Ereditarietà

L'**ereditarietà** è uno dei concetti base della programmazione orientata agli oggetti. Si definisce ereditarietà la possibilità per una classe (detta classe derivata) di ereditare da un'altra (la classe base) le variabili, i metodi, le proprietà e di estendere il concetto della classe base e specializzarlo.
Tornando all'**esempio delle automobili** potremmo avere come classi derivate da automobili le sottoclassi: «auto con cambio manuale» e «auto con cambio automatico». Entrambe le sottoclassi ereditano tutte le proprietà e i comportamenti della classe base ma specializzano il modo di cambiare le marce.
Tutte le classi del Framework .NET ereditano implicitamente dalla classe base `Object`.
In C# la **relazione di ereditarietà** tra le classi si esprime usando i due punti (`:`) che equivalgono a `Inherits` di VB.NET .
Riprendiamo una parte della classe `Persona` che abbiamo realizzato in precedenza:

```
public class Persona
{
  string mNome;
  protected string mCognome;

  public Persona(string Nome, string Cognome)
  {
    mNome = Nome;
    mCognome = Cognome;
  }

  public string Nome
  {
    get { return mNome; } set { mNome = value; }
  }

  //...
}
```

Osserviamo che la variabile `mCognome` è stata dichiarata come `protected`. Vogliamo ora **definire un nuovo concetto**, quello di `Studente`, ovvero una `Persona` dotata di un numero di matricola. Per fare questo, `Studente` deve ereditare da `Persona` (si può dire anche che `Studente` estende `Persona`):

```
public class Studente : Persona
{
  private int mMatricola;

  public Studente(string Nome, string Cognome, int Matricola)
  : base(Nome, Cognome)
  {
    mMatricola = Matricola;
  }

  public int Matricola
  {
    get { return mMatricola; }
    set { mMatricola = value; }
  }
}
```

Avendo definito questa relazione, nella classe Studente è possibile utilizzare tutto

ciò che in Persona è stato definito come `public`, `protected` o `internal`, sia esso una variabile, una routine o una proprietà.
Ad esempio, all'interno di `Studente` è possibile utilizzare la variabile `mCognome`, che in `Persona` è definita protected, mentre `mNome` non è visibile, poiché il modificatore di accesso predefinito è `private`.
Esaminiamo il frammento

```
: base(Nome, Cognome)
```

subito dopo il costruttore di `Studente`. Esso richiama il costruttore della classe base passandogli gli argomenti specificati: in questo modo è possibile impostare le variabili `mNome` e `mCognome` della classe `Persona` sugli stessi valori passati al costruttore di `Studente`.
Se avessimo voluto che il nome della persona, indipendentemente da quanto indicato nel costruttore di `Studente`, fosse sempre "Pippo", mentre il cognome fosse quello passato, sarebbe stato sufficiente scrivere:

```
: base("Pippo", Cognome)
```

Questo costrutto è necessario perché, quando si crea un oggetto di una classe che deriva da un'altra, prima del **costruttore** della classe derivata viene richiamato quello della classe base: nel nostro esempio, quando si istanzia un oggetto di tipo `Studente`, viene richiamato prima il costruttore della classe Persona, e, in seguito, quello di Studente.
Poiché la classe `Persona` non ha un costruttore senza argomenti, bisogna indicare esplicitamente tramite la **parola chiave «base»** quale costruttore richiamare. Se invece `Persona` avesse avuto un costruttore privo di argomenti, «base» non sarebbe servita.
Oltre che a questo scopo, «base» serve, in generale, per avere accesso ai metodi, alle variabili ed alle proprietà della classe che si sta derivando. Il corrispondente di «base» in Visual Basic è la parola chiave **MyBase**.
Provando a dichiarare oggetti di tipo `Persona` e `Studente`, si ottiene quanto segue:

```
Studente stud = new Studente("Marco", "", 0);

stud.Matricola = 232440;
// Corretto: Matricola è una property della classe Studente

stud.Cognome = "Minerva";
// Corretto: Cognome è una property ereditata da Persona
Persona pers = new Persona("Marco", "");

pers.Cognome = "Minerva";
// Corretto: Cognome è una property della classe Persona

pers.Matricola = 232440;
// Errato: la property Matricola non è visibile da Persona
```

Senza le istruzioni `: Persona` e `: base(Nome, Cognome)` nella classe `Studente`, eliminiremmo l'ereditarietà e otterremmo un messaggio di errore relativo all'istruzione:

```
stud.Cognome = "Minerva"
```

Grazie all'ereditarietà, dove è previsto l'uso di una certa classe, è quasi sempre possibile utilizzare una classe derivata. Questo è, probabilmente, uno degli aspetti più interessanti dell'ereditarietà.

Per **esempio**, supponiamo di avere una funzione che, ricevuti come argomenti due oggetti di tipo `Persona`, restituisce `true` se i loro nomi sono uguali:

```
public static bool NomiUguali(Persona p1, Persona p2)
{
  return (p1.Nome == p2.Nome);
}
```

Come argomenti a questa funzione, possono essere passati sia istanze della classe `Persona`, sia istanze di tutte le classi che ereditano da `Persona`, quindi anche oggetti di tipo `Studente`. Infatti, dal momento che esso eredita da `Persona`, dispone di tutte le proprietà pubbliche della classe base (in altre parole, uno **Studente è una Persona**, quindi può essere usato in tutti i contesti in cui è richiesta una `Persona`). Il seguente codice è corretto:

```
Persona pers = new Persona("Marco", "Rossi");
Studente stud = new Studente("Marco", "Minerva", 232440);
Console.WriteLine(NomiUguali(pers, stud)); // Stampa true
```

Un fatto molto interessante è che all'interno della routine `NomiUguali` la variabile `stud` viene vista come una `Persona`, e, quindi, di `stud` sono **visibili i metodi** e le proprietà pubbliche della classe `Persona`, non quelli di `Studente`.

L'ereditarietà è singola, in C# come in VB.NET, ovvero una classe può avere solo una classe base, a differenza del C++, che supporta l'ereditarietà multipla: in altre parole, non è possibile definire una classe C che eredita contemporaneamente da A e da B. Torneremo su questi concetti più avanti.