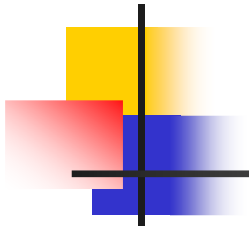
Tecniche di Programmazione avanzata

Corso di Laurea Specialistica in Ingegneria Telematica
Università Kore – Enna – A.A. 2009-2010

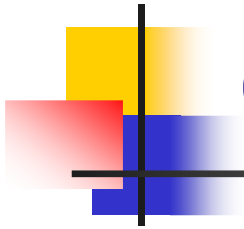
Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

Il linguaggio C#
Delegati



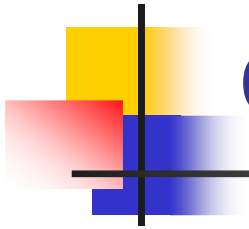
Delegati: motivazioni

- Quando è necessario modellare interazioni tra componenti, classi e interfacce non sempre sono sufficienti
  - Bi-directional caller/callee relationship often desired
  - Hard-coding target class type in caller not a generic solution
  - Shared base class approach not flexible in world of single inheritance
  - Widely used throughout the .NET framework



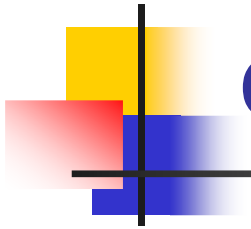
A class-based callback relationship: caller

```
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();
                Console.WriteLine("Worker: work progressing");
                if(_boss != null) _boss.WorkProgressing();
                Console.WriteLine("Worker: work completed");
                if(_boss != null) {
                        int grade = _boss.WorkCompleted();
                        Console.WriteLine("Worker grade =
{0}",grade);
                }
        }
```



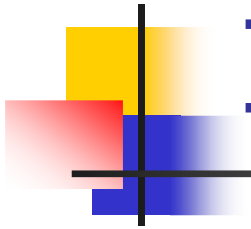
A class-based callback relationship: target

```
class Boss  {
        public void WorkStarted() {
                // Boss doesn't care
        }

        public void WorkProgressing() {
                // Boss doesn't care
        }

        public int WorkCompleted() {
                Console.WriteLine("It's about time!");
                return 2; // Out of 10
        }
}
```



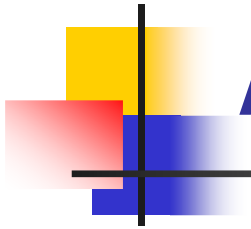
A class-based callback relationship: registration

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



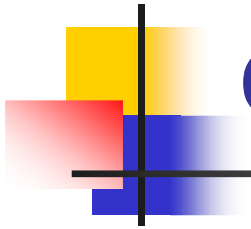
Interface based callback

- Interface-based designs è piu' adeguato del class-based designs per modellare relazioni bidirezionali
  - Più flessibile che class-based design
  - Non costringe l'implementatore a fare scelte di tipi base
  - Da mandato per l'implementazione completa di tutti i metodi



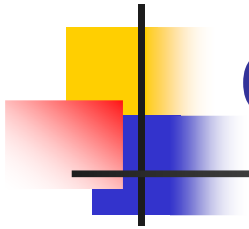
An interface used for callbacks

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



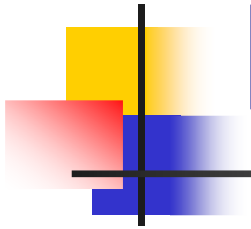
A interface-based callback relationship: caller

```
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();
                Console.WriteLine("Worker: work progressing");
                if(_target != null) _target.WorkProgressing();
                Console.WriteLine("Worker: work completed");
                if(_target != null) {
                        int grade = _target.WorkCompleted();
                        Console.WriteLine("Worker grade =
{0}",grade);
```



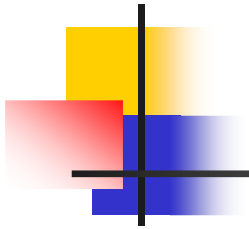
A interface-based callback relationship: target

```
class Boss : IWorkerEvents {
        public void WorkStarted() {
                // Boss doesn't care
        }

        public void WorkProgressing() {
                // Boss doesn't care
        }

        public int WorkCompleted() {
                Console.WriteLine("It's about time!");
                return 4; // Out of 10
        }}
```



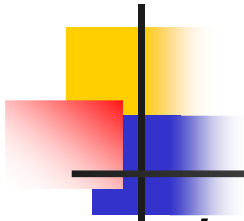
Delegati

- 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



Delegati

- In C#, un delegato è un oggetto che può fare riferimento a un metodo
  - ricorda il "puntatore a funzione" del C…
  - Un delegato serve per chiamare il metodo ad esso associato
  - l'indirizzo del delegato è l'entry point del metodo
  - il metodo da chiamare è deciso a run-time: se si cambia il metodo associato al delegato, si chiama un metodo diverso
- Sintassi: *<delegate tipoRitorno nome(parametri);>*
- A un delegato possono essere associati solo metodi di signature corrispondente a quella dichiarata.
- Esempio: 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);*



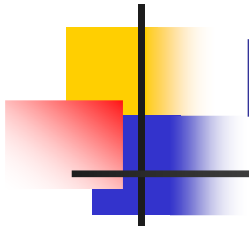
Delegati

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:
action(10);



Delegati: livello di dichiarazione

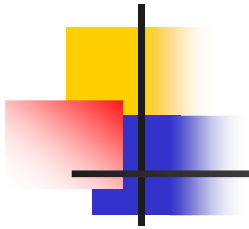
- Per C#, un delegato è un'entità allo stesso livello delle classi
  - la keyword delegate introduce un'entità assimilabile a una classe di oggetti, specifici per l'invocazione indiretta di funzioni che rispettano una ben precisa signature.
- Esempi di definizione:
- delegate int Transformer(int nomeParametro);
- delegate float Operation(float nomeParam);

- NB: i nomi dei parametri vengono ignorati, tuttavia sono sintatticamente indispensabili nella signature; ometterli causa errore di compilazione.

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; }
}
```

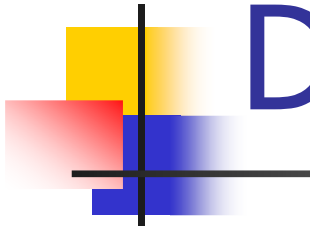
Risultato: 20
Risultato: 5



Delegati: Esempio

- 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

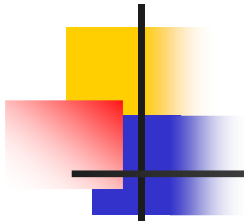
```
delegate int Action(int param);
class Class1 {
   int _y;
   Class1(int y) { _y = y; }
   int Moltiplica(int x) { return x * _y; }
   int Dividi(int x) { return x / _y; }
...
}
```



Delegati: esempio

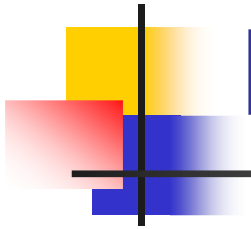
```
static void Main(string[] args) {

        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



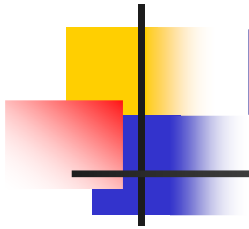
Delegati: esempio

```
delegate int Elaboration(int q);
class Test {
        static int calc1(int s){ return s/2; }
        static int calc2(int s){ return s*s-3; }
        public static void Main(){
                Elaboration t = new Elaboration(calc1);
                int s = t(18); // 9
                t = new Elaboration(calc2); // cambia metodo
puntato
                s = t(18); // 321
                Counter c = new Counter(); // una istanza!
                t = new Elaboration(c.inc); // cambia metodo
puntato
                s = t(18); // 19
        }
}
```



Delegati e Multicasting

- Un delegato è molto utile per il multicasting, in modo che una singola chiamata (al delegato) si traduca in una serie di chiamate a vari metodi.
- In pratica, a un delegato è associata una lista di metodi
- per aggiungere un metodo alla lista si usa l'operatore +=
- per togliere un metodo dalla lista si usa l'operatore -=
- Gli operatori += e -= aggiungono/rimuovono dalla lista associata a un delegato un altro delegato:
- t += new Transformer(nuovoMetodo);
- I metodi associati a un delegato sono invocati nell'ordine esatto in cui sono stati aggiunti al delegato.



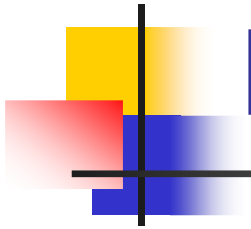
Delegati e Multicasting

- Possibilità di creare una lista di metodi che vengono chiamati automaticamente e in sequenza all'atto della chiamata del delegato

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

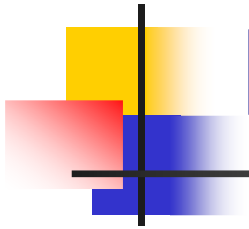
- Per togliere un metodo dalla lista: -=

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



Delegati e Multicasting

- Quando un delegato è usato per il multicasting, è molto opportuno che il suo tipo di ritorno sia void.
- Infatti,
  - tipi di ritorno non-void sono sintatticamente leciti, ma i valori restituiti dai primi N-1 metodi vanno persi, perché in una catena di chiamate (non eseguite da noi!) non c'è modo di recuperarli: resta solo l'ultimo.
- NB: ove i parametri siano riferimenti e debbano essere modificati, dovranno essere ovviamente passati per riferimento (tramite ref o out).
- È questo il caso, ad esempio, delle stringhe, che, essendo oggetti non modificabili, vengono in realtà sempre ri-allocate (a indirizzi ovviamente via via diversi) dopo ogni "modifica"

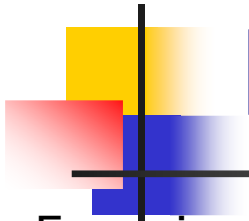


Delegati e Multicasting

Esempio con conservazione del valore attraverso le chiamate:

```
delegate void Transformer(ref int k); // nome k irrilevante
class TestMultiCast1 {
        static void trasf1(ref int s){ s = s/2; }
        static void trasf2(ref int s){ s = s*s-3; }
        public static void Main(){
                Transformer t = new Transformer(trasf1);
                int x = 18;
                t(x); // trasf1(18) = 9
                x = 18; // reinizializzo x a 18
                t += new Transformer(trasf2);
                                        // aggiungo anche trasf2
                t(x); // trasf2(trasf1(18)) = 81-3 = 78
        }}
```

Prima esegue trasf1 (primo in lista), poi trasf2 (secondo in lista) 18 diventa 9, poi 9 diventa 81-3 = 78

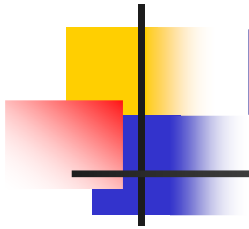


Delegati e Multicasting

Esempio con perdita del valore attraverso le chiamate:

```
delegate int Operation(int q); // nome q irrilevante
class TestMultiCast2 {
        static int calc1(int s){ return s/2; }
        static int calc2(int s){ return s*s-3; }
        public static void Main(){
                Operation t = new Operation(calc1);
                int res, x = 18;
                res = t(x); // res = calc1(18) = 9
                t += new Operation(calc2); // aggiungo anche trasf2
                res = t(x); // !!!!
        }
}
```

Prima esegue trasf1 (primo in lista), poi trasf2 (secondo in lista), ma perde il primo risultato fa solo trasf2 18 diventa 9, poi sempre 18 diventa 324 - 3 = 321



Delegati e Multicasting

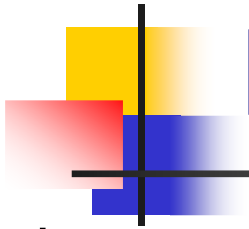
- Usare per il multicasting un delegato con tipo di ritorno non-void è possibile, ma occorre combinarlo con il passaggio per riferimento (tramite ref o out)
- Infatti, in tal modo il passaggio per riferimento assicura la "trasmissione in avanti" dei risultati intermedi (che così non vanno persi) mentre l'istruzione return recupera il valore dell'ultima chiamata in modo funzionale.
- Questo approccio non è consigliabile; può tuttavia essere utile in alcune situazioni particolari (ad esempio, ove occorra avere necessariamente una notazione funzionale, per poter utilizzare il risultato entro espressioni).
- In definitiva, l'uso di return è ortogonale al passaggio per riferimento; il primo si usa se serve una notazione funzionale, il secondo è necessario se (come è lecito presumere) le varie funzioni devono comunicare fra loro e passarsi l'una il risultato dell'altra (necessità che il return NON soddisfa)

Delegati e Multicasting

Altro Esempio in notazione funzionale:

```
delegate string Action(ref string st); // nome st irrilevante
class TestMultiCast3 {
        static string toUp(ref string s) { return s=s.ToUpper();}
        static string trimH(ref string s){ return s=s.TrimStart();}
        static string trimT(ref string s){ return s=s.TrimEnd();}
        public static void Main(){
                Action action =
                new Action(toUp) + new Action(trimH) + new Action(trimT);
                string s1 = " abcdefghijk ";
                Console.WriteLine("s1: |"" + action(ref s1) + ""|"");
        }
}
```

Prima esegue toUp, poi trimH, poi trimT e non perde le trasformazioni intermedie perché st passa anche per riferimento; l'ultima return in trimT restituisce s1 per poter usare l'espressione + entro WriteLine.

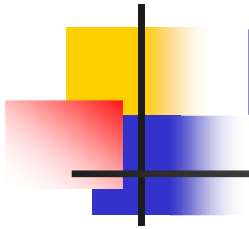


Delegati e Multicasting

Lo stesso esempio riscritto in notazione procedurale:

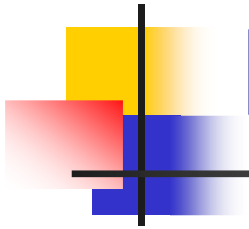
```
delegate void Action(ref string st); // nome st irrilevante
class TestMultiCast3 {
        static void toUp(ref string s) {s=s.ToUpper();} // no return
        static void trimH(ref string s){s=s.TrimStart();} // no return
        static void trimT(ref string s){s=s.TrimEnd();} // no return
        public static void Main(){
            Action action =
            new Action(toUp) + new Action(trimH) + new
Action(trimT);
            string s1 = " abcdefghijk ";
            action(ref s1);
            Console.WriteLine("s1: |"" + s1 + ""|""); }}
```

La notazione funzionale è stata eliminata rendendo void il delegato e quindi le tre funzioni toUp, trimH, trimT. L'azione action è perciò invocata in modo procedurale, usando poi s1 nell'espressione +



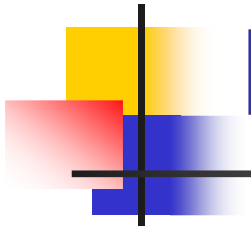
Delegati e Multicasting

- "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 e Multicasting

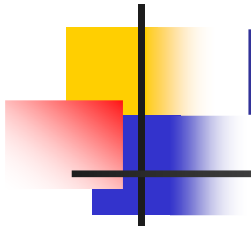
```
delegate string Action(ref string param);

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 e Multicasting

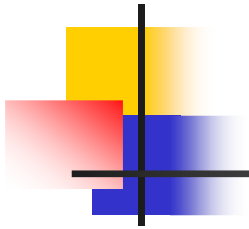
```
static void Main(string[] args) {

        string s1 = " abcdefghijk ";
        Action action =
        new Action(ToUpper) +
        new Action(TrimStart) +
        new Action(TrimEnd);
        Console.WriteLine("s1: |"" + action(ref s1) +
"|"");
        }
}
```

s1: "ABCDEFGHIJK"



Delegati e Multicasting

- "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 do; all that matters is that the method's argument types and return type match the delegate's. **This makes delegates perfectly 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:

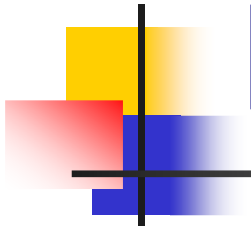
System.Object

System.Delegate

System.MulticastDelegate

Action

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



Delegati

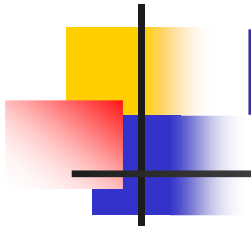
Esempio di invocazione di un metodo ordinario su un delegato:

```
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
```

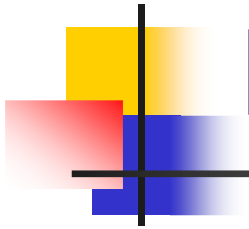
```
foreach (Action a in action.GetInvocationList())
        Console.WriteLine(a.Method.ToString());
```

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



Funzionamento Delegati

- Un delegato è una astrazione di uno o più puntatori a funzione derivati da System.MulticastDelegate
- Una istanza è un oggetto che racchiude un puntatore a funzione ed un oggetto target: quando il delegato è chiamato viene chiamato il metodo corrispondente di target
- Un delegato ha una signature e un valore di ritorno (type-safe)
- La classe Delegate è la classe base per i tipi delegati.
- Tuttavia, soltanto il sistema e i compilatori possono derivare in modo esplicito dalla classe Delegate o dalla classe MulticastDelegate.
- Non è inoltre consentita la derivazione di un nuovo tipo da un tipo delegato.
- La classe Delegate non è considerata un tipo delegato, in quanto classe utilizzata per derivare i tipi delegati.



Funzionamento Delegati

- Dichiarazione di un metodo delegato

  delegate void Notifier (string sender);
- Dichiarazione di una variabile delegata

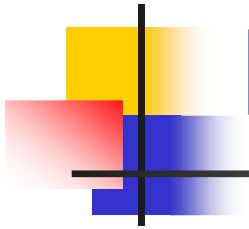
  Notifier greetings;
- Assegnare un metodo ad una variabile

  void SayHello(string sender) {

  Console.WriteLine("Hello from " + sender); }

  greetings = new Notifier(SayHello);
- Invocazione di una variabile delegata

  greetings("John");



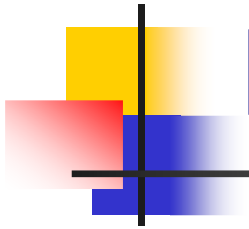
Funzionamento Delegati

- Qualsiasi metodo compatibile può essere assegnato ad un delegato

```
void SayGoodBye(string sender) {
        Console.WriteLine("Good bye from " + sender);
}
greetings = new Notifier(SayGoodBye);
greetings("John");
// SayGoodBye("John") => "Good bye from John"
```

Note

- Un delegato può valere null (nessun metodo assegnato)
- se *null* , un delegato non può essere invocato
- I delegati possono essere memorizzati in una struttura dati, passati come parametro, ecc.



Funzionamento Delegati

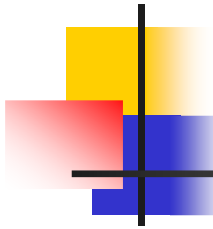
- Dopo la definizione:
  new *DelegateType* (*obj*.*Method*)
- Una variabile delegato memorizza un metodo e il suo ricevitore, ma nessun parametro:
  new Notifier(myObj.SayHello);
- *obj* può essere *this* (e può essere sottinteso):
  new Notifier(SayHello);
- I metodi possono essere statici. In questo caso deve essere specificato il nome della classe in *obj*.
  new Notifier(MyClass.StaticSayHello);
- I metodi non possono essere astratti, ma può essere *virtual*, *override*, or *new*.
- La firma del *Metodo* deve coincidere con quella del *tipo delegato*
  - Stesso numero di parametri
  - Stesso tipo dei parametri (compreso il tipo restituito)
  - Stesso tipo dei parametri (ref, out, value)

Funzionamento Delegati

- Una variabile delegata può contenere più valori!!

```
Notifier greetings;
greetings = new Notifier(SayHello);
greetings = greetings + new Notifier(SayGoodBye);

greetings("John"); // "Hello from John"
        // "Good bye from John"

greetings = greetings - new Notifier(SayHello);

greetings("John"); // "Good bye from John"
```

- se il delegato multicast è una funzione restituisce il valore dell'ultima invocazione
- Se il delegato multicast ha un parametro out, è restituito il parametro dell'ultima chiamata. I parametri per riferimento sono passati a tutti i metodi.



Funzionamento Delegati

```
delegate void Printer(string s);
void Foo(string s) {
        Console.WriteLine(s);
}
```

```
Printer print;
print = new Printer(Foo);
print = this.Foo;   // OK
print = Foo;                    // OK
```

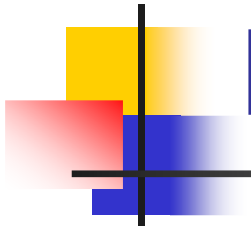
Forma semplificata di dichiarazione delegato:
il tipo del delegato è ricavato dal tipo dell'espressione sinistra

```
delegate double Function(double x);
double Foo(double x) {
        return x * x;
}
```

```
Printer print = Foo;
Function square = Foo;
```

Assegna Foo(string s)
Assegna Foo(double x)

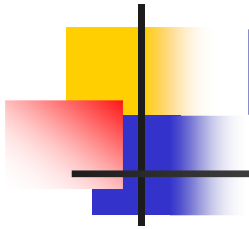
L'overloading è risolto dal tipo di sinistra



Funzionamento Delegati

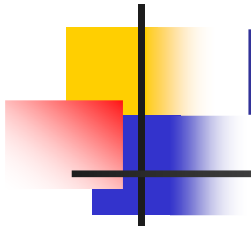
I delegati sono utili :

- Perché viene chiamato un unico metodo delegato per tanti metodi.
- Quando una classe può richiedere più implementazioni della specifica del metodo.
- È preferibile consentire l'utilizzo di un metodo static per implementare la specifica.
- Si desidera uno schema di progettazione di tipo evento.
- Quando il chiamante non ha interesse a conoscere o ottenere l'oggetto su cui è definito il metodo (garantisce l'indipendenza del codice del chiamante rispetto al nome dell'oggetto della classe sfruttata dal chiamante).
- Il fornitore dell'implementazione desidera passare l'implementazione della specifica solo ad alcuni componenti selezionati.
- È preferibile una composizione semplice.



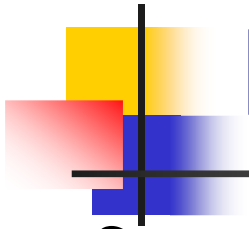
Esempio Delegati

```
class TestOperazioniMatematiche {

// MODO CLASSICO (SENZA DELEGATI)
Math m1=new Math();
int res = m1.somma(10,20);
int res = m1.differenza(10,20);

// MODO CON DELEGATI...
delegate int Action(int x, int y);
Action a;
Math m=new Math();
a=new Action(m.somma); int res = a(10,20);
a=new Action(m.differenza); int res = a(10,20);

// ATTRAVERSO I DELEGATI, LE DUE CHIAMATE SONO DIRETTE VERSO
// METODI DIVERSI MA HANNO LO STESSO ASPETTO
// QUINDI SI OTTIENE RIUTILIZZO ED INDIPENDENZA DEL CODICE

class Math{
public int somma(int s1, int s2) { return (s1+s2);}
public int differenza(int d1, int d2) { return (d1-d2);}
```



Esempio Delegati

```
class TestOperazioniMatematiche {

// DELEGATI CON METODI E PARAMETRI OCCULTATI
// PER PERMETTERE UNA TOTALE INDIPENDENZA DELLA CLASSE TEST DA
    MATH

delegate int Action(
int x, int y);
Action a;
int res;
List l=metodo_per_ricavare_elenco_metodi_di_math_tramite_reflection();
foreach(metodo in l){
    a=new Action(metodo);
    par1=metodo_che_prende_il_valore_attuale_del_primo_parametro();
    par2=metodo_che_prende_il_valore_attuale_del_secondo_parametro();
    res=a(par1,par2);
}
}
```



Esempio Delegati

- Come detto, i delegati non sono una novità, essendo derivati dai puntatori a funzione del C. Un esempio di puntatore a funzione:

```
#include "stdafx.h"
#include "iostream.h"
double (*CalcolaArea)(double Base, double Altezza);
double AreaQuadrato(double Base, double Altezza) { return Base*Altezza; }
double AreaTriangolo(double Base, double Altezza) { return Base*Altezza/2; }
int main(int argc, char* argv[]) {
double b=0;  double h=0;  double A=0;  char figura;
 cout << "Base : ";  cin >> b;
 cout <<"Altezza : ";  cin >> h;
 cout << "(Q)uadrato o (T)riangolo : ";  cin >> figura;
 if (figura=='T') {          CalcolaArea = &AreaTriangolo; }
else {                       CalcolaArea = &AreaQuadrato; }
cout <<"Area = ";
     cout << (*CalcolaArea)(b,h);
     return 0; }
```

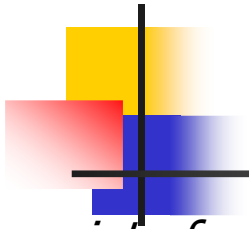
Notare che un delegato è type safe, un puntatore a funzione NO

Controvarianza e covarianza

- Esistono due meccanismi che incrementano la flessibilità dei delegati: **covarianza e controvarianza**. Covarianza: Il tipo restituito da un metodo può essere una derivazione del tipo restituito dal delegato. Esempio:

```
class Mammals        {        …        }
class Dogs : Mammals {        …        }
class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();
    public static Mammals FirstHandler()    {        return null;    }
    public static Dogs SecondHandler()    {        return null;    }
    static void Main()    {
                HandlerMethod handler1 = new HandlerMethod(FirstHandler);
        // Covariance allows this delegate.
        HandlerMethod handler2 = new HandlerMethod(SecondHandler);
    }}
```

- Controvarianza: I parametri del delegato possono essere derivazioni dei parametri della signature, ovvero i metodi che saranno usati nel delegato possono avere come tipo del parametro una qualunque superclasse del tipo dichiarato nel delegato stesso

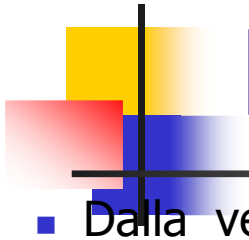


Delegati in Java

```
interface Notifier {                    // corrisponde al delegato
   void greetings(String sender);
}
class HelloSayer implements Notifier {          // valore delegato
   public void greetings (String sender) {
      System.out.println("Hello from" + sender);
   }
}
Notifier n = new HelloSayer(); // inizializzazione del delegato

n.greetings("John");
```

Le interfacce hanno la funzione del delegato

- Una classe che implementa l'interfaccia.
- I metodi statici non possono essere delegati.
- Il multicast richiede un meccanismo ad-hoc per registrare i metodi e chiamarli.



Metodi anonimi

- Dalla versione 2.0 di C# sono stati introdotti i metodi anonimi, che consente essenzialmente di passare un blocco di codice come parametro del delegato evitando di creare il metodo (nelle versioni precedenti era possibile dichiarare un delegato solo tramite metodi denominati). Utilizzando i metodi anonimi si riduce l'overhead della codifica nella creazione di istanze dei delegati, eliminando la necessità di creare un metodo separato. Esempio:

```
delegate void Del(int x);
Del d = delegate(int k) { /* CODICE SENZA NOME */ };
```

Altro esempio:

```
delegate void Printer(string s);
class TestClass
{    static void Main()    {
        // Instatiate the delegate type using an anonymous method:
        Printer p = delegate(string j)    {    System.Console.WriteLine(j);    };
        p("The delegate using the anonymous method is called.");
        // The delegate instantiation using a named method "DoWork":
        p = new Printer(TestClass.DoWork);
        p("The delegate using the named method is called.");    }
    // The method associated with the named delegate:
    static void DoWork(string k)    {        System.Console.WriteLine(k);    }}
```

Metodi anonimi

- Altro esempio di utilizzo dei delegati anonimi:
- È necessario dichiarare un metodo (SumUp, Print, ...)
- SumUp e Print non possono accedere alle variabili locali Foo => quindi sum deve essere dichiarato globale nella classe C

```
delegate void Visitor(Node p);
class List {
    Node[] data = ...;
    ...
    public void ForAll(Visitor visit) {
        for (int i = 0; i < data.Length; i++)
            visit(data[i]);    }}
```

```
class C {
    int sum = 0;
    void SumUp(Node p) { sum += p.value; }
    void Print(Node p) { Console.WriteLine(p.value); }
    void Foo() {
        List list = new List();
        list.ForAll(SumUp);
        list.ForAll(Print);  } }
```

Metodi anonimi

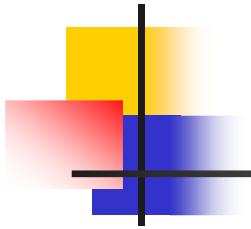
```
delegate void Visitor(Node p);

class C {

      void Foo(ref int sum) {
            List list = new List();
            int sum = 0;
            list.ForAll(delegate (Node p) { Console.WriteLine(p.value); });
            list.ForAll(delegate (Node p) { sum += p.value; });        }}
```

```
class List {
      ...
      public void ForAll(Visitor visit) {
            ...
      }
}
```

- Il codice è specificato sul posto
- Non richiede la dichiarazione di un metodo "named"
- Il metodo anonimo può accedere alla variabile locale sum di Foo's
- L'istruzione return termina il metodo anonimo (non il metodo chiamante)

Restrizioni:

- I metodi anonimi non possono avere parametri formali del tipo params T[]
- I metodi anonimi non possono essere assegnati ad object
- I metodi anonimi non possono accedere ai parametri ref o out del chiamante



Metodi anonimi

- I metodi anonimi sono metodi senza nome
- Utilizzato quando un metodo è necessario solo come delegato
  - Serve ad evitare la definizione di un metodo separato
  - Maggiore chiarezza
- Il metodo anonimo non ha firma
  - La firma è ricavata da quella del delegante (delegate inference)
  - Il valore restituito deve coincidere con quella del metodo delegato
  - Vantaggio: è associabile a qualsiasi delegato
  - Svantaggio: non è possibile accedere ai parametri del delegato, quindi non è utilizzabile quando sono presenti parametri ref o out

Il compilatore:

- Verifica che il delegato è senza parametri out;
- Ricava la firma del delegato dal delegante
- Verifica che il tipo restituito sia compatibile
- Crea un nuovo delegato inizializzato con il metodo anonimo

Metodi anonimi

- Semplificazione sintassi metodi anonimi:

```
delegate void EventHandler (object sender, EvtArgs arg);
```

```
Button button = new Button();
Button.Click += delegate (object sender, EvtArgs arg) { Console.WriteLine("click"); };
```

Può essere semplificato con:

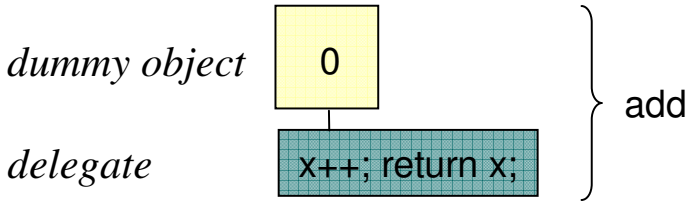
Button.Click += delegate {Console.WriteLine("clicked"); };

- I parametri formali possono essere omessi se non sono usati nel metodo
- I parametri formali possono essere omessi solo se il tipo delegate non ha parametri *out*

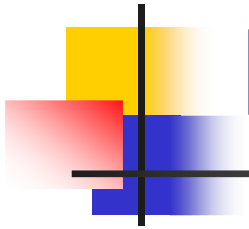
Metodi anonimi

- Se il metodo anonimo accede a variabili del metodo chiamante, queste sono gestite tramite un oggetto dummy (capturing)

```
delegate int Adder();

class Test {
    static Adder CreateAdder() {
        int x = 0;
        return delegate { x++; return x; };
    }
    static void Main() {
        Adder add = CreateAdder();
        Console.WriteLine(add());
        Console.WriteLine(add());
        Console.WriteLine(add());
    }
}
```



- Il tempo di vita dell'oggetto dummy è uguale a quello dell'oggetto delegate



Esempio Delegati

- http://msdn.microsoft.com/it-it/library/aa288459(VS.71).aspx
- L'esempio mette in evidenza che si possono separare l'utilizzo del delegato dalla sua inizializzazione, anche in file diversi, cosi che la chiamata possa ignorare le sue azioni effettive, operando un totale disaccoppiamento fra i due, che potrebbero anche essere sviluppati da persone diverse
- **Esercitazione**: realizzare un sistema per la gestione dei movimenti eseguiti da vari clienti di una banca. In particolare, i movimenti sono memorizzati su file, uno per riga, con indicazione del tipo di operazione (v per versamento, p prelievo), importo e tipologia cliente (privato o azienda). Il sistema, utilizzando i delegati, deve permettere di calcolare il totale delle operazioni divise per tipo, ed il massimo valore di prelievo per privati (o aziende).