**Programmazione orientata agli oggetti**

http://www.aspitalia.com/articoli/asp.net2/Object-Oriented-Programming.aspx

La programmazione procedurale propria di linguaggi ormai datati come il Pascal o il C oppure di linguaggi di scripting più recenti come il VBScript si basa principalmente sull'organizzazione e suddivisione del codice in funzioni e procedure. A ciascuna operazione corrisponde una routine, che accetta un insieme di parametri e, in funzione di essi, produce eventualmente un risultato. Con un simile approccio, la separazione tra i dati e la logica applicativa risulta essere netta: una procedura riceve le informazioni utili all'esecuzione dell'operazione richiesta solamente sotto forma di parametri e il suo stato interno in genere viene del tutto perduto una volta terminata l'elaborazione. In molti casi questa netta separazione tra dati e codice rappresenta un vincolo importante tale da rendere questo paradigma di programmazione poco efficace.

La programmazione orientata agli oggetti (in modo abbreviato OOP, che in inglese corrisponde a Object Oriented Programming) introduce un modo diverso e, se si vuole, più efficiente per strutturare il codice e la logica applicativa. OOP è infatti un paradigma di programmazione che si basa sulla definizione e sull'utilizzo di una serie di entità tra loro collegate e interagenti, ciascuna delle quali è caratterizzata da un insieme di informazioni di stato e di comportamenti specifici. Queste entità sono denominate oggetti e ciascuna di esse può contenere contemporaneamente dati, funzioni e procedure. In questo modo una routine associata ad un oggetto può sfruttare lo stato interno dell'oggetto di appartenenza per ricavare le informazioni utili all'esecuzione dell'elaborazione prevista.

Il grosso vantaggio dell'approccio object-oriented rispetto agli altri paradigmi di programmazione consiste nel fatto che, per strutturare le applicazioni, lo sviluppatore si trova ad utilizzare una logica che è molto vicina a quella che è la percezione comune del mondo reale. Pensare ad oggetti significa infatti saper riconoscere gli aspetti che caratterizzano una particolare realtà e saper fornire di conseguenza una rappresentazione astratta in un'ottica OOP.

Per fare un esempio, si consideri la contabilità di un'azienda. Dal momento che le fatture o le bolle sono sicuramente elementi del mondo reale che riguardano l'amministrazione aziendale, un sistema gestionale realizzato secondo l'approccio OOP deve necessariamente includere una rappresentazione di queste entità sotto forma di oggetti. L'idea di fondo di OOP consiste infatti nell'individuare l'insieme degli oggetti che costituiscono e caratterizzano la realtà in esame e, al tempo stesso, nel definire il modo con cui essi interagiscono tra loro ed evolvono nel tempo. Ciascun oggetto individuato non necessariamente deve corrispondere ad un elemento del mondo reale; in taluni casi esso può essere una pura invenzione introdotta dallo sviluppatore per uno specifico scopo.

La struttura degli oggetti e l'insieme delle correlazioni tra essi esistenti concorrono a determinare l'architettura dell'applicazione.

Sia Visual C# 2005 che Visual Basic 2005 sono linguaggi fortemente tipizzati per la realizzazione di applicazioni che sfruttano il .NET Framework come ambiente di esecuzione. Entrambi sfruttano i principi della programmazione orientata agli oggetti e utilizzano le classi, di cui gli oggetti sono istanze particolari, come elementi fondamentali per l'organizzazione e la strutturazione del codice.

#### Principi fondamentali di OOP

Come si è avuto modo di dire nel corso del capitolo 1, i tipi (sia quelli di valore che di riferimento) sono gli elementi portanti di ogni applicazione che gira nell'ambito del Common Language Runtime. Oltre ai tipi built-in inclusi nella BCL del .NET Framework, lo sviluppatore può definire i suoi tipi personalizzati.

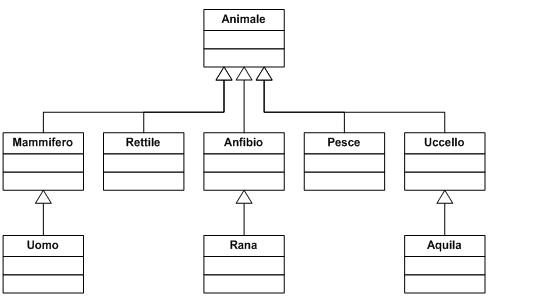
Le classi rappresentano i tipi di riferimento definiti dal programmatore. Gli elementi costitutivi di una classe (detti anche membri) possono essere sia dati, che funzioni e procedure.

I principi fondamentali su cui si fonda la programmazione orientata agli oggetti e che riguardano le classi sono sostanzialmente tre: ereditarietà, polimorfismo e incapsulamento. Avere dimestichezza con questi concetti è di fondamentale importanza per poter capire e, al tempo stesso, sfruttare al meglio i meccanismi che regolano la programmazione orientata agli oggetti.

##### Ereditarietà

Il principio di ereditarietà si basa sul fatto di poter definire un legame di dipendenza di tipo gerarchico tra classi diverse. Una classe deriva da un'altra se da essa ne eredita il comportamento e le caratteristiche. La classe figlia si dice classe derivata, mentre la classe padre prende il nome di classe base (o superclasse). La classe base contiene il codice comune a tutte le classi da essa derivate. Ogni classe derivata rappresenta invece una specializzazione della superclasse, ma eredita in ogni caso dalla classe base i comportamenti in essa definiti.

Per fare un esempio classico, si supponga di voler rappresentare il genere animale definendo una struttura di classi equivalente. È possibile individuare una prima classe base Animale, da cui tutte le altre devono derivare. A questo punto le classi derivate possono essere: Mammifero, Rettile, Anfibio, Pesce, Uccello. La classe Uomo deriva direttamente da Mammifero e indirettamente da Animale. La classe Rana deriva da Anfibio, mentre la classe Aquila deriva da Uccello; entrambe derivano indirettamente da Animale (vedi figura 4.1). Si noti che in tutti i casi vale la relazione "è un" (l'uomo è un mammifero, la rana è un anfibio, l'aquila è un uccello; l'uomo, la rana e l'aquila sono animali).



**Figura 4.1 - La classe base Animale e le sue classi derivate**

##### Polimorfismo

Il polimorfismo rappresenta il principio in funzione del quale diverse classi derivate possono implementare uno stesso comportamento definito nell'ambito della classe base in modo differente.

Riprendendo l'esempio proposto nel caso dell'ereditarietà, si considerino due comportamenti comuni a tutti gli animali: Respira e Mangia. È indubbio che nel mondo animale questi comportamenti vengano messi in atto secondo modalità peculiari a seconda delle specie: un carnivoro mangia in modo differente rispetto ad un erbivoro, un pesce respira in modo diverso rispetto ad un uccello. In base a queste considerazioni, i comportamenti Mangia e Respira definiti nelle diverse classi derivate da Animale possono essere implementati in modo specifico e del tutto indipendente dalle altre implementazioni (e, in particolar modo, da quello della classe base).

##### Incapsulamento

L'incapsulamento rappresenta il principio in base al quale una classe può mascherare la sua struttura interna e proibire ad altri oggetti di accedere ai suoi dati o di richiamare le sue funzioni che non siano direttamente accessibili dall'esterno.

Lo scopo principale dell'incapsulamento è quello di dare accesso allo stato e ai comportamenti di un oggetto solo attraverso un sottoinsieme di elementi pubblici. L'incapsulamento permette quindi di considerare una classe come una sorta di black-box, ovvero una scatola nera che permette di far vedere solo ciò che è necessario, mascherando ciò che non deve trasparire verso l'esterno.

#### Classi

Le classi sono i tipi di riferimento definiti dallo sviluppatore. Esse vengono dichiarate tramite la parola chiave class in C# o Class in Visual Basic, seguita dal nome identificativo. L'esempio 4.1 mostra la sintassi valida per la dichiarazione di una classe che rappresenta una persona.

**Esempio 4.1 - C#**

class Person  
{  
  \\ ...  
}

Come detto, una classe contiene sia dati che funzioni. I membri di una classe si suddividono in campi, proprietà, metodi ed eventi (questi ultimi vengono trattati in un paragrafo specifico nel corso del capitolo). I campi rappresentano le variabili interne alla classe (i dati), mentre i metodi sono le funzioni e le procedure (i comportamenti). Per i campi e i metodi valgono le stesse notazioni sintattiche viste nel corso dei capitoli precedenti per la dichiarazione delle variabili locali e delle routine (esempio 4.2).

**Esempio 4.2 - C#**

class Person  
{  
  string \_fullName = string.Empty;  
  int \_age = 18;  
  
  string GetFirstName()  
  {  
    // ...  
  }  
}

Le proprietà sono membri che permettono di accedere in lettura e scrittura ai campi di una classe. In genere una proprietà definisce una coppia di metodi (uno per la lettura di un campo e uno per la sua scrittura) e ad essi associa un nome identificativo. In Visual Basic per la dichiarazione di una proprietà si utilizza la parola chiave Property (esempio 4.3).

La dichiarazione di una proprietà avviene specificando due blocchi di codice contrassegnati dalle parole chiave get e set in C# o Get e Set in Visual Basic, corrispondenti rispettivamente alle operazioni di lettura e di scrittura (esempio 4.3). Questi due blocchi sono detti anche accessor (o funzioni di accesso). Per una proprietà non è obbligatorio definire sempre entrambi gli accessor.

**Esempio 4.3 - C#**

public class Person  
{  
  private string \_fullName = string.Empty;  
  public string FullName  
  {  
    get { return \_fullName; }  
    set { \_fullName = value; }  
  }  
}

Nell'esempio precedente le parole chiave come public e Public rappresentano i cosiddetti access modifier o modificatori di accessibilità, che hanno lo scopo di definire il grado di visibilità di un determinato elemento.

In base a quanto detto relativamente all'incapsulamento, le classi e, in particolare, i loro membri (campi, proprietà e metodi) possono avere un diverso grado di accessibilità. Questo significa che i membri di una classe possono essere mascherati e resi invisibili agli altri oggetti. L'insieme degli elementi accessibili è detto interfaccia della classe. Esistono diversi livelli di accessibilità a ciascuno dei quali corrisponde una parola chiave specifica (vedi tabella 4.1).

**Tabella 4.1 - Modificatori di accessibilità in C# e Visual Basic**

|  |  |
| --- | --- |
| **Visual C# 2005** | **Accessibilità** |
| public | Da tutte le classi. |
| protected | Solo dalle classi derivate. |
| private | Non accessibile. |
| internal | All'interno dell'assembly. |
| protected internal | Combinazione delle due. |

Il modificatore di accessibilità per una classe si riferisce alla sua visibilità in relazione all'assembly di appartenenza. Una classe pubblica può essere richiamata anche dagli oggetti presenti in altri assembly, una classe definita come internal in C# o Friend in Visual Basic è accessibile unicamente all'interno dell'assembly di appartenenza.

Gli access modifier sono facoltativi. In caso di omissione, sia in C# che in Visual Basic i membri di una classe vengono considerati di tipo privato, mentre le classi diventano visibili unicamente all'interno dell'assembly di appartenenza. In Visual Basic l'omissione del modificatore di accessibilità nella dichiarazione di un campo (vedi esempio 4.2) implica l'utilizzo della keyword Dim come nel caso della definizione di variabili locali (quindi Dim equivale a Private).

**Nota**  
È buona norma definire sempre i campi di una classe come privati e renderli eventualmente accessibili dall'esterno tramite una proprietà corrispondente.

Gli accessor hanno lo stesso livello di accessibilità definito per la proprietà a cui si riferiscono. In Visual C# 2005 è peraltro possibile differenziare l'accessibilità dei due blocchi, in modo tale, per esempio, che l'operazione di lettura sia pubblica, mentre l'operazione di scrittura abbia un livello di accesso limitato (esempio 4.4).

**Esempio 4.4 - C#**

public string FullName  
{  
  get { return \_fullName; }  
  private set { \_fullName = value; }  
}

Le istanze delle classi sono gli oggetti definiti nell'introduzione del capitolo. Dal momento che per ogni oggetto il CLR alloca un'area di memoria distinta nell'ambito del managed heap, ciascuno di essi è caratterizzato da un'identità univoca, da uno stato particolare (memorizzato nei campi) e da comportamenti specifici (metodi). Questo significa che oggetti diversi di una stessa classe mantengono in memoria copie differenti dei dati (quindi presentano uno stato distinto da tutti gli altri oggetti simili), anche se in generale condividono gli stessi comportamenti.

Un oggetto viene istanziato invocando un metodo particolare, che prende il nome di costruttore. Il costruttore di default non presenta parametri, ma per una classe è possibile definire più costruttori, ciascuno caratterizzato da una firma differente. Questo è possibile in quanto nell'ambito delle classi i metodi possono essere soggetti ad overload, ovvero possono esistere più metodi con lo stesso nome, ma con parametri diversi. La condizione di overload è applicabile a tutti i metodi di una classe in base alla firma. L'esempio 4.5 mostra un caso di overload del costruttore.

**Esempio 4.5 - C#**

public class Person  
{  
  private string \_fullName;  
  private readonly int \_age;  
  
  public Person()  
  {  
    \_fullName = string.Empty;  
    \_age = 18;  
  }  
  public Person(string name, int age)  
  {  
    \_fullName = name;  
    \_age = age;  
  }  
}  
  
Person x = new Person();  
Person y = new Person("Riccardo Golia", 35);

La sintassi per la dichiarazione dei costruttori è diversa nei due linguaggi. In C# il costruttore è contrassegnato dallo stesso nome della classe a cui si riferisce (non occorre utilizzare la keyword void). In Visual Basic il costruttore è una vera e propria procedura con nome identificativo New, ovvero lo stesso nome della keyword utilizzata per la creazione di un oggetto.

**Nota**  
In una classe è possibile utilizzare una o più costanti secondo le regole sintattiche già esposte nei capitoli dedicati ai linguaggi. In aggiunta è possibile definire un campo come readonly (la parola chiave in C# è readonly, in Visual Basic è ReadOnly), che può essere inizializzato e modificato unicamente nell'ambito di un costruttore (esempio 4.5). Una volta che una classe viene istanziata, i suoi campi readonly non sono più modificabili.

II costruttore viene richiamato durante la creazione di un oggetto utilizzando la parola chiave new in C# e la keyword New in Visual Basic (esempio 4.5). La creazione e la dichiarazione degli oggetti è stata affrontata anche nel corso dei capitoli dedicati ai linguaggi, ai quali si rimanda per ulteriori dettagli.

**Nota**  
Un altro tipo particolare di metodo è il distruttore. Questo metodo viene richiamato ogni qualvolta un oggetto viene finalizzato. A differenza di quanto succede nei linguaggi unmanaged, la distruzione non è deterministica, in quanto gli oggetti obsoleti contenuti nel managed heap vengono rimossi in modo autonomo e non prevedibile dal Garbage Collector. Per ulteriori informazioni sul distruttore di una classe, si rimanda alla documentazione in linea di MSDN.

Finora si è parlato solamente degli elementi associati ad una particolare istanza di una classe (detti membri d'istanza). Oltre ad essi è possibile definire un insieme di elementi associati direttamente al tipo e condivisi da tutte le sue istanze (detti membri statici).

I metodi, i campi e le proprietà statiche possono essere utilizzati senza la necessità di invocare il costruttore per il tipo a cui si riferiscono. Essi sono contrassegnati con una particolare keyword che li identifica come statici: in C# la parola chiave è static, mentre in Visual Basic è Shared.

I membri statici possono essere contenuti in una classe indipendentemente dal fatto che in essa siano presenti anche membri d'istanza. Una classe che contiene unicamente membri statici viene definita anch'essa statica e in C# viene contrassegnata con la stessa keyword usata per i singoli membri (esempio 4.6).

**Nota**  
Il costrutto Module (modulo) presente in Visual Basic è equiparabile ad una classe statica. I moduli in C# non esistono, per la loro definizione si usano direttamente le classi statiche. Per ulteriori dettagli sui moduli di Visual Basic si rimanda alla documentazione in linea di MSDN.

**Esempio 4.6 - C#**

// Classe statica  
public static class Calculator  
{  
  public static int Sum(int x, int y) { return x + y; }  
  public static int Diff(int x, int y) { return x - y; }  
  public static int Multiply(int x, int y) { return x \* y; }  
  public static int Divide(int x, int y) { return x / y; }  
}  
  
// I metodi vengono invocati direttamente sul tipo  
int i = Calculator.Sum(18, 8);  
int j = Calculator.Divide(18, 8);

**Nota**  
Come si può notare nell'esempio 4.6, per accedere ai membri statici di un tipo o agli elementi di un'istanza si utilizza il carattere "." (punto) seguito dal nome identificativo del membro e, nel caso dei metodi, dall'elenco dei parametri inclusi in una coppia di parentesi tonde. Questa notazione è valida per entrambi i linguaggi.

Internamente ai metodi e alle proprietà d'istanza di una classe è possibile utilizzare la keyword this in C# o Me in Visual Basic come riferimento all'istanza corrente. L'uso della parola chiave permette in molti casi di migliorare la leggibilità del codice e, al tempo stesso, di eliminare qualsiasi possibile ambiguità, in caso di omonimia degli identificatori (esempio 4.7).

**Esempio 4.7 - C#**

public class Person  
{  
  public string \_fullName = string.Empty;  
  
  public string GetFirstName()  
  {  
    // Utilizza il campo \_fullName dell'istanza corrente  
    return this.\_fullName.Split(' ')[0];  
  }  
}

Un aspetto interessante delle classi è rappresentato dal fatto che esse possono essere definite come parziali. La dichiarazione di una classe può essere infatti distribuita in più file separati, ciascuno dei quali include un sottoinsieme dei membri. La classe deve essere contrassegnata con la parola chiave partial in C# e Partial in Visual Basic per indicare che si tratta di una dichiarazione parziale e non completa (esempio 4.8).

L'omissione della parola chiave comporta un errore in fase di compilazione, in quanto in questo caso al compilatore risultano essere presenti due tipi aventi lo stesso nome identificativo. Durante la fase di compilazione infatti le varie dichiarazioni parziali vengono unite a formare un'unica entità finale.

**Esempio 4.8 - C#**

public partial class Person  
{  
  // Dichiarazione parziale  
}

#### Ereditarietà e polimorfismo

Le regole sintattiche utilizzate in C# e Visual Basic riguardanti l'ereditarietà e il polimorfismo sono molto diverse tra loro: ad uno stile molto compatto ed essenziale in C#, si contrappone uno stile molto verboso e ridondante in Visual Basic. Al di là di queste diversità sintattiche, i meccanismi e i concetti che stanno alla base dei due linguaggi sono del tutto equivalenti.

Sia in C# che in Visual Basic l'ereditarietà multipla non è contemplata. Per questo motivo una classe può derivare unicamente da una sola classe base. Questo aspetto non rappresenta tuttavia un vincolo di particolare rilievo: l'ereditarietà singola risulta essere decisamente sufficiente nella maggior parte dei casi.

In C# il legame di ereditarietà si indica con il carattere ":" (due-punti) posto subito dopo la dichiarazione della classe e seguito dal nome del tipo base. In Visual Basic il legame di ereditarietà tra due classi è indicato dalla parola chiave Inherits. Si noti che in Visual Basic il nome del tipo base va posto nella linea di codice successiva a quella della dichiarazione della classe derivata. L'esempio 4.9 mostra la dichiarazione della classe Employee che eredita dalla classe base Person.

**Esempio 4.9 - C#**

// Employee deriva da Person  
public class Employee : Person  
{  
  // ...  
}

I membri pubblici e protetti del tipo base sono accessibili anche nell'ambito delle classi derivate. Inoltre tutte le forme di visibilità, ovvero gli access modifier di ogni membro del tipo base, vengono ereditate a meno che esse non siano reimpostate diversamente nella classe derivata.

**Nota**  
I membri statici, in quanto associati al tipo e non alle istanze, non vengono derivati, indipendentemente dal loro livello di accessibilità. Una classe definita come statica non può essere derivata.

Oltre alla keyword che identifica l'istanza corrente (this in C# e Me in Visual Basic), esiste una parola chiave che viene usata nelle classi derivate come riferimento al tipo base. In C# questa keyword è base, mentre in Visual Basic è MyBase. Questa parola chiave torna molto utile per richiamare in modo esplicito i metodi e, in generale, i membri pubblici o protetti del tipo base nei metodi e negli accessor delle proprietà di una classe derivata (esempio 4.10).

**Esempio 4.10 - C#**

public class Employee : Person  
{  
  private string \_firstName;  
    
  // Viene richiamato implicitamente il costruttore  
  // di default della classe base  
  public Employee() : base()  
  {  
    \_firstName = base.GetFirstName();  
  }  
}

A seconda di come viene dichiarata, una classe può non essere derivabile piuttosto che non essere istanziabile direttamente.

Una classe che non può essere derivata deve essere contrassegnata con la keyword sealed in C# e NotInheritable in Visual Basic (esempio 4.11).

**Esempio 4.11 - C#**

// Customer deriva da Person e non può avere classi derivate  
public sealed class Customer : Person  
{  
  // ...  
}

Una classe che non può essere direttamente istanziata e che quindi deve essere obbligatoriamente derivata si dice astratta. Per dichiarare una classe astratta occorre specificare la parola chiave abstract in C# e MustInherit in Visual Basic. Le classi astratte sono caratterizzate dal fatto di avere tutti i metodi e le proprietà o un loro sottoinsieme definiti anch'essi come astratti.

Un membro astratto è un elemento della classe per il quale viene riportata semplicemente la dichiarazione insieme alla parola chiave specifica che in C# è sempre abstract, mentre in Visual Basic è MustOverride (esempio 4.12). In nessun caso viene specificata l'implementazione.

**Esempio 4.12 - C#**

// Person è una classe astratta e deve essere derivata  
public abstract class Person  
{  
  // Metodo astratto  
  public abstract GetFirstName();  
  
  // Proprietà astratta  
  public abstract string FullName { get; set; }  
}

Come si è avuto modo di dire nella prima parte del capitolo, il polimorfismo rappresenta la possibilità di ridefinire le proprietà e i metodi nelle classi derivate rispetto a quelli dichiarati nel tipo base (override). Affinché possa essere polimorfico, un membro deve essere marcato come astratto oppure come virtuale.

Dato che per un membro astratto nell'ambito del tipo base è presente unicamente la sua dichiarazione, è chiaro che esso debba essere implementato ogni volta in ciascuna delle classi derivate. Questo implica che ogni classe derivata è caratterizzata da comportamenti simili che devono peraltro adottare necessariamente strategie implementative diverse.

A differenza di quanto succede per i membri astratti, nel caso dei membri virtuali esiste sempre nella superclasse un'implementazione di base. Peraltro un membro virtuale può essere ridefinito in modo personalizzato nelle classi derivate. Questo significa che, oltre all'implementazione di base, possono esistere implementazioni diverse dello stesso membro, ciascuna delle quali afferenti ad una diversa classe derivata.

Se per proprietà e metodi astratti è sempre obbligatorio definire un'implementazione nella classe derivata, questo non è vero per i membri virtuali: nel caso in cui un membro virtuale non venga ridefinito nel tipo derivato, rimane valida la versione presente nella superclasse.

**Nota**  
I campi di una classe non possono essere soggetti a override, né definiti come astratti (del resto contengono solamente dati, non definiscono comportamenti). Gli unici membri che possono essere polimorfici sono i metodi e le proprietà.

La keyword che permette di contrassegnare un membro come virtuale è virtual in C# e Overridable in Visual Basic (esempio 4.13). Per eseguire l'override di un membro nell'ambito di una classe derivata, occorre specificare la parola chiave override in C# e Overrides in Visual Basic (esempio 4.13). Per indicare che un membro non può essere ulteriormente soggetto a override nelle classi derivate, occorre invece specificare la keyword sealed in C# e NotOverridable in Visual Basic.

**Nota**  
La parola chiave new in C# e la keyword Shadows in Visual Basic permettono di nascondere in modo esplicito un membro ereditato da una classe base. Questo significa che la versione derivata del membro sostituisce la versione della classe base. Lo scopo principale dello shadowing è infatti quello di proteggere la definizione dei membri di una classe. Se nel tipo base viene aggiunto un nuovo elemento con il nome uguale a quello del membro già definito nella classe derivata, le keyword new e Shadows impongono che i riferimenti attraverso la classe vengano comunque risolti nel membro della classe derivata, anziché nel nuovo elemento della superclasse.

È sempre bene ricordare che un membro virtuale viene risolto in base al tipo dell'istanza su cui viene richiamato, non in funzione del tipo del riferimento. L'esempio 4.13 illustra questo importante concetto, mostrando una casistica significativa di invocazione di un metodo virtuale.

**Esempio 4.13 - C#**

public class Person  
{  
  // Metodo virtuale  
  public virtual string GetFirstName()  
  {  
    // Implementazione di base del metodo  
  }    
}  
  
public class Employee : Person  
{  
  public override string GetFirstName()  
  {  
    // Nuova implementazione del metodo  
  }    
}  
  
Employee empl = new Employee(); // Istanza di Employee  
Person pers = empl; // Viene eseguito il  
string name = pers.GetFirstName(); // metodo di Employee

#### Interfacce

Una interfaccia è un tipo simile ad una classe astratta pura, ossia composta solamente da metodi e da proprietà astratte. Essa è infatti priva di qualsiasi implementazione, dato che il suo scopo è semplicemente quello di definire un contratto valido per le classi che la vanno ad implementare.

Il grosso vantaggio nell'utilizzo delle interfacce è rappresentato dal fatto che una classe può implementare più di un'interfaccia contemporaneamente. Questo aspetto va a compensare almeno parzialmente la mancanza dell'ereditarietà multipla per le classi.

Un'interfaccia viene definita specificando la parola chiave interface in C# e Interface in Visual Basic seguita dal nome identificativo. Non occorre specificare gli access modifier per i suoi elementi, ma per ognuno è sufficiente inserire la dichiarazione, omettendo qualsiasi forma di implementazione (esempio 4.14). In un'interfaccia possono essere incluse dichiarazioni di metodi, proprietà ed eventi.

**Esempio 4.14 - C#**

// Interfaccia che definisce un metodo per la stampa  
public interface IPrintable  
{  
  void Print(); // Metodo dell'interfaccia  
}

L'implementazione di un'interfaccia in C# segue le stesse regole sintattiche utilizzate nel caso dell'ereditarietà. In Visual Basic la sintassi è invece completamente diversa rispetto alla derivazione. Come è possibile vedere nell'esempio 4.15, la parola chiave da specificare è Implements ed essa va utilizzata sia nella dichiarazione del tipo (su una linea di codice dedicata), sia per ogni elemento dell'interfaccia che viene implementato nella classe (sulla stessa linea della dichiarazione del membro).

**Nota**  
Nel caso di implementazione multipla, si usa il carattere "," (virgola) per separare tra loro le diverse interfacce associate alla classe. In C# il tipo base e le interfacce implementate vengono indicate nello stesso elenco.

Oltre alla dichiarazione, l'esempio 4.15 riporta anche una casistica di utilizzo. Ogni istanza relativa ad una classe che implementa un'interfaccia può essere assegnata ad una variabile dell'interfaccia stessa. In questo caso i membri che possono essere richiamati sono solamente quelli associati all'interfaccia e non quelli esposti dalla classe che implementa l'interfaccia (per poterli invocare occorre necessariamente fare una cast al tipo).

**Esempio 4.15 - C#**

public class Employee : Person, IPrintable  
{  
  public void Print()  
  {  
    // ...  
  }  
}  
  
IPrintable p = new Employee();  
p.Print();  
  
string x = p.GetFirstName(); // È necessario il cast  
string x = (Employee)p.GetFirstName(); // OK

A questo punto sorge spontanea una domanda: quando usare le interfacce e quando invece utilizzare le classi astratte pure?

È stato detto che entrambe definiscono un contratto per le classi a cui sono associate, dato che internamente non contengono implementazioni, ma solo dichiarazioni. Peraltro le classi astratte pure forniscono un tipo di contratto "più forte", in quanto, oltre a definire i comportamenti per le classi associate, esse ne rappresentano anche il tipo base.

Diversamente le interfacce permettono una maggiore flessibilità, in quanto sono tipi indipendenti e trasversali rispetto alla gerarchia delle classi definita tramite i legami di ereditarietà. L'uso delle interfacce è quindi da preferire nel caso in cui si vogliano definire contratti di natura generale che possano essere usati indipendentemente dai legami di ereditarietà e che non impongano comportamenti specifici e di valenza esclusiva.

#### Eventi

Oltre ai campi, alle proprietà e ai metodi le classi possono includere un'ulteriore categoria di membri, gli eventi. Un evento è un elemento della classe che permette di inviare notifiche verso l'esterno in corrispondenza di certi avvenimenti che tendono a variare lo stato interno. A ciascun evento possono essere associate una o più azioni (dette handler) che vengono eseguite in corrispondenza della notifica dell'evento stesso.

Nel .NET Framework gli eventi sfruttano un tipo di dato particolare per agganciare i rispettivi handler. Si tratta dei delegate, che non sono altro che l'equivalente .NET dei puntatori a funzione presenti nei linguaggi unmanaged come il C++.

I delegate rappresentano per i metodi ciò che le interfacce rappresentano per le classi. Se un'interfaccia definisce un contratto per un particolare tipo, allo stesso modo un delegate stabilisce un contratto per un metodo basato sulla sua firma, indipendentemente dalla classe di appartenenza della funzione.

Il grosso vantaggio dei delegati rispetto ai puntatori a funzione è rappresentato dal fatto che essi sono tipizzati. Infatti ogni delegate è un'istanza del tipo System.Delegate e possono referenziare sia metodi relativi a istanze di oggetti, sia elementi statici associati ad un tipo. Le keyword delegate in C# e Delegate in Visual Basic definiscono implicitamente un tipo derivato dalla classe indicata.

Nella definizione di un delegate vanno indicati sia i parametri del metodo sia il tipo di ritorno. Ogni metodo associato ad un delegate ne deve assolutamente rispettare la firma, ovvero deve avere lo stesso numero e tipo di parametri, e deve restituire lo stesso tipo di dato.

**Esempio 4.16 - C#**

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

L'esempio 4.16 mostra il tipo base per tutti i delegate relativi agli handler degli eventi, ovvero il tipo System.EventHandler. Questo delegate è caratterizzato da due parametri e nessun tipo di ritorno (gli handler sono procedure, non funzioni). Il parametro sender di tipo object rappresenta l'oggetto da cui l'evento viene scatenato, mentre il parametro e di tipo System.EventArgs o un suo derivato rappresenta l'oggetto che include l'insieme degli argomenti che vengono passati al metodo di gestione dell'evento. Tutti gli handler di un evento devono avere una firma compatibile con quella definita dal delegate corrispondente.

Per definire un evento occorre specificare la parola chiave event in C# e Event in Visual Basic e fare riferimento ad un delegate di tipo System.EventHandler o ad un suo derivato (esempio 4.17).

ASP.NET utilizza abbondantemente il meccanismo degli eventi per intercettare le interazioni e i comandi dell'utente nelle pagine delle applicazioni. L'esempio 4.17 mostra un estratto della dichiarazione del web control Button di ASP.NET: gli eventi Command e Click forniscono il meccanismo di notifica che permette di segnalare che il bottone è stato selezionato e cliccato.

**Esempio 4.17 - C#**

public class Button : WebControl, IButtonControl, IPostBackEventHandler  
{  
  public event EventHandler Click;  
  public event CommandEventHandler Command;  
  // ...  
  // ...  
  // ...  
}

Ogni evento mantiene un lista di handler che vengono iterati per eseguire la notifica. Per associare un handler ad un evento occorre eseguire un'operazione di sottoscrizione ed inserire tramite il delegate un riferimento al metodo di gestione nell'ambito della lista dell'evento. L'esempio 4.18 mostra come associare in modo programmatico un handler all'evento Click di un controllo Button.

**Esempio 4.18 - C#**

// Il riferimento all'handler viene aggiunto alla lista  
// dell'evento Click del controllo MyButton  
MyButton.Click += new EventHandler(OnButtonClick);  
  
private void OnButtonClick(object sender, EventArgs e)  
{  
  // Gestione dell'evento  
}

#### Strutture

Oltre alle enumerazioni e alle classi, lo sviluppatore può definire in modo personalizzato anche i tipi di valore attraverso le strutture o struct.

Una struttura è un aggregato di dati simile ad una classe, dal momento che può contenere campi, metodi, proprietà ed eventi. La differenza rispetto ad una classe sta principalmente nel modo in cui una struttura viene persistita in memoria: essendo un tipo di valore, essa non viene inserita nel managed heap come avviene per le classi, ma viene allocata direttamente sullo stack.

**Nota**  
Tutti i tipi primitivi di valore che sono stati trattati nei capitoli precedenti sono strutture. Per esempio, System.Int32 piuttosto che System.Boolean sono struct.

Quanto è stato detto finora per le classi rimane generalmente valido anche per la definizione di strutture. Peraltro le struct presentano alcune limitazioni rispetto alle classi. La tabella 4.2 riassume le principali differenze esistenti tra classi e strutture.

**Tabella 4.2 - Differenze tra classi e strutture**

|  |  |
| --- | --- |
| **Classi** | **Strutture** |
| Possono definire campi, proprietà, metodi ed eventi. | Possono definire campi, proprietà, metodi ed eventi. |
| Supportano tutti i tipi di costruttori e l'inizializzazione dei membri. | Non supportano i costruttori di default e l'inizializzazione dei membri. |
| Supportano il metodo Finalize, ovvero il distruttore. | Non supportano il metodo Finalize. |
| Supportano l'ereditarietà. | Non supportano l'ereditarietà. |
| Sono tipi di riferimento. | Sono tipi di valore. |
| Vengono allocati nel managed heap. | Vengono allocati sullo stack. |

Come si può notare in tabella, le strutture non possono derivare da altre strutture, né essere un tipo base per altri tipi. Inoltre le struct non supportano direttamente il costruttore di default, in quanto il compilatore lo genera sempre in modo automatico, inizializzando i membri al loro valore predefinito. È in ogni caso possibile dichiarare in modo personalizzato costruttori con parametri in overload a quello di default.

Per definire una struttura occorre specificare la parola chiave struct in C# e Structure in Visual Basic. La dichiarazione segue le stesse regole sintattiche che si sono viste per le classi nel corso del capitolo (esempio 4.19).

**Esempio 4.19 - C#**

// Definizione di una struttura per i numeri complessi  
Public struct Complex  
{  
  public float Real; // Parte reale  
  public float Imaginary; // Parte immaginaria  
  
  // Costruttore con parametri  
  public Complex(float r, float i)  
  {  
    this.Real = r;  
    this.Imaginary = i;  
  }  
  
  // Metodo statico per la somma di due numeri complessi  
  public static Complex Sum(Complex x, Complex y)  
  {  
    return new Complex(x.Real + y.Real, x.Imaginary + y.Imaginary);  
  }  
}

*...omissis...*

#### Conclusioni

La programmazione orientata agli oggetti è un paradigma di programmazione che si basa sulla definizione e sull'utilizzo di una serie di entità tra loro collegate, ciascuna delle quali è caratterizzata da un insieme di informazioni di stato e di comportamenti specifici. Queste entità sono denominate oggetti e ciascuna di esse può contenere contemporaneamente dati, funzioni e procedure.

Le classi, di cui gli oggetti sono istanze particolari, rappresentano gli elementi fondamentali per l'organizzazione e la strutturazione del codice nei linguaggi managed del .NET Framework. Esse sono i tipi di riferimento definiti dallo sviluppatore, contenenti campi, proprietà, metodi ed eventi.

Per le classi valgono i principi fondamentali della programmazione orientata agli oggetti, ovvero ereditarietà, polimorfismo e incapsulamento. Nei linguaggi del .NET Framework e, in particolare, in C# e Visual Basic una classe può derivare solamente da un'altra classe (ereditarietà singola), può ridefinire in modo personalizzato le implementazioni dei suoi membri, anche se ereditati dal tipo base, e può mascherare la sua struttura interna, utilizzando in modo appropriato gli access modifier.

Le interfacce sono tipi astratti privi di qualsiasi implementazione, dato che il loro scopo è semplicemente quello di definire un contratto valido per le classi che le implementano. Il grosso vantaggio nell'utilizzo delle interfacce è rappresentato dal fatto che ciascuna classe può implementare più interfacce contemporaneamente. Questo aspetto compensa la mancanza dell'ereditarietà multipla nei linguaggi managed del .NET Framework.

Le strutture rappresentano i tipi di valore definiti dallo sviluppatore. Esse sono un aggregato di dati simile ad una classe, dal momento che possono contenere campi, metodi, proprietà ed eventi. Le differenze tra strutture e classi sono minime: principalmente le strutture non supportano l'ereditarietà e, dato che sono tipi di valore, vengono allocate sullo stack invece che nell'ambito del managed heap.

**Ereditarietà e classi derivate**

http://msdn.microsoft.com/it-it/library/ms228387.aspx

È possibile estendere la funzionalità di una classe esistente creando una nuova classe che deriva da essa. La classe derivata eredita le proprietà della classe base ed è possibile aggiungere o eseguire l'override di metodi e proprietà secondo le necessità.

In C# sia l'ereditarietà che l'implementazione delle interfacce vengono definite dall'operatore :, equivalente a extends e implements di Java. La classe base deve trovarsi sempre all'estrema sinistra nella dichiarazione della classe.

Analogamente a Java, C# non supporta l'ereditarietà multipla, ossia le classi non possono ereditare da più di una classe. È tuttavia possibile utilizzare le interfacce.

Nel codice riportato di seguito viene definita una classe denominata CoOrds con due variabili membro private x e y che rappresentano la posizione del punto. Queste variabili sono accessibili mediante proprietà denominate rispettivamente X e Y:

public class CoOrds

{

private int x, y;

public CoOrds() // constructor

{

x = 0;

y = 0;

}

public int X

{

get { return x; }

set { x = value; }

}

public int Y

{

get { return y; }

set { y = value; }

}

}

È possibile derivare una nuova classe, denominata ColorCoOrds, dalla classe CoOrds, come illustrato di seguito:

public class ColorCoOrds : CoOrds

ColorCoOrds eredita quindi dalla classe base tutti i campi e i metodi, a cui è possibile aggiungerne di nuovi per fornire funzionalità aggiuntive nella classe derivata, secondo le proprie esigenze. In questo esempio vengono aggiunti un membro privato e funzioni di accesso per aggiungere un colore alla classe:

public class ColorCoOrds : CoOrds

{

private System.Drawing.Color screenColor;

public ColorCoOrds() // constructor

{

screenColor = System.Drawing.Color.Red;

}

public System.Drawing.Color ScreenColor

{

get { return screenColor; }

set { screenColor = value; }

}

}

Il costruttore della classe derivata chiama in modo implicito quello della classe base, o superclasse secondo la terminologia Java. Nell'ereditarietà tutti i costruttori della classe base vengono chiamati prima di quelli della classe derivata, nell'ordine di visualizzazione della gerarchia delle classi.

* Cast del tipo in una classe base

Come in Java, non è possibile utilizzare un riferimento a una classe base per accedere ai membri e ai metodi di una classe derivata, anche se il riferimento alla classe base può contenere un riferimento valido a un oggetto del tipo derivato.

È possibile fare riferimento in modo implicito a una classe derivata con un riferimento al tipo derivato:

ColorCoOrds color1 = new ColorCoOrds();

CoOrds coords1 = color1;

In questo codice il riferimento alla classe base, coords1, contiene una copia del riferimento color1.

* Parola chiave base

È possibile accedere ai membri della classe base in una sottoclasse utilizzando la parola chiave base anche quando tali membri base sono sottoposti a override nella superclasse. È ad esempio possibile creare una classe derivata contenente un metodo con la stessa firma della classe base. Anteponendo al metodo la parola chiave new si specifica che si tratta di un metodo completamente nuovo appartenente alla classe derivata. È comunque possibile fornire un metodo per accedere al metodo originario nella classe base con la parola chiave base.

Si prenda come esempio una classe base CoOrds che dispone di un metodo denominato Invert() per l'inversione delle coordinate x e y. È possibile fornire un sostituto per questo metodo nelle classe derivata ColorCoOrds utilizzando il codice riportato di seguito:

public new void Invert()

{

int temp = X;

X = Y;

Y = temp;

screenColor = System.Drawing.Color.Gray;

}

Questo metodo inverte le coordinate x e y, quindi ne imposta il colore su grigio. È possibile fornire l'accesso all'implementazione base di questo metodo creando un altro metodo in ColorCoOrds, come quello riportato di seguito:

public void BaseInvert()

{

base.Invert();

}

Richiamare quindi il metodo base su un oggetto ColorCoOrds mediante una chiamata al metodo BaseInvert().

ColorCoOrds color1 = new ColorCoOrds();

color1.BaseInvert();

Tenere presente che si ottiene lo stesso effetto assegnando un riferimento alla classe base a un'istanza di ColorCoOrds e quindi accedendo ai relativi metodi:

CoOrds coords1 = color1;

coords1.Invert();

* Scelta dei costruttori

Gli oggetti della classe base vengono costruiti sempre prima delle eventuali classi che derivano da essi. Il costruttore della classe base viene pertanto eseguito prima di quello della classe derivata. Se la classe base ha più costruttori, la classe derivata può decidere quale costruttore chiamare. È ad esempio possibile modificare la classe CoOrds per aggiungere un secondo costruttore, come illustrato di seguito:

public class CoOrds

{

private int x, y;

public CoOrds()

{

x = 0;

y = 0;

}

public CoOrds(int x, int y)

{

this.x = x;

this.y = y;

}

}

È quindi possibile modificare la classe ColorCoOrds per utilizzare uno dei costruttori disponibili ricorrendo alla parola chiave base:

public class ColorCoOrds : CoOrds

{

public System.Drawing.Color color;

public ColorCoOrds() : base ()

{

color = System.Drawing.Color.Red;

}

public ColorCoOrds(int x, int y) : base (x, y)

{

color = System.Drawing.Color.Red;

}

}

* Override dei metodi

Una classe derivata può eseguire l'override del metodo di una classe base fornendo una nuova implementazione del metodo dichiarato. Java e C# differiscono tra loro per una caratteristica importante. I metodi Java sono contrassegnati come virtuali per impostazione predefinita, mentre i metodi C# devono essere contrassegnati come virtuali in modo esplicito utilizzando il modificatore virtual. Le funzioni di accesso alle proprietà, così come i metodi, possono essere sottoposte a override in modo analogo.

* Metodi virtuali

Un metodo che deve essere sottoposto a override in una classe derivata viene dichiarato con il modificatore virtual. In una classe derivata il metodo sottoposto a override viene dichiarato utilizzando il modificatore override. Il modificatore override indica un metodo o una proprietà di una classe derivata che sostituisce un metodo o una proprietà con lo stesso nome e la stessa firma nella classe base. Il metodo base, che deve essere sottoposto a override, deve essere dichiarato come virtual, abstract o override. Non è possibile eseguire in questo modo l'override di un metodo non virtuale o statico. Il metodo (o la proprietà) sottoposto a override e quello che esegue l'override devono avere gli stessi modificatori a livello di accesso.

Nell'esempio riportato di seguito viene illustrato un metodo virtuale, denominato StepUp, sottoposto a override in una classe derivata con il modificatore di override:

public class CountClass

{

public int count;

public CountClass(int startValue) // constructor

{

count = startValue;

}

public virtual int StepUp()

{

return ++count;

}

}

class Count100Class : CountClass

{

public Count100Class(int x) : base(x) // constructor

{

}

public override int StepUp()

{

return ((base.count) + 100);

}

}

class TestCounters

{

static void Main()

{

CountClass counter1 = new CountClass(1);

CountClass counter100 = new Count100Class(1);

System.Console.WriteLine("Count in base class = {0}", counter1.StepUp());

System.Console.WriteLine("Count in derived class = {0}", counter100.StepUp());

}

}

Quando si esegue questo codice, è possibile notare che il costruttore della classe derivata utilizza il corpo del metodo fornito nella classe base, consentendo di inizializzare il membro count senza duplicare il codice. L'output è il seguente:

Count in base class = 2

Count in derived class = 101

* Classi astratte

Una classe astratta dichiara come astratti uno o più metodi o proprietà. Per questi metodi non viene fornita un'implementazione nella classe che li dichiara, benché una classe astratta possa contenere anche metodi non astratti, ossia metodi per i quali è stata fornita un'implementazione. Non è possibile creare un'istanza di una classe astratta direttamente, ma solo come classe derivata. Queste classi derivate devono fornire le implementazioni per tutti i metodi e le proprietà astratti tramite la parola chiave override, a meno che il membro derivato non sia esso stesso dichiarato come astratto.

Nell'esempio riportato di seguito viene dichiarata una classe Employee astratta. Viene inoltre creata una classe derivata, denominata Manager, che fornisce un'implementazione del metodo Show() astratto definito nella classe Employee:

public abstract class Employee

{

protected string name;

public Employee(string name) // constructor

{

this.name = name;

}

public abstract void Show(); // abstract show method

}

public class Manager: Employee

{

public Manager(string name) : base(name) {} // constructor

public override void Show() //override the abstract show method

{

System.Console.WriteLine("Name : " + name);

}

}

class TestEmployeeAndManager

{

static void Main()

{

// Create an instance of Manager and assign it to a Manager reference:

Manager m1 = new Manager("H. Ackerman");

m1.Show();

// Create an instance of Manager and assign it to an Employee reference:

Employee ee1 = new Manager("M. Knott");

ee1.Show(); //call the show method of the Manager class

}

}

Questo codice richiama l'implementazione di Show() fornita dalla classe Manager e visualizza il nome del dipendente. L'output è il seguente:

Name : H. Ackerman

Name : M. Knott

* Interfacce

Un'interfaccia è una sorta di classe scheletro che contiene le firme dei metodi, senza le relative implementazioni. Le interfacce sono pertanto simili a classi astratte contenenti solo metodi astratti. Le interfacce C# sono molto simili e funzionano in modo analogo a quelle Java.

Tutti i membri di un'interfaccia sono pubblici per definizione. Inoltre, un'interfaccia non può contenere costanti, campi (membri dati privati), costruttori, distruttori o qualsiasi tipo di membro statico. Se viene specificato un modificatore per i membri di un'interfaccia, il compilatore genererà un errore.

È possibile derivare classi da un'interfaccia per implementare quest'ultima. Queste classi derivate devono fornire implementazioni per tutti i metodi dell'interfaccia, a meno che non vengano dichiarate astratte.

In Java e C# le interfacce vengono dichiarate nello stesso modo. In una definizione di interfaccia, per una proprietà vengono indicati solo il tipo e la caratteristica in sola lettura, sola scrittura o lettura/scrittura mediante le semplici parole chiave get e set. L'interfaccia riportata di seguito dichiara una proprietà in sola lettura:

public interface ICDPlayer

{

void Play(); // method signature

void Stop(); // method signature

int FastForward(float numberOfSeconds);

int CurrentTrack // read-only property

{

get;

}

}

Una classe può ereditare da questa interfaccia utilizzando i due punti. La classe di implementazione deve fornire le definizioni per tutti i metodi, nonché le eventuali funzioni di accesso alle proprietà, come illustrato di seguito:

public class CDPlayer : ICDPlayer

{

private int currentTrack = 0;

// implement methods defined in the interface

public void Play()

{

// code to start CD...

}

public void Stop()

{

// code to stop CD...

}

public int FastForward(float numberOfSeconds)

{

// code to fast forward CD using numberOfSeconds...

return 0; //return success code

}

public int CurrentTrack // read-only property

{

get

{

return currentTrack;

}

}

// Add additional methods if required...

}

### Implementazione di più interfacce

Una classe può implementare più interfacce utilizzando la sintassi riportata di seguito:

public class CDAndDVDComboPlayer : ICDPlayer, IDVDPlayer

Se una classe implementa più interfacce, eventuali ambiguità nei nomi dei membri vengono risolte utilizzando il qualificatore completo per il nome della proprietà o del metodo. In altri termini, la classe derivata può risolvere il conflitto utilizzando il nome completo del metodo per indicare a quale interfaccia appartiene, come in ICDPlayer.Play().

**Programmazione orientata a oggetti in C#**

http://msdn.microsoft.com/it-it/library/dd460654.aspx

Tutti i linguaggi gestiti in .NET Framework, ad esempio Visual Basic e C#, forniscono il supporto completo perla programmazione orientata a oggetti, inclusi incapsulamento, ereditarietà e polimorfismo.

L'incapsulamento indica che un gruppo di proprietà, metodi e altri membri correlati vengono considerati come una singola unità o un singolo oggetto.

L'ereditarietà indica la capacità di creare nuove classi sulla base di una classe esistente.

Il Polimorfismo indica la capacità di utilizzare più classi in modo intercambiabile, anche se in ognuna di esse le stesse proprietà o gli stessi metodi sono implementati in modi diversi.

In questa sezione vengono descritti i concetti seguenti:

Classi e oggetti

Membri di classe

Proprietà e campi

Metodi

Costruttori

Distruttori

Eventi

Classi annidate

Modificatori di accesso e livelli di accesso

Creazione di istanze di classi

Classi e membri statici

Ereditarietà

Override di membri

* **Classi e oggetti**

I termini classe e oggetto vengono talvolta utilizzati in modo intercambiabile. Di fatto, però, le classi descrivono il tipo degli oggetti, mentre gli oggetti sono istanze utilizzabili delle classi. L'atto di creare un oggetto viene pertanto chiamato creazione di istanze. Rifacendoci all'analogia precedente, la classe corrisponde al progetto iniziale e l'oggetto all'edificio realizzato in base a tale progetto.

Per definire una classe:

class SampleClass

{

}

C# fornisce anche una versione ridotta delle classi chiamate strutture utili quando è necessario creare una matrice di grandi dimensioni di oggetti e non si desidera utilizzare a tale scopo una quantità eccessiva di memoria.

Per definire una struttura:

struct SampleStruct

{

}

### *Membri di classe*

Ogni classe può disporre di membri della classe diversi che includono proprietà che descrivono i dati della classe, i metodi che definiscono il comportamento della classe e gli eventi che forniscono la comunicazione tra classi e oggetti diversi.

#### Proprietà e campi

I campi e le proprietà rappresentano le informazioni contenute in un oggetto.I campi sono simili a variabili in quanto possono essere letti o impostati direttamente.

Per definire un campo:

class SampleClass

{

public string sampleField;

}

Le proprietà dispongono di routine Get e Set, che forniscono un maggiore controllo sul modo in cui i valori vengono impostati o restituiti.

Sia C# che Visual Basic consentono di creare un campo privato per archiviare il valore della proprietà o utilizzare le cosiddette proprietà implementate automaticamente che creano automaticamente questo campo e forniscono la logica di base per le routine della proprietà.

Per definire una proprietà implementata automaticamente:

class SampleClass

{

public int SampleProperty { get; set; }

}

Se è necessario eseguire alcune operazioni aggiuntive per la lettura e la scrittura del valore della proprietà, definire un campo per archiviare il valore della proprietà e fornire la logica di base per archiviarlo e recuperarlo:

class SampleClass

{

private int \_sample;

public int Sample

{

// Return the value stored in a field.

get { return \_sample; }

// Store the value in the field.

set { \_sample = value; }

}

}

La maggior parte delle proprietà dispone di metodi o di routine per impostare e ottenere il valore della proprietà.È possibile, tuttavia, creare proprietà di sola lettura o di sola scrittura per impedirne la modifica o la lettura.In Visual Basic è possibile utilizzare le parole chiave ReadOnly e WriteOnly.In C#, è possibile omettere il metodo della proprietà get o set.Sia in Visual Basic che in C#, tuttavia, le proprietà implementate automaticamente non possono essere di sola lettura o di sola scrittura.

**Metodi**

Un metodo è un'azione che può essere eseguita da un oggetto.

Per definire un metodo di una classe:

class SampleClass

{

public int sampleMethod(string sampleParam)

{

// Insert code here

}

}

Una classe può disporre di diverse implementazioni, o overload, dello stesso metodo che differiscono per il numero di parametri o per i tipi di parametro.

Per essere in rapporto di overload con un metodo:

public int sampleMethod(string sampleParam) {};

public int sampleMethod(int sampleParam) {}

Nella maggior parte dei casi si dichiara un metodo all'interno di una definizione della classe.Sia Visual Basic che C#, tuttavia, supportano inoltre i metodi di estensione che consentono di aggiungere metodi a una classe esistente al di fuori della definizione effettiva della classe.

**Costruttori**

I costruttori sono metodi di classe che vengono eseguiti automaticamente durante la creazione di un oggetto di un tipo specifico.I costruttori in genere inizializzano i membri dati del nuovo oggetto.Un costruttore può essere eseguito solo una volta alla creazione di una classe.Inoltre, il codice nel costruttore viene sempre eseguito prima di qualsiasi altro codice in una classe.Tuttavia, è possibile creare più overload del costruttore esattamente come per qualsiasi altro metodo.

Per definire un costruttore per una classe:

public class SampleClass

{

public SampleClass()

{

// Add code here

}

}

**Distruttori**

I distruttori sono utilizzati per distruggere istanze di classi.In .NET Framework, il Garbage Collector gestisce l'allocazione e il rilascio di memoria per gli oggetti gestiti di un'applicazione.Potrebbero, tuttavia, essere necessari distruttori per pulire eventuali risorse non gestite create dall'applicazione.Può esistere un solo distruttore per classe.

**Eventi**

Tramite gli eventi una classe o un oggetto sono in grado di segnalare ad altre classi o oggetti una situazione di interesse.La classe che invia (o genera) l'evento è chiamata autore e le classi che ricevono (o gestiscono) l'evento sono chiamate sottoscrittori.Per ulteriori informazioni sugli eventi e sulla corrispondente modalità di generazione e gestione, vedere [Gestione e generazione di eventi](http://msdn.microsoft.com/it-it/library/edzehd2t.aspx).

Per dichiarare un evento in una classe, utilizzare la parola chiave [event](http://msdn.microsoft.com/it-it/library/8627sbea.aspx).

Per generare un evento, richiamare il delegato dell'evento.

**Classi annidate**

Una classe definita all'interno di un'altra classe è denominata annidata.Per impostazione predefinita, la classe annidata è privata.

class Container

{

class Nested

{

// Add code here.

}

}

Per creare un'istanza della classe annidata, utilizzare il nome della classe dei contenitori seguita dal punto, quindi dal nome della classe annidata:

Container.Nested nestedInstance = new Container.Nested()

**Modificatori di accesso e livelli di accesso**

Tutte le classi e i membri della classe possono specificare quale livello di accesso forniscono alle altre classi utilizzando i modificatori di accesso.

Sono disponibili i seguenti modificatori di accesso:

|  |  |
| --- | --- |
| **Modificatore di C#** | **Definizione** |
| [public](http://msdn.microsoft.com/it-it/library/yzh058ae.aspx) | Il tipo o il membro è accessibile da altro codice nello stesso assembly o in un altro assembly che vi fa riferimento. |
| [private](http://msdn.microsoft.com/it-it/library/st6sy9xe.aspx) | Il tipo o il membro è accessibile solo dal codice nella stessa classe. |
| [protected](http://msdn.microsoft.com/it-it/library/bcd5672a.aspx) | Il tipo o il membro è accessibile solo dal codice nella stessa classe o in una classe derivata. |
| [internal](http://msdn.microsoft.com/it-it/library/7c5ka91b.aspx) | Il tipo o il membro è accessibile dal codice nello stesso assembly ma non da un altro assembly. |
|  |  |

Per ulteriori informazioni, vedere [Modificatori di accesso (Guida per programmatori C#)](http://msdn.microsoft.com/it-it/library/ms173121.aspx).

### Creazione di istanze di classi

Per creare un oggetto, è necessario creare un'istanza di una classe.

SampleClass sampleObject = new SampleClass();

Dopo avere creato un'istanza di una classe, è possibile assegnare i valori alle proprietà e ai campi dell'istanza e richiamare i metodi della classe.

// Set a property value.

sampleObject.sampleProperty = "Sample String";

// Call a method.

sampleObject.sampleMethod();

Per assegnare i valori alle proprietà durante il processo di creazione dell'istanza della classe, utilizzare gli inizializzatori di oggetto:

// Set a property value.

SampleClass sampleObject = new SampleClass

{ FirstProperty = "A", SecondProperty = "B" };

**Classi e membri statici**

Un membro statico della classe è una proprietà, una routine o un campo condiviso da tutte le istanze di una classe.

Per definire un membro statico:

static class SampleClass

{

public static string SampleString = "Sample String";

}

Per accedere al membro statico, utilizzare il nome della classe senza creare un oggetto di questa classe:

Console.WriteLine(SampleClass.SampleString);

Le classi statiche in C# dispongono solo di membri statici e non è possibile crearne un'istanza. I membri statici non possono accedere inoltre a proprietà, campi o metodi non statici.

* **Ereditarietà**

L'ereditarietà permette di creare una nuova classe che riutilizza, estende e modifica il comportamento definito in un'altra classe. La classe i cui membri vengono ereditati è denominata classe base, mentre la classe che eredita tali membri è denominata classe derivata. I linguaggi gestiti in.NET Framework non supportano l'ereditarietà multipla, i.e. è possibile specificare una sola classe di base per una classe derivata.

Per ereditare da una classe base:

class DerivedClass:BaseClass{}

Per impostazione predefinita, tutte le classi possono essere ereditate.Tuttavia, è possibile specificare se una classe non deve essere utilizzata come classe base oppure creare una classe utilizzabile solo come classe base.

Per specificare che una classe non può essere utilizzata come classe base:

public sealed class A { }

Per specificare che una classe può essere utilizzata solo come classe base e che non è possibile crearne un'istanza:

public abstract class B { }

**Override di membri**

Per impostazione predefinita, in una classe derivata vengono ereditati tutti i membri della classe base relativa.Se si desidera modificare il comportamento del membro ereditato, è necessario eseguirne l'override.È possibile definire una nuova implementazione del metodo, della proprietà o dell'evento nella classe derivata.

I seguenti modificatori consentono di controllare le modalità di override di proprietà e metodi:

|  |  |
| --- | --- |
| **Modificatore di C#** | **Definizione** |
| [virtual (Riferimenti per C#)](http://msdn.microsoft.com/it-it/library/9fkccyh4.aspx) | Consente a un membro della classe di essere sottoposto a override in una classe derivata. |
| [override (Riferimenti per C#)](http://msdn.microsoft.com/it-it/library/ebca9ah3.aspx) | Esegue l'override di un membro virtuale (sottoponibile a override) definito nella classe base. |
| [abstract (Riferimenti per C#)](http://msdn.microsoft.com/it-it/library/sf985hc5.aspx) | Richiede che un membro della classe venga sottoposto a override nella classe derivata. |
| [Modificatore new (Riferimenti per C#)](http://msdn.microsoft.com/it-it/library/435f1dw2.aspx) | Nasconde un membro ereditato da una classe base. |

**Programmazione ad Oggetti - Glossario**

**Namespace**: è un insieme di classi o altri namespace a cui viene attribuito un nome.

**Incapsulamento**: è il meccanismo che riunisce insieme il codice e i dati da esso manipolati e che mette entrambi al sicuro da interferenze o errati utilizzi.

**Costruttore**: è un particolare metodo di una classe che porta lo stesso nome della classe e permette l’inizializzazione automatica dell’oggetto su cui intende operare.

**Distruttore**: svolge l’operazione complementare del costruttore , ha lo stesso nome del costruttore ma viene preceduto da “tilde” (~).

**Static**: quando la dichiarazione di una variabile membro è preceduta dalla parola chiave static , si chiede al compilatore di creare una sola copia di tale variabile e di utilizzare tale copia per tutti gli oggetti della classe.

**Puntatore**: l’indirizzo di memoria di un oggetto viene detto puntatore. E possibile gestire il puntatore dell’oggetto che richiama un metodo utilizzando la parola chiave this .

**Overloading**: consiste nell’impiegare lo stesso nome per 2 o più metodi che si differenziano per numero e tipo di parametri passati. La caratteristica principale dell’overloading dei metodi è il fatto che questi devono differire per quanto riguarda il tipo e/o il numero dei parametri, dunque 2 metodi non possono differire solo per il tipo di dato restituito.

**Polimorfismo**: Il Polimorfismo consente ad un’interfaccia di controllare l’accesso a una categoria generale di azioni; trova la sua realizzazione nell’uso di metodi e di costruttori modificati tramite overloading.

**Ereditarietà**: è lo strumento che permette di costruire nuove classi utilizzando quelle già realizzate. La classe ereditata viene chiamata classe base. La classe che riceve l’eredità è detta classe derivata. A sua volta la classe derivata può fungere da classe base per un’altra classe derivata; in questo modo è possibile riprodurre, a più livelli, il meccanismo di ereditarietà.

**Override**: Per impostazione predefinita, in una classe derivata vengono ereditati tutti i membri della classe base relativa.Se si desidera modificare il comportamento del membro ereditato, è necessario eseguirne l'override.È possibile definire una nuova implementazione del metodo, della proprietà o dell'evento nella classe derivata.