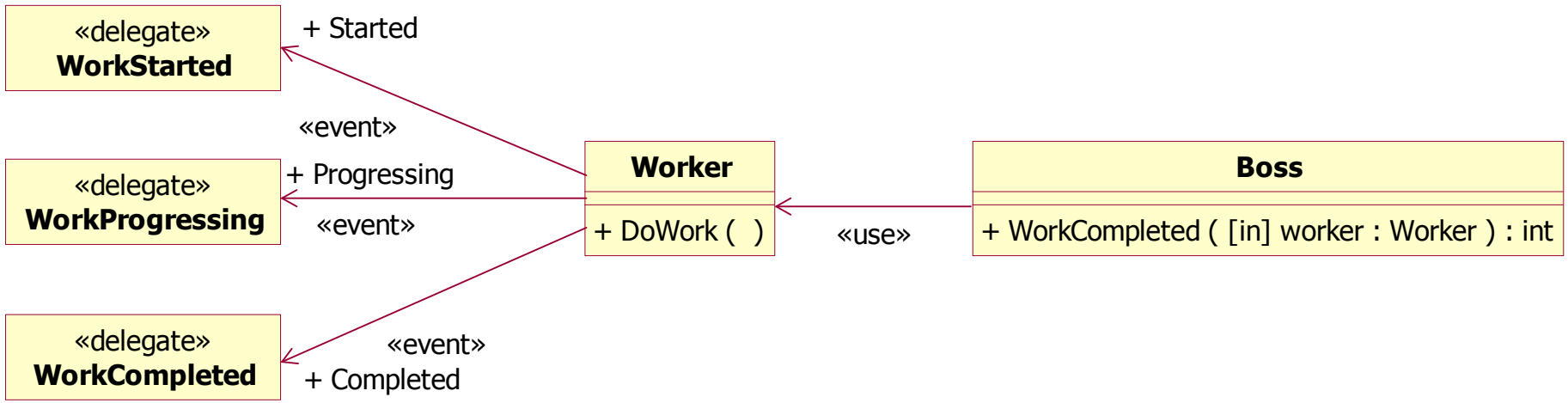
An event-based callback relationship: caller

```
class Worker {
   public event WorkStarted Started;
   public event WorkProgressing Progressing;
   public event WorkCompleted Completed;
   public void DoWork()
   {
     Console.WriteLine("Worker: work started");
     if(Started != null) Started(this);
     Console.WriteLine("Worker: work progressing");
     if(Progressing != null) Progressing(this);
     Console.WriteLine("Worker: work completed");
     if(Completed != null)
       {
       int grade = Completed(this);
       Console.WriteLine("Worker grade = {0}",grade);
       }
   }
}
```

An event-based callback relationship: registration

```
Worker peter = new Worker();
Boss boss = new Boss();

// Illegal: assignment to event field not supported:
peter.Completed = new WorkCompleted(boss.WorkCompleted);
peter.Started = new WorkStarted(WorkerStartedWork);
peter.Progressing = null;

// Illegal: execution of event not supported:
peter.Completed(peter);

// Registration still supported:
peter.Completed += new WorkCompleted(WorkerDoneWorking);

peter.DoWork();
```

An event-based callback relationship



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
- Custom event delegates are needed only when an event generates event data
- 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

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__ è invocare sempre quello

```
public event EventHandler Changed;

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

- **Limitazione rispetto ai delegati**
  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**)

Aggangiarsi a un evento
(hooking up to an event)

- 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 sono:
  - Aggiungere un nuovo delegato alla lista dei delegati mediante l'operatore **+=**
  - Rimuovere un delegato dalla lista dei delegati mediante l'operatore **−=**

Aggangiarsi a un evento
(hooking up to an event)

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

- **Definire il metodo** (*event handler*) **che verrà 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 e farlo riferire al metodo:

```
EventHandler listChangedHandler =
  new EventHandler(ListChanged);
```

- **Aggiungere il delegato** alla lista dei delegati associati **all'evento**, utilizzando l'operatore **+=**:

```
List.Changed += listChangedHandler;
```

Sgangiarsi 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, utilizzando l'operatore **-=**:

```
List.Changed -= listChangedHandler;
```

- Per aggiungere e rimuovere un delegato dalla lista dei delegati associati all'evento si può anche scrivere:

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

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