

OOP EREDITARIETÀ



Andrea Zoccheddu

CORSO INFORMATICA ITI ANGIOY SASSARI



Sintesi

Questa dispensa introduce l'argomento dell'ereditarietà, della derivazione di classi a partire da altre classi. Si introduce la terminologia di classi base e classi derivate.

EREDITARIETÀ

CLASSI DERIVATE

CONCETTO

■ Fonte: <http://msdn.microsoft.com/it-it/library/ms173149.aspx>

L'ereditarietà, insieme all'incapsulamento e al polimorfismo, rappresenta una delle tre principali caratteristiche (o pilastri) della programmazione orientata a oggetti.

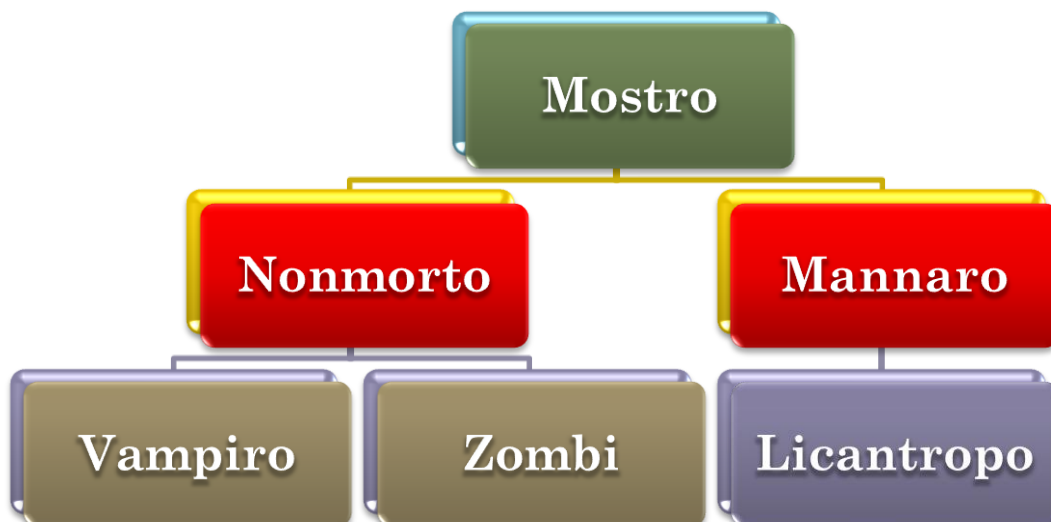
L'ereditarietà permette di creare nuove classi che riutilizzano, estendono e modificano la definizione ed il comportamento definito in altre classi. La classe da cui costruire altre classi è denominata classe base, mentre la classe che eredita tali caratteristiche è denominata classe derivata.

La metodologia dell'ereditarietà assume quindi che si può costruire una classe a partire da un'altra classe. Secondo questa metodologia una classe derivata prende automaticamente le caratteristiche della classe base. Quindi la classe riceve attributi, metodi e altre caratteristiche dalla classe base.

Quando si definisce una classe derivandola da un'altra classe, la classe derivata acquista implicitamente tutti i membri della classe base, con l'eccezione dei costruttori e dei distruttori. Di conseguenza, la classe derivata può riutilizzare il codice definito nella classe base senza doverlo implementare nuovamente. Nella classe derivata è possibile aggiungere altri membri. In questo modo, la classe derivata estende la funzionalità della classe base.

In Visual C# una classe derivata può avere al massimo una sola classe di base diretta. In Visual C# però è possibile definire classi derivate da altre classi derivate creando di fatto una gerarchia; in questo senso si può dire che l'ereditarietà è transitiva. Se ClassC è derivata da ClassB e ClassB è derivata da ClassA, allora ClassC eredita sia i membri dichiarati in ClassB sia quelli dichiarati in ClassA.

Per esempio potremmo pensare di voler costruire un gioco in cui compaiono dei Mostri; alcuni di questi mostri sono Nonmorti altri sono Mannari; tra i Nonmorti vogliamo definire due categorie di base Zombi e Vampiri; i Mannari invece hanno una sottocategoria che sono i Licantropi. In questa biologia fantastica, la gerarchia assume un aspetto simile alla seguente illustrazione:



Un Antenato è un oggetto da cui si derivano altri oggetti; per esempio Mostro è antenato di Vampiro o di Licantropo. Un Discendente è un oggetto derivato da altri oggetti; per esempio Zombi è un discendente di Mostro, ma non di Mannaro.

DEFINIZIONE

La definizione di una classe derivata è simile a quella di una qualsiasi classe, con l'esplicitazione di due punti dopo il nome seguiti dal nome della classe base, come nella seguente sintassi:

```
class Genitore {           //classe base
    //attributi
    //metodi
}

class Figlio : Genitore {   //classe derivata
    //altri attributi
    //altri metodi
}
```


Ovviamente la definizione di una classe consente di dichiarare variabili sia della classe base che della classe derivata:

```
Genitore g = new Genitore();
Figlio f = new Figlio();
```

UTILIZZO

Poiché la classe derivata eredita le caratteristiche della classe base, essa può utilizzare attributi e metodi ereditati. Per esempio si consideri la seguente dichiarazione:

<pre>class Mostro { public string nome ; public int punti ; public Mostro (string n) { nome = n ; punti = 100 ; } }</pre>	<pre>class Nonmorto : Mostro { public bool visibile ; public Nonmorto (string n) { nome = n ; visibile = true ; punti = 500 ; } }</pre>
---	---

A red arrow points from the `Nonmorto` class definition to the `Mostro` class definition, indicating that `Nonmorto` inherits from `Mostro`.

In questo esempio il costruttore della classe **Nonmorto** accede all'attributo **nome** che ha ereditato dalla classe **Mostro**.

Se si dichiarano istanze delle classi, anche esse possono utilizzare questi attributi e metodi:

<pre>Mostro m; m = new Mostro ("Cerbero"); m.nome = "Caronte"; int p = m.punti;</pre>	<pre>Nonmorto n; n = new Nonmorto ("Dracula"); n.nome = "Vlad"; int q = n.punti;</pre>
---	--

SCOPO

Uno dei motivi che ha rafforzato l'adozione della Programmazione a oggetti (OOP) è la riusabilità del codice. Infatti, poiché la classe ereditata ottiene dalla classe base caratteristiche già definite, risulta comodo definire nella nuova classe solo quegli elementi innovativi che portano un'estensione delle capacità dell'oggetto, ma sfruttando quanto già definito precedentemente nelle classi antenate.

GERARCHIE ED ALBERI DI CLASSI

La creazione di una gerarchia si fonda sull'ipotesi che un oggetto discendente sia ANCHE un oggetto antenato. Nella gerarchia illustrata all'inizio della dispensa si può osservare che:

- Un Mannaro è un Mostro; ma non tutti i Mostri sono Mannari;
- Un Licantropo è un Mostro; ma non tutti i Mostri sono Licantropi;
- Un Vampiro è un NonMorto; ma non tutti i Nonmorti sono Vampiri;
- Un Vampiro è un Mostro; ma non tutti i Mostri sono Vampiri;



Questa assunzione consente di utilizzare le variabili di tipo oggetto consapevoli che potrebbero essere delle specializzazioni. Se per esempio si ha una variabile di tipo Mostro e la si usa, si deve essere consapevoli che, talvolta, l'istanza di quella variabile potrebbe essere un Zombi, un Mannaro o un Vampiro. Si tornerà su questo aspetto delle gerarchie più volte nel seguito della dispensa.

EREDITARIETÀ DEGLI ATTRIBUTI

POSSESSO DEGLI ATTRIBUTI

Una classe derivata possiede tutte le caratteristiche delle classi antenato. Il fatto di averle non sempre implica che sia possibile accedervi. In alcuni casi le caratteristiche dell'antenato sono nascoste, invisibili e protette dalla classe antenato, che ne impedisce l'accesso.

ATTRIBUTI OMONIMI

Una classe derivata può dichiarare attributi omonimi a quelli della classe base. In questo caso il nome dell'attributo sarà coperto in visibilità. È anche possibile definire l'attributo con un tipo diverso. Per esempio:

```

class Mostro
{
    public string nome ;
    public double punti ;
    public string Nome ()
    {
        return nome; //nome del Mostro
    }
}
  
```

```

class Mannaro : Mostro
{
    public string nome ;
    public int punti ;
    public string Nome ()
    {
        return nome; //nome del Mannaro
    }
}
  
```

Nell'esempio, il metodo del Mostro restituisce il valore dell'attributo definito per le istanze della classe Mostro; ma nella classe Mannaro, derivata da Mostro, l'attributo nome è ridefinito, per cui il metodo Nome non «vede» il nome ereditato dal mostro (anche se le istanze di Mannaro lo possiedono) e invece restituisce il valore dell'attributo nome del mannaro. Per esempio:

```

Mostro m1 = new Mostro();
m1.nome = "Gerione";
Mannaro m2 = new Mannaro();
m2.nome = "Anubi"; //accesso all'attributo del Mannaro
MessageBox.Show( m2.Nome() ); //accesso al metodo del Mannaro (scrive Anubi)
  
```

ACCESSO A ATTRIBUTI DI ANTENATI (BASE)

Ci si potrebbe chiedere come sia possibile accedere a un attributo ereditato ma nascosto in visibilità da una ridenominazione, considerato che le istanze della classe derivata hanno a disposizione entrambi gli attributi. In effetti Visual C# offre una soluzione al programmatore.

La parola chiave `base` (minuscola) consente infatti di accedere agli elementi della classe genitore (quella da cui ha discendenza diretta). Consideriamo l'esempio seguente:

```
class Mostro
{
    public double punti ;
    public double Valore ()
    { // punti del Mostro
        return punti;
    }
}
```

Esempio d'uso:

```
Mannaro m2 = new Mannaro ();
m2.punti = 10;
double x = m2.Valore_Proprio ();
double y = m2.Valore_Padre ();
double z = m2.Valore ();
```

```
class Mannaro : Mostro
{
    public int punti ;
    public double Valore_Proprio()
    { // punti Mannaro
        return punti;
    }

    public double Valore_Padre()
    { // punti Mostro
        return base.punti;
    }

    public double Valore()
    { // somma punti
        return punti + base.Valore();
    }
}
```

Nell'esempio sopra proposto, il metodo `Valore_proprio()` rende il valore dell'attributo della classe Mannaro (come già detto in precedenza). Il metodo `Valore_Padre()` invece rende il valore dell'attributo ereditato dalla classe Mostro (nascosto in visibilità). Infine il metodo `Valore Totale()` rende il valore calcolato dal metodo ereditato dalla classe Mostro e ci somma il valore dell'attributo proprio.

INACCESSIBILITÀ DEGLI ATTRIBUTI PRIVATI

Rispetto all'esempio appena discusso, occorre tuttavia soffermarsi su una situazione di protezione della classe, che riguarda le caratteristiche private. Per esempio:

```
class Mostro
{
    private double punti ;
    private double Valore ()
    { // punti del Mostro
        return punti;
    }
}
```


Esempio d'uso:

```
Mannaro m2 = new Mannaro ();
m2.punti = 10;
double x = m2.Valore_Proprio ();
double y = m2.Valore_Padre ();
double z = m2.Valore ();
```

```
class Mannaro : Mostro
{
    public int punti ;
    public double Valore_Proprio()
    { // punti del Mannaro
        return punti;
    }

    public double Valore_Padre()
    { // punti Mostro
        return base.punti;
    }

    public double Valore()
    { // somma punti
        return punti + base.Valore();
    }
}
```



Nell'esempio proposto sopra il compilatore solleva due errori (provare): l'accesso all'attributo `base.punti` è vietato (risulta non esistente) poiché privato; analogamente anche l'invocazione al metodo `base.Valore()` è vietata (risulta non esistente) poiché anch'esso privato. Questa indisponibilità delle caratteristiche (nascoste dalla privatezza dell'incapsulamento) impedisce un uso pieno e completo delle caratteristiche di cui comunque l'oggetto ha possesso.

ACCESSO E PROTEZIONE CON PROTECTED

Per ampliare la disponibilità delle caratteristiche incapsulate, pur mantenendole inaccessibili dall'esterno, il linguaggio mette a disposizione la parola chiave `protected`. Una caratteristica `protected` è considerata privata dall'esterno della classe, ma pubblica per i suoi discendenti.

```
class Mostro
{
    protected double punti ;
    protected double Valore ()
    { // punti del Mostro
        return punti;
    }
}
```

Esempio d'uso:

```
Mannaro m2 = new Mannaro ();
m2.punti = 10;
double x = m2.Valore_Proprio ();
double y = m2.Valore_Padre ();
double z = m2.Valore ();
```

```
class Mannaro : Mostro
{
    public int punti ;
    public double Valore_Proprio()
    { // punti del Mannaro
        return punti;
    }

    public double Valore_Padre()
    { // punti del Mostro
        return base.punti;
    }

    public double Valore()
    { // somma dei punti
        return punti + base.Valore();
    }
}
```

Nell'esempio proposto sopra le caratteristiche dell'oggetto `Mostro` sono state dichiarate protette. Questo tipo di dichiarazione impedisce che dall'esterno si possa accedere alle caratteristiche, in modo del tutto identico alla dichiarazione privata.

Ma quando ad accedervi è una classe derivata, nel nostro esempio `Mannaro`, è del tutto lecito (e non solleva errori o eccezioni) e consente di definire nuovi metodi sulla base di queste caratteristiche incapsulate.

EREDITARIETÀ DEI METODI

POSSESSO DEI METODI

Analogamente agli attributi, anche per i metodi vige la stessa regola degli attributi. Una classe derivata eredita tutti i metodi (ma non i costruttori) della classe base (e per conseguenza anche quelli degli antenati nella gerarchia).

Se i metodi della classe base sono pubblici, la classe derivata ne ha pieno accesso.

Se i metodi della classe base sono privati, la classe derivata non ne ha accesso, pur avendone il possesso; cioè li possiede ma non ne può disporre.

Se i metodi della classe base sono protetti, la classe derivata ne ha pieno accesso.

```
class Mostro
{
    public double punti ;
    public double Valore ()
    { // punti del Mostro
        return punti;
    }
}
```

```
class Mannaro : Mostro
{
    public double punti ;
    public double Prova ()
    { // punti del Mannaro
        base.punti = 100;
        return Valore();
    }
}
```

```
Mannaro lupo = new Mannaro (); //creo l'istanza di Mannaro
lupo.punti = 123;           //scrivo 123 nell'istanza
double x = lupo.Prova (); //scrivo 100 nell'attributo punti ereditato
MessageBox.Show("" + x); //visualizza 100
double y = lupo.Valore(); //chiama il metodo ereditato
MessageBox.Show("" + y); //visualizza 100
```

OVERLOAD DEI METODI

Si è già discusso dei metodi overload anche prima di discutere dell'ereditarietà, quando metodi omonimi sono dichiarati nella stessa classe. Una classe derivata può ridefinire metodi omonimi alla classe base, persino con firma identica. La ridefinizione nasconde i metodi della classe base, anche se la classe ne ha ancora possesso. La ridefinizione di un metodo omonimo tra classi ereditate è detto **overload** (sovraccarico) del metodo.

```
class Mostro
{
    public double punti ;
    public double Valore () ←
    { // punti del Mostro
        return punti;
    }
}
```

```
class Mannaro : Mostro
{
    public double Valore () ←
    { // punti del Mannaro
        return 2*punti;
    }
}
```

OK

Nella classe discendente dell'esempio, il metodo Valore è sovraccaricato. Quando un'istanza di tipo Mannaro invoca il metodo Valore questo si riferisce a quello definito nella classe Mannaro. Per esempio:

```
Mannaro anubi = new Mannaro ();
anubi.punti = 120;
double x = anubi.Valore(); //x vale 240;
```

Il metodo Valore è legato all'istanza al momento della sua istanziazione. Nell'esempio appena proposto, il metodo Valore è legato all'istanza linkata da anubi al momento in cui si esegue l'istruzione new. Questa osservazione sarà presto molto importante, per cui vale la pena di rifletterci attentamente.

INVOCAZIONE DI METODI OVERLOADED DI ANTENATI (BASE)

Se il metodo è sovraccaricato è sempre possibile invocarlo dall'interno di un metodo di una classe derivata, sfruttando ancora una volta la parola chiave **base**.

```
class Mostro {
    public double punti ;
    public double Valore () ←
    { // punti del Mostro
        return punti;
    }
}
```

```
class Mannaro : Mostro {
    public double Valore ()
    { // punti del Mannaro
        return 2*base.Valore(); ←
    }
}
```

Nella classe discendente dell'esempio, il metodo Valore è sovraccaricato ma nel suo corpo è presente un'invocazione al metodo della sua classe base. Quando si invoca il metodo Valore di Mannaro, questo richiama il metodo Valore come definito nella classe Mostro.

```
Mannaro anubi = new Mannaro ();
anubi.punti = 321;
double x = anubi.Valore(); //x vale 642;
```

INACCESSIBILITÀ DEI METODI PRIVATI

Analogamente a quanto accade per gli attributi, se un metodo è privato la classe discendente non ne ha accesso e non può invocarlo. Per esempio:

<pre>class Mostro { public double punti ; private double Valore () { // punti del Mostro return punti; } }</pre>	<pre>class Mannaro : Mostro { public double Valore () { // punti del Mannaro return 2*base.Valore(); } }</pre>
--	--

Il tentativo di invocazione del metodo `base.Valore()` solleva un errore di compilazione, poiché la classe discendente non ha visibilità del metodo privato della classe base.

ACCESSO AI METODI DERIVATI: PROTECTED

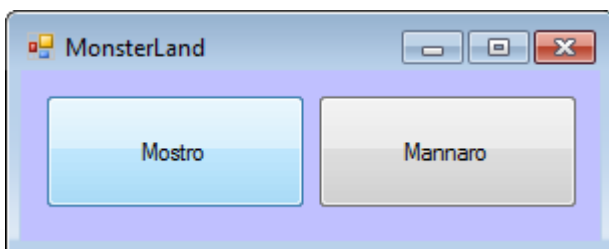
Ancora una volta è possibile usare il dichiaratore di accesso `protected` per incapsulare i metodi da proteggere, ma consentire alle classi base di accedervi.

<pre>class Mostro { public double punti ; protected double Valore () { return punti; } }</pre>	<pre>class Mannaro : Mostro { public double Valore () { return 2*base.Valore(); } }</pre>
--	---

PROGETTO GUIDATO

- Si crei un nuovo progetto con Visual Studio e si prepari un Form1 come nella figura; poi si definiscano le classi `Mostro` e `Mannaro`, la seconda ereditata dalla prima;

<pre>class Mostro { protected double punti ; protected string verso ; public Mostro () { punti = 25; verso = "grugnito"; } public double Valore () { return punti; } public string Verso () { return verso; } }</pre>	<pre>class Mannaro : Mostro { protected bool mutato; public Mannaro () { punti = 33; verso = "ululato"; mutato = true; } public double Valore () { if (mutato) return 2*base.Valore(); else return base.Valore(); } public string Verso () { if (mutato) return verso; else return base.verso; } public void Muta () { mutato = !mutato; } }</pre>
---	--



- Prima di eseguire il programma provare ad anticipare l'effetto che avranno i metodi e se ci sono errori
- Associare al button1 il codice seguente:

```
private void button1_Click(object sender, EventArgs e)
{
    Mostro demone = new Mostro();
    MessageBox.Show(demone.Valore() + " " + demone.Verso());
}
```

- Associare al button2 il codice seguente:

```
private void button2_Click(object sender, EventArgs e)
{
    Mannaro umano = new Mannaro ();
    MessageBox.Show(umano.Valore() + " " + umano.Verso());
    umano.Muta ();
    MessageBox.Show(umano.Valore() + " " + umano.Verso());
}
```

- Prima di eseguire l'applicazione, formula un'ipotesi di cosa accadrà premendo i pulsanti!

INVOCAZIONE DEL COSTRUTTORE BASE

Si è già osservato che una classe derivata può accedere e sfruttare attributi e metodi della classe base da cui discende. In modo analogo può essere importante richiamare anche un costruttore della classe base, soprattutto per inizializzare l'istanza correttamente (come si suppone che faccia la classe base) e con accesso anche a attributi privati, inaccessibili dalla classe derivata.

<pre>class Mostro { private int punti ; public Mostro () : this (0) { } public Mostro (int q) { this.punti = q; } }</pre>		<pre>class Spettro : Mostro { private bool invisibile; public Spettro () : base (25) { invisibile = true; } public Spettro (int p) : base (p) { invisibile = true; } }</pre>
---	--	--

Nell'esempio qui sopra, il Mostro ha un attributo privato punti. La classe derivata Spettro non può accedervi, ma possiede l'attributo.

La classe Spettro ha due costruttori rispettivamente senza e con parametri. Il costruttore senza parametri, prima di eseguire il corpo, richiama il costruttore con un parametro passando l'argomento 25 che sarà assegnato all'attributo punti. Per esempio:

```
Spettro unoSpettro = new Spettro ();           //unoSpettro vale 25 punti
```

Il costruttore di Spettro con un parametro chiama lo stesso costruttore ma questa volta l'argomento è lo stesso valore che sarà passato al costruttore Spettro. Per esempio:

```
Spettro altroSpettro = new Spettro (17);       //altroSpettro vale 17 punti
```

COMPATIBILITÀ TRA TIPI DERIVATI

ASSEGNAZIONE TRA ISTANZE

All'inizio della dispensa si è accennato al fatto che le classi derivate formano una gerarchia. Questo implica che una istanza di una classe discendente è anche una istanza di una classe antenata.

Riprendendo l'esempio introduttivo si ha che:

- Un Mannaro è un Mostro; ma non tutti i Mostri sono Mannari;
- Un Licantropo è un Mostro; ma non tutti i Mostri sono Licantropi;
- Un Vampiro è un NonMorto; ma non tutti i Nonmorti sono Vampiri;
- Un Vampiro è un Mostro; ma non tutti i Mostri sono Vampiri;



Questa considerazione apre all'idea che sia possibile compiere assegnazioni tra variabili di classi diverse, ma della stessa gerarchia. In generale, poiché una classe discendente è anche una classe antenata, è possibile assegnare una istanza di tipo discendente a una variabile di tipo antenata.

```

Mostro uno = new Mostro();
Mannaro alfa = new Mannaro();
Mostro due = uno;      // OK assegnazione tra istanze dello stesso tipo
Mannaro beta = alfa;   // OK
//
uno = alfa ;           // OK perché un Mannaro è anche un Mostro
beta = due ;           // NO perché un Mostro NON è detto che sia un Mannaro
//
uno = new Mannaro (); // OK perché un Mannaro è anche un Mostro
beta = new Mostro() ; // NO perché un Mostro NON è detto che sia un Mannaro
//
  
```

COMMENTI E RIFLESSIONI

Sebbene possa essere contro intuitivo non è possibile assegnare una istanza di una classe antenata a una variabile di una classe discendente. Per esempio, nel precedente esempio:

```

beta = due ;           // NO perché un Mostro NON è detto che sia un Mannaro
beta = new Mostro() ; // NO perché un Mostro NON è detto che sia un Mannaro
  
```

si solleva un errore in entrambi i casi poiché la variabile due è di tipo Mostro (quindi potrebbe essere uno Zombi) e il rischio di assegnarlo a una variabile di tipo Mannaro (come beta) non è accettabile dal compilatore.

COSTRUTTORE DI UN ANTENATO

Una variabile discendente non può essere istanziata con un costruttore di un antenato:

```

beta = new Mostro() ; // NO perché un Mostro NON è detto che sia un Mannaro
  
```

per lo stesso motivo.

COLLEZIONI DI OGGETTI E CLASSI DERIVATE

Si è quindi appreso che una locazione di tipo antenato (e quindi più generale) può accogliere una qualsiasi istanza di tipo discendente. Nel modo più semplice si può eseguire una assegnazione come la seguente:

```

Mannaro alfa = new Mannaro();
Mostro uno = alfa ;    // OK
  
```

un modo più ampio di utilizzare questa possibilità è considerare una collezione di locazioni di tipo oggetto. Per esempio:

```
List<Mostro> torma = new List<Mostro>(); // gruppo di mostri
Mannaro pippo = new Mannaro (); //singolo mannaro
torma.Add(pippo) ; //aggiungo un mannaro nel gruppo di mostri
Vampiro vlady = new Vampiro (); //singolo vampiro
torma.Add(vlady) ; //aggiungo un vampiro nel gruppo di mostri
```

in questo esempio si osserva come una collezione (ma anche un vettore andrebbe bene) di mostri è in grado di accogliere vari tipi di sotto-mostri, come mannari e vampiri. In questo modo l'elenco può contenere molti elementi diversi tra loro, accomunati dal fatto di discendere dalla classe su cui è definita la collezione.

CONTROLLO DI TIPO: IS

Se in un insieme di istanze è possibile che siano presenti istanze di diverso tipo, è utile poter verificare quale sia il tipo effettivo dell'istanza a cui fa riferimento una data locazione. Per esempio se avessimo un caso simile al seguente:

```
Mannaro alfa = new Mannaro();
Nonmorto zeta = new Nonmorto();
Mostro uno ;
if ( casuale.Next(2) == 0 )
    uno = alfa ;    // OK
else
    uno = zeta;     // OK
```

si nota che c'è una probabilità del 50% che la variabile uno possa fare riferimento a una istanza di tipo Mannaro oppure di tipo Nonmorto, entrambi i casi ammissibili. Il tipo dichiarato della variabile uno è sempre Mostro, ma il tipo dell'istanza a cui si riferisce può essere diversa. Ma come verificare il tipo effettivo puntato dalla locazione?

Visual C# ha un operatore che rende un valore logico (bool) per accertare se una istanza sia di un certo tipo: l'operatore is (minuscolo).

La sintassi è:

```
locazione is Tipo
```

ed il valore reso è un booleano. Esempio:

```
Mostro uno = . . . . ; //non sappiamo cosa verrà assegnato
if ( uno is Mannaro )
    MessageBox.Show("il mostro è un mannaro");
else
    MessageBox.Show("il mostro NON è un mannaro");
```

CAST DI TIPO: AS

Consideriamo adesso il seguente esempio:

```
class Mostro
{
    public void Verso ()
    {
        return "grugnito";
    }
}
```

```
class Mannaro : Mostro
{
    public void Verso ()
    {
        return "ululato";
    }
}
```

```
class Zombi : Mostro
{
    public void Verso ()
    {
        return "rantolo";
    }
}
```

Sulla base delle precedenti dichiarazioni ipotizziamo che casualmente (al 50%) una variabile mostro sarà istanziata come Mannaro oppure come Zombi:

```
Mostro guardiano ;
if ( casuale.Next(2) == 0 )
    guardiano = new Mannaro ();           // costruttore di Mannaro
else
    guardiano = new Zombi ();             // costruttore di Zombi
```

Se a questo punto invochiamo il metodo Verso, come segue, cosa succede?

```
MessageBox.Show ( "verso: " + guardiano.Verso() );
```

possiamo scoprire che il verso prodotto sarà sempre "grugnito" a prescindere dalla istanza linkata. Il motivo è che la variabile guardiano è in ogni caso un Mostro e il compilatore lega il metodo da eseguire a tempo di compilazione (prima di eseguire il programma quindi).

Per fortuna, oltre a controllare se l'istanza puntata dalla locazione appartiene a una certa classe, è anche possibile forzare l'uso dell'istanza in modo da comportarsi secondo i metodi dell'istanza linkata e non della classe dichiarata. Questa conversione forzata (e rischiosa) si ottiene con l'operatore as, la cui sintassi è la seguente:

```
locazione as Tipo
```

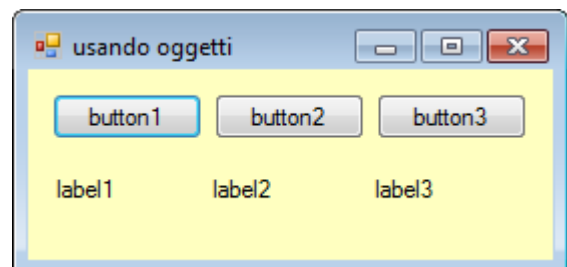
ed il valore ottenuto è del tipo esplicitato. Per esempio:

```
if ( guardiano is Mannaro )
    MessageBox.Show ( "verso: " + ( guardiano as Mannaro ).Verso() );
```

il metodo Verso adesso è compilato in modo da legarsi al metodo del Mannaro, come imposto dalla conversione forzata elaborata dall'operatore as.

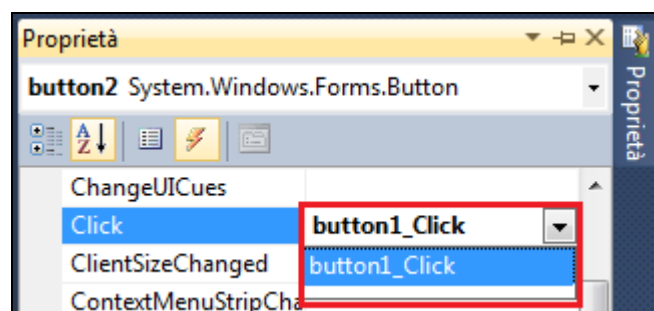
PROGETTO GUIDATO

- ▶ Avviare Visual Studio
- ▶ Progettare un Form1 come quello della figura illustrata qui di lato:
- ▶ Associare al **button1** il seguente gestore:



```
private void button1_Click(object sender, EventArgs e)
{
    if (sender is Button)
        (sender as Button).Text = "hello";
    if (sender is Label)
        (sender as Label).BackColor = Color.Red;
}
```

- ▶ Selezionare la paletta degli eventi delle proprietà
- ▶ Modificare l'evento Click dei cinque controlli rimanenti (i due pulsanti e le tre etichette).
- ▶ Nella figura qui a lato è illustrata la associazione per il pulsante button2
- ▶ Alla fine c'è un solo gestore è associato all'evento dei sei controlli visuali



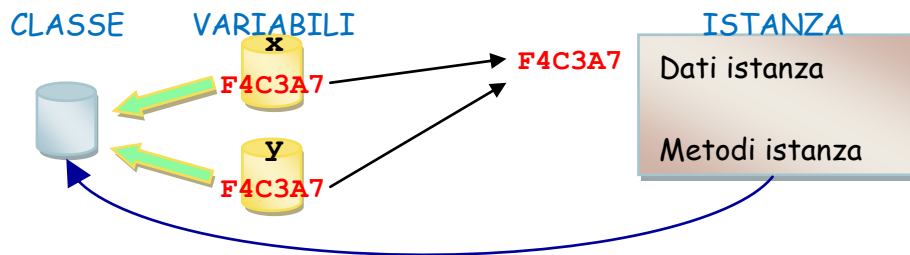
- Provare l'applicazione e premere con un clic del mouse in un qualsiasi ordine i controlli a video
- L'applicazione dovrebbe comportarsi come nella figura illustrata qui a lato
- Termina l'applicazione
- Si rifletta con attenzione sul significato del parametro sender e su come avviene il passaggio di oggetti (es. Button e Label) ad un parametro di tipo Object.



PARAMETRI DI TIPO OGGETTO

UN TIPO RIFERIMENTO

Si è visto come sia possibile assegnare istanze di classi derivate ad una variabile di un tipo classe antenata. Questa possibilità viene sfruttata sia nelle assegnazioni tra variabili, sia per costruire collezioni di oggetti, sia per il passaggio di parametri. Gli oggetti sono tipi riferimento e quindi le assegnazioni tra istanze corrispondono in realtà a assegnazioni di indirizzi e non si strutture.



Nella figura appena illustrata si è rappresentato le variabili come dei contenitori (cilindri) che contengono un indirizzo; l'indirizzo è un riferimento alla istanza, cioè al conglomerato di dati che rappresenta l'oggetto in memoria. Quando si assegna una variabile (x) ad un'altra (y) la copia è solo il duplicato dell'indirizzo, senza duplicare alcuna istanza. Quindi le due variabili hanno ciascuna una copia del dato indirizzo ma entrambe fanno riferimento alla unica istanza.

PASSAGGIO PER VALORE

Quando si ha un passaggio per valore di un oggetto, la situazione è identica a quanto avviene per una assegnazione tra variabili. La variabile destinazione riceve una copia dell'indirizzo, senza che sia duplicata l'istanza; le due locazioni quindi puntano alla stessa istanza. Tuttavia le locazioni sono distinte e pertanto una modifica dell'indirizzo contenuto in un parametro NON si ripercuoterà sull'argomento passato.

```
public Mostro Clona (Mostro p)
{
    p = new Mostro();
    return p;
}
```

```
//invocazione
Mannaro a;
Mostro b;
a = new Mannaro ();
b = Clona (a) ;
```

Nell'esempio proposto qui sopra, il parametro non sarà modificato, ma il metodo renderà una nuova istanza, distinta dall'argomento.

PASSAGGIO PER RIFERIMENTO

Quando si ha un passaggio per riferimento di un oggetto, le due locazioni (argomento e parametro) coincidono. Qualsiasi avvenimento accada sul parametro in realtà avviene sull'argomento. Per questo però il tipo dell'argomento deve essere identico a quello del parametro, e non solo compatibile.

In questo caso una nuova istanziazione con un costruttore modificherà anche il contenuto dell'argomento.

<pre>public Mostro Clona (ref Mostro p) { p = new Mostro(); return p; }</pre>	<pre>//invocazione Mostro a , b; a = new Mostro (); b = Clona (ref Mostro a)</pre>
---	--

Nell'esempio proposto qui sopra, il parametro sarà modificato, e il metodo renderà la stessa nuova istanza puntata sia dal parametro che dall'argomento.

PASSAGGIO PER RISULTATO

Quando si ha un passaggio per risultato di un oggetto, le due locazioni (argomento e parametro) coincidono. L'effetto è identico al passaggio per riferimento, con la variante che l'argomento passato come parametro non deve essere istanziato né ci si aspetta che lo sia. In questo caso è d'obbligo una nuova istanziazione con un costruttore che modificherà anche il contenuto dell'argomento.

<pre>public Mostro Clona (out Mostro p) { p = new Mostro(); return p; }</pre>	<pre>//invocazione Mostro a , b; b = Clona (out Mostro a)</pre>
---	---

Nell'esempio proposto qui sopra, il parametro sarà modificato, e il metodo renderà la stessa nuova istanza puntata sia dal parametro che dall'argomento.

PROGETTO GUIDATO

- ▶ Si avvia un nuovo progetto con Visual Studio
- ▶ Si predisponga il Form1 come nella figura
- ▶ Si dichiarino le seguenti classi:



```
public class Mostro
{
    public string nome;
    public Mostro()
        : this ("Mostro")
    {
    }
    public Mostro(string nome)
    {
        this.nome = nome;
    }
    public void Rivela()
    {
        MessageBox.Show(nome);
    }
}
```

```
public class Mannaro : Mostro
{
    new public string nome;
    public Mannaro()
        : base ()
    {
        this.nome = "Mannaro";
    }
    public Mannaro(string nome)
        : base (nome)
    {
        this.nome = "Mannaro" + nome;
    }
    new public void Rivela()
    {
        string s = nome + (char) (32);
        MessageBox.Show(s + base.nome);
    }
}
```


- Prima di procedere, si analizzino con cura le diverse invocazioni reciproche tra costruttori della stessa classe e della classe base;
- Fuori dalle classi Mostro e Mannaro, si definisca un metodo del Form1 come segue:

```
public Mostro Clona_V (Mostro p) //rende un Mostro nuovo?
{
    p = new Mostro("ValueX");
    p.nome = "Cambia";
    return p;
}
public Mostro Clona_R (ref Mostro p) //rende un Mostro nuovo?
{
    p = new Mostro("ReferX");
    p.nome = "Cambia";
    return p;
}
public Mostro Clona_O (out Mostro p) //rende un Mostro nuovo?
{
    p = new Mostro("ResulX");
    p.nome = "Cambia";
    return p;
}
```

- Si associ il seguente codice al button1:

```
private void button1_Click
    (object sender, EventArgs e)
{
    Mannaro x = new Mannaro();
    Mostro y;
    y = Clona_V(x);
    x.Rivela();
    y.Rivela();
}
```

- Si associ il seguente codice al button2:

```
private void button2_Click
    (object sender, EventArgs e)
{
    Mostro x = new Mannaro();
    Mostro y;
    y = Clona_R (ref x);
    x.Rivela();
    y.Rivela();
}
```

- Si associ il seguente codice al button3:

```
private void button3_Click
    (object sender, EventArgs e)
{
    Mostro x = new Mannaro();
    Mostro y;
    y = Clona_O(out x);
    x.Rivela();
    y.Rivela();
}
```

- Prima di eseguire il progetto si provi ad anticipare il risultato!!!
- Prima di eseguire il progetto si provi ad anticipare il risultato!!!
- Prima di eseguire il progetto si provi ad anticipare il risultato!!!
- Poi si provi il progetto

- Salvare e chiudere il progetto

ESERCIZI

CLASSI DERIVATE

ESERCIZIO 1. PERSONE E STUDENTI

- ▶ Si definisca una classe **Persona** con attributi protetti nome, cognome, sesso e età;
- ▶ Si definisca un metodo **Set** per impostare tutti gli attributi
- ▶ Si definisca un metodo **Get** per leggere tutti gli attributi
- ▶ Si definisca una classe **Studente** derivata da **Persona**, con un attributo **voti**, come collezione di voti interi;
- ▶ Si definisca un metodo **media** che rende la media dei voti (**double**)

ESERCIZIO 2. COMPLESSI E QUATERNIONI

- ▶ Si consulti: http://it.wikipedia.org/wiki/Numero_complesso
- ▶ Si definisca una classe **Complesso** con attributi privati **reale** e **immaginario** di tipo **double**; l'oggetto deve modellare un numero complesso;
- ▶ Si definiscano tre costruttori rispettivamente con nessun, uno e due parametri
- ▶ Si definisca un metodo **Get** per leggere tutti gli attributi
- ▶ Si definisca un metodo **Modulo** che rende la radice quadrata della somma degli attributi al quadrato
$$|z| = \sqrt{x^2 + y^2}$$
- ▶ Si definisca un metodo **Coniugato** che inverte il segno della componente immaginaria
- ▶ Si definisca una classe **Quaternione** facendo riferimento a <http://it.wikipedia.org/wiki/Quaternione>

ESERCIZIO 3. SCOPA

- ▶ Si definisca una classe **Carta** con attributi privati **valore** e **seme**, un costruttore con due parametri (**valore** e **seme**) e un metodo **Lettura** con due parametri per risultato (**valore** e **seme**)
- ▶ Si definisca una classe **Mazzo** di carte con un attributo privato (una collezione di carte), un costruttore senza parametri che crea un mazzo da 40 carte, i metodi **Mescola** ed **Estrai**
- ▶ Si definisca una classe **Mano** di carte con un attributo privato (una collezione di carte), un costruttore senza parametri che svuota la collezione, i metodi **Lettura** (rende una collezione di carte) e **Gioca** (estrae una carta in base ad un indice)
- ▶ Si definisca una classe **Giocatore** con attributi privati **nome** (**string**), **mano** e **prese** (di tipo **mano**), un costruttore con il solo **nome** che svuota le collezioni, i metodi **Lettura** e **Prese** (rendono una collezione di carte) e **Gioca** (estrae una carta in base ad un indice)
- ▶

SOMMARIO

EREDITARIETÀ.....	2
Classi Derivate	2
Concetto	2
Definizione	3
Utilizzo	3
Scopo	3
Gerarchie ed alberi di classi	4
Ereditarietà degli attributi	4
Possesso degli attributi	4
Attributi omonimi	4
Accesso a attributi di antenati (base)	4
Inaccessibilità degli attributi privati	5
Accesso e protezione con protected.....	6
Ereditarietà dei metodi.....	6
Possesso dei metodi.....	6
Overload dei metodi	7
Invocazione di metodi overloaded di antenati (base)	7
Inaccessibilità dei metodi privati	8
Accesso ai metodi derivati: protected.....	8
Progetto guidato	8
Invocazione del costruttore base	9
Compatibilità tra tipi derivati.....	10
Assegnazione tra istanze	10
Commenti e riflessioni	10
Costruttore di un antenato	10
Collezioni di oggetti e classi derivate.....	10
Controllo di tipo: is	11
Cast di tipo: as	11
Progetto guidato	12
Parametri di tipo oggetto	13
Un Tipo riferimento	13
Passaggio per valore.....	13
Passaggio per riferimento	13
Passaggio per risultato	14
Progetto guidato	14
ESERCIZI.....	16
Classi derivate.....	16
Esercizio 1. Persone e studenti	16
Esercizio 2. Complessi e quaternioni.....	16
Esercizio 3. Scopa	16
SOMMARIO	17