**Programmazione orientata ad oggetti (OOP)**

http://www.html.it/guide/guida-programmazione-orientata-agli-oggetti/

**Tecniche di Programmazione**

Storicamente sono state utilizzate “vecchie” metodologie di programmazione che sono state lentamente ma inesorabilmente messe da parte con la diffusione dell’OOP. In questa lezione verranno descritte brevemente tali metodologie al fine di comprendere meglio le differenze con la programmazione ad oggetti.

### Programmazione Non Strutturata

Con la programmazione non strutturata il programma è costituito da **un unico blocco di codice detto “main”** dentro il quale vengono manipolati i dati in maniera totalmente sequenziale.

È importante notare che tutti i dati sono rappresentati soltanto da **variabili di tipo globale**, ovvero visibili da ogni parte del programma ed allocate in memoria per tutto il tempo che il programma stesso rimane in esecuzione.

È facile capire che un simile contesto è **fortemente limitato e pieno di svantaggi**. Ad esempio, sarà facile incappare in spezzoni di codice ridondanti o ripetuti che non faranno altro che rendere presto ingestibile ed “illeggibile” il codice, causando oltretutto un enorme spreco di risorse di sistema.

Nella figura seguente, viene rappresentata graficamente l’architettura di questo tipo di paradigma di sviluppo.

**Figura 1. Schema della programmazione non strutturata**  


### Programmazione Procedurale

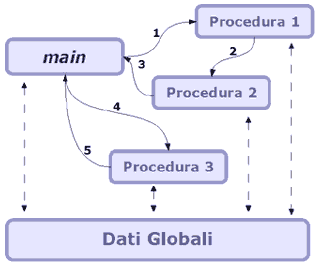
Un notevole passo in avanti, rispetto alla Programmazione Non Strutturata, venne fatto con l’avvento della Programmazione Procedurale. I programmatori in Visual Basic 5.0 o 6.0 sapranno certamente di cosa si sta parlando (anche se il Visual Basic utilizza anche il paradigma di Programmazione Modulare, descritta nel successivo paragrafo).

Il concetto base qui è quello di raggruppare i pezzi di programma ripetuti in porzioni di codice utilizzabili e richiamabili ogni volta che se ne presenti l’esigenza: **nascevano le Procedure**.

Ogni procedura può essere vista come un sottoprogramma che svolge una ben determinata **funzione** (ad esempio, il calcolo della radice quadrata di un numero) e che è visibile e richiamabile dal resto del codice.

Inoltre ogni procedura ha la capacità di poter utilizzare uno o più **parametri** che ne consentono una maggiore duttilità. Anche il flusso del programma è decisamente diverso rispetto a quello visto nella programmazione non strutturata: infatti, il main continua ad esistere ma al suo interno appaiono soltanto le invocazioni alle procedure definite nel programma.

Quando una procedura ha terminato il suo compito il controllo ritorna nuovamente al main (o alla procedura che ne ha effettuato l’**invocazione**) che esegue una nuova chiamata ad un’altra procedura. fino alla terminazione del programma. Un semplice schema puo’ facilitare la comprensione di tale concetto:

**Figura 2. Flusso di un programma con programmazione procedurale**  


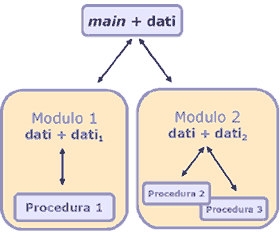
Il grande vantaggio della programmazione procedurale, rispetto alla precedente non strutturata consiste in un notevole **abbattimento del numero di errori**, che deriva dal fatto che se una procedura è corretta allora vi è la certezza che essa restituirà ad ogni invocazione dei risultati corretti in output.

### Programmazione Modulare

La programmazione modulare rappresenta un’ ulteriore conquista. Sorgeva l’esigenza di poter **riutilizzare le procedure** messe a disposizione da un programma in modo che anche altri programmi ne potessero trarre vantaggio.

Così, l’idea fu quella di raggruppare le **procedure aventi un dominio comune** (ad esempio, procedure che eseguissero operazioni matematiche) in moduli separati.

Quando sentiamo parlare di **librerie** di programmi, in sostanza si fa riferimento proprio a **moduli di codice** indipendenti che ben si prestano ad essere inglobati in svariati programmi.

**Figura 3. Struttura di un programma «modulare»**  


Il risultato, dunque, adesso è che un singolo programma non è più costituito da un solo file (in cui è presente il main e tutte le procedure) ma da diversi moduli (uno per il main e tanti altri quanti sono i moduli a cui il programma fa riferimento).

È importante, inoltre, dire che i singoli moduli possono contenere anche dei dati propri che, in congiunzione ai dati del main, vengono utilizzati all’interno delle procedure in essi contenute.

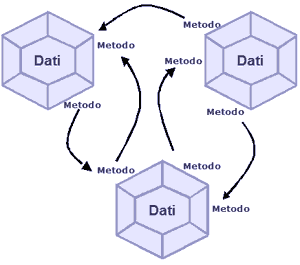
### Programmazione ad Oggetti

La programmazione procedurale/modulare ha rappresentato il punto di riferimento nello sviluppo applicativo per tanti anni. Gradualmente ma inevitabilmente però, man mano che gli orizzonti della programmazione diventavano sempre più ampi, si andarono evidenziando i **limiti** di tale metodologia.

In particolare, un programma procedurale mal si prestava a realizzare il concetto di “Componente Software”, ovvero di un prodotto in grado di garantire le caratteristiche di riusabilità, modificabilità e manutenibilità.

Una delle cause di tale limite è da ricercare sicuramente nel fatto che esiste un evidente scollamento tra i dati e le strutture di controllo che agiscono su di essi; in altre parole i moduli risultano avere un approccio orientato alla procedura piuttosto che ai dati.

Con l’avvento della programmazione ad oggetti (i cui concetti saranno dettagliati a partire dal prossimo capitolo) questi limiti vennero superati. Contrariamente alle tecniche fin qui descritte, il paradigma OOP è basato sul fatto che esiste una serie di oggetti che interagiscono vicendevolmente, scambiandosi messaggi ma mantenendo ognuno il proprio stato ed i propri dati. Graficamente:

**Figura 4. Colloquio tra oggetti**  


La [programmazione ad oggetti](http://programmazione.html.it/guide/leggi/38/guida-programmazione-orientata-agli-oggetti/), naturalmente, pur cambiando radicalmente l’approccio mentale all’analisi progettuale non ha fatto a meno dei vantaggi derivanti dall’uso dei moduli. Al contrario, tale tecnica è stata ulteriormente raffinata avvalendosi delle potenzialità offerte dalla programmazione ad oggetti.

**Rappresentazione della realtà**

L’**idea principale** che sta dietro la Programmazione ad Oggetti risiede, in buona parte, nel mondo reale. Come spesso accade, gli esempi pratici forniscono una comprensione più chiara ed immediata della teoria e, pertanto, si userà tale approccio per fissare bene in mente i concetti basilari del mondo OOP.

Si prenda in considerazione un comune masterizzatore, come quelli ormai diffusi nella stragrande maggioranza dei moderni PC. Di tale oggetto si conoscono le sue caratteristiche ma quasi nessuno (se non coloro che lo hanno progettato e assemblato) è a conoscenza dei componenti elettronici e dei sofisticati meccanismi che ne regolano il corretto funzionamento, ne’ a qualcuno salterebbe in mente di smontare un masterizzatore prima di acquistarlo per vedere come è fatto all’interno (supposto che il negoziante sia disposto ad accettare una tale richiesta, decisamente fuori dal comune!).

Il concetto che sta dietro alla programmazione ad oggetti nasce dallo stesso principio: ciò che importa non è l’implementazione interna del codice (che corrisponde ai componenti elettronici, nel caso del masterizzatore) ma, piuttosto, **le caratteristiche e le azioni** che un componente software è in grado di svolgere e che mette a disposizione (espone) all’esterno.

Dunque, con riferimento a quanto gia detto in precedenza, un programma che segue **il paradigma OOP** è costituito da un numero variabile di tali componenti, che ora denominiamo oggetti, i quali interagiscono tra di essi attraverso lo scambio di messaggi.

Proseguendo nell’**esempio** del masterizzatore e supponendo di voler implementare un oggetto software che ne gestisca le funzionalità, potremmo iniziare a definirne le **caratteristiche**. Ad esempio:

* Marca
* Velocità di scrittura su supporti CD-R
* Velocità di scrittura su supporti CD-RW
* Velocità di lettura
* Interfaccia
* Dimensione del buffer dati

Mentre, per quanto riguarda le **azioni** potremmo considerare le seguenti:

* Scrivi su CD
* Scrivi su DVD
* Leggi CD
* Espelli CD

In parole molto semplici, potremmo dire che le azioni altro non sono che “le cose che un oggetto è in grado di fare” mentre le sue caratteristiche rappresentano i dati che le azioni stesse possono utilizzare per eseguire le operazioni che da esse ci si aspetta.

In OOP, le caratteristiche di un oggetto vengono denominate **proprietà** e le azioni sono dette **metodi**. Questi sono due concetti fondamentali ed è molto importante che siano ben chiari prima di introdurre nuove definizioni.

Altrettanto importante, però, è ribadire il contesto in cui abbiamo iniziato a ragionare: non dobbiamo pensare più ad un programma che racchiuda tutto in un unico grande calderone ma dobbiamo iniziare a **pensare “ad oggetti”**. Dobbiamo, cioè, essere in grado di identificare gli oggetti che entrano in gioco nel programma che vogliamo sviluppare e saperne gestire l’interazione degli uni con gli altri.

In un **programma di tipo procedurale**, si è soliti iniziare a ragionare in maniera top-down, partendo cioè dal main e creando mano a mano tutte le procedure necessarie. Tutto questo non ha più senso nella programmazione ad oggetti, dove serve invece definire prima le classi e poi associare ad esse le proprietà ed i metodi opportuni.

**Metodi e proprietà**

**I metodi: le azioni che un oggetto è in grado di compiere**

Come detto, un metodo rappresenta una azione che può essere compiuta da un oggetto. Una delle domande principali da porsi quando si vuole creare un oggetto è: “Cosa si vuole che sia in grado di fare?”.

In effetti, quando ci si pone questa domanda, si sta involontariamente dando per buono il fatto che qualsiasi oggetto sia in qualche modo capace di eseguire delle azioni, trattandolo come se avesse natura umana. Eppure, anche se inizialmente può apparire quantomeno curioso e insolito, questo approccio rappresenta un importante riferimento nella fase di disegno e creazione degli oggetti.

In generale, potremmo delineare almeno **tre buone regole** per identificare i metodi da associare ad un oggetto:

1. **Un oggetto che abbia uno o due soli metodi deve fare riflettere**. Potrebbe essere perfettamente lecito definire un oggetto del genere (ed è sicuramente possibile farlo praticamente) ma, spesso, un oggetto creato con questi requisiti indica la necessità di “mescolarlo” con un altro oggetto con simile definizione.
2. Ancora più **da evitare sono gli oggetti con nessun metodo**. È bene che un oggetto incapsuli (vedremo meglio in seguito cosa si intenda con tale terminologia) dentro sé sia informazioni (le proprietà), sia azioni (i metodi, appunto). In linea di massima, un oggetto senza metodi può facilmente essere convertito in uno o più attributi da assegnare ad un altro oggetto.
3. Sicuramente **da evitare sono anche gli oggetti con troppi metodi**. Un oggetto, in generale, dovrebbe avere un insieme facilmente gestibile di proprie responsabilità. Assegnare ad un oggetto troppe azioni, potrebbe rendere ardua la manutenzione futura dello stesso oggetto. È consigliabile, in questo caso, cercare di spezzare l’oggetto in due oggetti più piccoli e semplificati.

**Le proprietà: informazioni su cui opera un oggetto**

Le proprietà rappresentano **i dati dell’oggetto**, ovvero le informazioni su cui i metodi possono eseguire le loro elaborazioni.

**Uno degli errori più comuni** che si commette quando si definiscono le proprietà di un oggetto è quello di associare ad esso quante più proprietà possibili, ritenendo che questo possa, in qualche modo, facilitare la stesura del programma. Un oggetto, invece, per essere ben definito deve contenere le proprietà che, effettivamente, gli competono e non tutte quelle che gli si potrebbero comunque attribuire.

Questa regola di buona programmazione nasce, oltretutto, dall’esigenza di rendere più facile la fase di disegno e quella di debug che altrimenti risulterebbero certamente complesse.

Per evitare, dunque, l’inconveniente di ritrovarsi dei super-oggetti sarà bene **porsi la seguente domanda** nella fase di definizione: «quali proprietà sono necessarie affinché l’oggetto sia in grado di eseguire le proprie azioni?»

**Le Classi**

Probabilmente, il termine più importante e rappresentativo nella Programmazione ad Oggetti è quello di Classe.

Riprendiamo ancora l’esempio del masterizzatore. Sappiamo benissimo che non esiste un solo tipo di masterizzatore ma, a secondo delle caratteristiche, è possibile citarne tanti modelli: ci sono, ad esempio, quelli in grado di scrivere su CD e DVD o quelli che scrivono solo su CD.

Sicuramente, però, tutti sono in grado di eseguire la scrittura di dati o file multimediali su CD e tutti hanno, altresì, le proprietà esposte nelle lezioni precedenti.

Diremo allora che più oggetti software che **hanno le stesse proprietà e gli stessi metodi** possono essere raggruppati in una classe ben definita di oggetti: nel nostro caso particolare, nella classe masterizzatore.

Dunque, una classe rappresenta, sostanzialmente, una **categoria particolare di oggetti** e, dal punto di vista della programmazione, è anche possibile affermare che una classe funge da tipo per un determinato oggetto ad essa appartenente (dove con tipo si intende il tipo di dato, come lo sono gli interi o le stringhe).

Diremo, inoltre, che un particolare oggetto che appartiene ad una classe costituisce un’**istanza della classe** stessa. In altre parole, un’istanza della classe masterizzatore sarà costituita da un oggetto di tale classe che è in grado di scrivere solo su CD mentre un’altra istanza potrà essere rappresentata da un oggetto che è in grado di masterizzare anche i DVD.

È anche possibile creare due o più istanze separate di **oggetti uguali** (sarebbe meglio dire di oggetti che hanno le proprietà valorizzate allo stesso modo) ma ciò non vorrà dire che in un determinato istante nel corso del programma tutte queste istanze eseguano la medesima operazione o che, in generale, siano necessariamente correlati.

L’univocità di ogni istanza viene definita con il termine di **identità** (identity): ogni oggetto ha una propria identità ben distinta da quella di tutte le altre possibili istanze della stessa classe a cui appartiene l’oggetto stesso.

Naturalmente, le proprietà di un oggetto possono avere valori che variano nel tempo. Ad esempio, la velocità di scrittura potrebbe essere selezionata dall’utente e, pertanto, potrebbe variare nel corso del programma. Si definisce **stato di un oggetto**, l’insieme dei valori delle sue proprietà in un determinato istante di tempo. Se cambia anche una sola proprietà di un oggetto, il suo stato varierà di conseguenza.

L’insieme dei metodi che un oggetto è in grado di eseguire viene definito, invece, **comportamento** (behavior). Insomma, appare sempre più chiaro come un oggetto rappresenti un’entità a sé stante, ben definita che, nel corso dell’elaborazione, sia soggetta ad una creazione, ad un suo utilizzo e, infine, alla sua distruzione.

Inoltre, un oggetto non dovrebbe mai manipolare direttamente i dati interni (le proprietà) di un altro oggetto ma ogni tipo di comunicazione tra oggetti dovrebbe essere sempre gestita tramite l’**uso di messaggi**, ovvero tramite le chiamate ai metodi che un oggetto espone all’esterno.

**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 per la 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.

**L’incapsulamento**

Nei programmi Object Oriented, come abbiamo già osservato, si è soliti mettere in stretta relazione tra loro un pezzo di informazione con il comportamento specifico che agisce su tale informazione. Questo è ciò che abbiamo definito oggetto.

L’incapsulamento è proprio legato al concetto di **“impacchettare” in un oggetto i dati e le azioni** che sono riconducibili ad un singolo componente.

Un altro modo di guardare all’incapsulamento, che abbiamo già accennato, è quello di pensare a suddividere un’applicazione in piccole parti (gli oggetti, appunto) che raggruppano al loro interno alcune funzionalità legate tra loro.

Ad esempio, pensiamo ad un conto bancario. Le informazioni utili (le proprietà) potranno essere rappresentate da: il numero di conto, il saldo, il nome del cliente, l’indirizzo, il tipo di conto, il tasso di interesse e la data di apertura.

Le azioni che operano su tali informazioni (i metodi) saranno, invece: apertura, chiusura, versamento, prelevamento, cambio tipologia conto, cambio cliente e cambio indirizzo. L’oggetto Conto incapsulerà queste informazioni e azioni al suo interno.

Come risultato, ogni modifica al sistema informatico della banca che implichi una modifica ai conti correnti, potrà essere implementata semplicemente nell’oggetto Conto.

Un altro vantaggio derivante dall’incapsulamento è quello di limitare gli effetti derivanti dalle modifiche ad un sistema software.

Chiariamo meglio il concetto avvalendoci di un classico esempio. Si pensi, ad un sistema software come ad una distesa di acqua e ad una richiesta di modifica del software come ad una enorme masso. Gettando il masso nell’acqua si creeranno tante onde e spruzzi in tutte le direzioni. Tali onde si propagheranno per tutta la distesa di acqua, rimbalzeranno sulla costa e, alla fine, si scontreranno con altre onde. In definitiva, il masso lanciato in acqua avrà causato un notevole effetto di disturbo su tutta la superficie su cui si estende la distesa d’acqua.

Proviamo, adesso, ad utilizzare il concetto di incapsulamento, definito in precedenza, applicandolo a tale esempio. La distesa d’acqua viene suddivisa in tante piccole porzioni di acqua, ognuna delle quali è ben delimitata con delle barriere per evitare che l’acqua fuoriesca da essa. Se gettiamo un masso su una di queste porzioni di acqua avremo ancora una volta il verificarsi di onde in tutte le direzioni ma, questa volta, esse termineranno la loro azione scontrandosi con le barriere delimitatrici. In sostanza, tutte le altre porzioni di acqua non verranno minimamente intaccate.

Cerchiamo ora di esaminare una situazione che preveda una modifica da apportare rimanendo nel contesto del sistema bancario di cui abbiamo definito proprietà e metodi. Supponiamo che la banca in questione abbia deciso che se un determinato cliente ha un conto di credito nella stessa banca, tale conto possa essere utilizzato come copertura per eventuali scoperti sul conto corrente.

Per gestire tale situazione, in un sistema non incapsulato, si procederebbe con un approccio decisamente poco efficace. Ovvero, non sapendo dove siano localizzati nel codice i punti in cui viene gestita l’operazione di prelievo, l’unica alternativa possibile sarebbe quella di ricercare dappertutto, aggiungendo la nuova funzionalità al programma ogni volta che si identifichi un punto da modificare.

Con buona probabilità, se saremo stati bravi, saremo riusciti ad individuare l’80 per cento dei punti da modificare. Con l’utilizzo dell’incapsulamento sarà, invece, sufficiente identificare l’oggetto che gestisce l’azione di prelevamento (ovvero l’oggetto Conto) ed eseguire la modifica all’interno di esso, completando la variazione del sistema richiesta senza intaccare tutto il resto del sistema stesso.

### Implementare l’incapsulamento:

public class cubo

{

private int lunghezza;

private int larghezza;

private int altezza;

public int Lunghezza

{

get

{

return lunghezza;

}

set

{

lunghezza = value;

}

}

public int Larghezza

{

get

{

return larghezza;

}

set

{

larghezza = value;

}

}

public int Altezza

{

get

{

return altezza;

}

set

{

altezza = value;

}

}

// Metodo pubblico che visualizza il volume del cubo, usando le proprietà

// interne della classe

public int visualizzaVolume()

{

System.Console.WriteLine("Volume: " + (lunghezza \* larghezza \* altezza));

}

}

**L’ereditarietà**

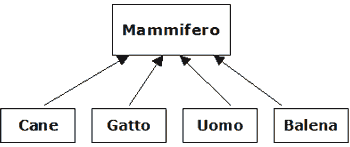
L’ereditarietà costituisce il secondo principio fondamentale della programmazione ad oggetti. In generale, essa rappresenta un meccanismo che consente di creare nuovi oggetti che siano basati su altri già definiti.

Si definisce **oggetto figlio** (child object) quello che eredita tutte o parte delle proprietà e dei metodi definiti nell’**oggetto padre** (parent object).

È semplice poter osservare esempi di ereditarietà nel mondo reale. Ad esempio, esistono al mondo centinaia tipologie diverse di mammiferi: cani, gatti, uomini, balene e così via. Ognuna di tali tipologie di mammiferi possiede alcune caratteristiche che sono strettamente proprie (ad esempio, soltanto l’uomo è in grado di parlare) mentre esistono, d’altra parte, determinate caratteristiche che sono comuni a tutti i mammiferi (ad esempio, tutti i mammiferi hanno il sangue caldo e nutrono i loro piccoli).

Nel mondo Object Oriented, potremmo riportare tale esempio definendo un oggetto Mammifero che inglobi tutte le caratteristiche comuni ad ogni mammifero. Da esso, poi, deriverebbero gli altri child object: Cane, Gatto, Uomo, Balena, etc.

L’oggetto cane, per citarne uno, erediterà, quindi, tutte le caratteristiche dell’oggetto mammifero e a sua volta conterrà delle caratteristiche aggiuntive, distintive di tutti i cani come ad esempio: ringhiare o abbaiare. Il paradigma OOP, ha quindi carpito l’idea dell’ereditarietà dal mondo reale, come mostrato nella figura seguente:

**Figura 1. Esempio di classificazione**  


e, pertanto lo stesso concetto viene applicato ai sistemi software che utilizzano tale tecnologia.

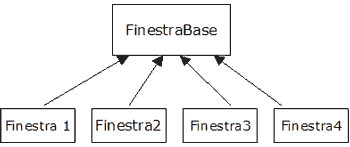
Uno dei maggiori vantaggi derivanti dall’uso dell’ereditarietà è la maggiore facilità nella manutenzione del software. Infatti, rifacendoci all’esempio dei mammiferi, se qualcosa dovesse variare per l’intera classe dei mammiferi, sarà sufficiente modificare soltanto l’oggetto padre per consentire che tutti gli oggetti figli ereditino la nuova caratteristica.

Ad esempio, se i mammiferi diventassero improvvisamente (e anche inverosimilmente!) degli animali a sangue freddo, soltanto l’oggetto padre mammifero necessiterà di tale variazione. Il gatto il cane, l’uomo, la balena, e tutti gli altri oggetti figli erediterebbero automaticamente la caratteristica di avere il sangue freddo, senza nessuna modifica.

Un esempio che metta in luce la potenza dell’ereditarietà, in un programma Windows object oriented potrebbe avere come oggetto fulcro le finestre associate al programma stesso. Supponiamo, ad esempio, che tale software utilizzi, in totale, 100 finestre differenti, visualizzabili a secondo del contesto in cui sta navigando l’utente.

Se, un giorno, si volesse inserire un campo comune a tutte le finestre del programma, in che modo sarà bene procedere? Se non avremo utilizzato la potenza dell’ereditarietà l’unica strada percorribile sarà quella di andarsi a prendere una per una tutte le definizioni delle finestre e inserire il nuovo campo.

Se, invece, si sarà utilizzato in maniera efficiente il paradigma Object Oriented, definendo una classe FinestraBase contenente tutte le caratteristiche comuni ad ogni finestra e derivando da tale classe tutte le finestre in gioco nel programma, allora la modifica sarà banalmente (allo stesso modo dell’esempio sui mammiferi) quella di inserire il nuovo campo nella classe FinestraBase. Graficamente:

**Figura 2. Classificazione di oggetti software**  


Ancora un esempio. In un sistema bancario, si potrebbe utilizzare l’ereditarietà per definire tutte le tipologie di conto esistenti. Supponiamo che i conti possibili siano quattro : conti correnti bancario, libretti di risparmio, carte di credito e certificati di deposito.

Tutte queste differenti tipologie di conto hanno in comune alcune caratteristiche: un numero di conto, un tasso di interesse ed un sottoscrittore. In tal modo potremo creare un oggetto padre chiamato Conto che contenga tutte le caratteristiche comuni prima enunciate.

Gli oggetti figli, in aggiunta, avranno le loro specifiche caratteristiche definite nelle rispettive classi. Ad esempio, la carta di credito avrà un limite di spesa mentre il il conto corrente bancario avrà uno o più libretti di assegni associati. Anche in questo caso, le eventuali modifiche apportate alla classe padre saranno ereditate automaticamente da tutte le classi figlie.

È importante, però chiarire un aspetto importante quando si parla di caratteristiche ereditate. Non sempre, infatti, un determinato metodo definito nella classe padre può produrre risultati corretti e congruenti con tutte le classi figlie. Ad esempio, supponiamo di aver definito una classe padre denominata Uccello, dalla quale faremo derivare le seguenti classi figlie: Passerotto, Merlo e Pinguino.

Nella classe padre, avremo definito il metodo vola(), in quanto rappresenta un comportamento comune a tutti gli uccelli. In tal modo, secondo quanto si è detto in questo paragrafo, tutte le classi figlie non avranno la necessità di implementare tale metodo ma lo erediteranno dalla classe Uccello. Purtroppo, però, nonostante il pinguino appartenga alla categoria degli uccelli, è noto che esso non è in grado di volare, seppur provvisto di ali.

In questo caso, il metodo vola() definito nella classe Uccello, sicuramente valido per la stragrande maggioranza di uccelli, non sarà utile (anzi, sarà proprio sbagliato) per la classe Pinguino. Come comportarsi in questi casi?

In OOP, ogni oggetto derivante da una classe padre ha la possibilità di ignorare uno o più metodi in essa definiti riscrivendo tali metodi al suo interno. Questa caratteristica è nota come **overriding**.

Utilizzando la tecnica dell’overriding, la classe Pinguino reimplementerà al suo interno il metodo vola(), conservando, comunque, la possibilità di richiamare in qualunque momento, anche il metodo definito nella classe padre. In quest’ultimo caso si parlerà di **overriding parziale**.

**Implementare l’ereditarietà:**

public class Impiegato

{

private String nome;

private double salario;

private String matricola;

private int anniDiServizio;

public Impiegato(String n, double s, String m, int ads)

{

nome = n;

salario = s;

matricola = m;

anniDiServizio = ads;

}

public void incrementaSalario(double percentuale)

{

salario \*= 1 + percentuale / 100;

}

public void stampaInfo()

{

System.Console.WriteLine (nome + " " + salario + " " + matricola);

}

public String Nome

{

get { return nome; }

}

public int AnniServizio

{

get{ return anniDiServizio; }

}

}

class Manager : Impiegato

{

private String nomeSegretaria;

public Manager(String n, double s, String m, int ads) : base(n, s, m, ads)

{

nomeSegretaria = String.empty;

}

public void incrementaSalario(double percentuale)

{

// Aggiunge alla percentuale lo 0.5% per ogni anno di servizio

double bonus = 0.5 \* AnniServizio;

base.incrementaSalario(percentuale + bonus);

}

public String Segretaria()

{

get { return nomeSegretaria; }

set { nomeSegretaria = value; }

}

}

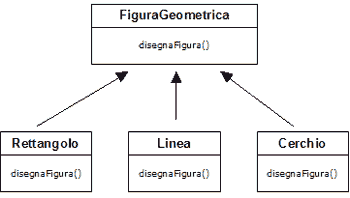
**Il polimorfismo**

Il terzo elemento fondamentale della programmazione ad Oggetti è il polimorfismo. Letteralmente, la parola polimorfismo indica la possibilità per uno stesso oggetto di assumere più forme.

Per rendere l’idea più chiara, utilizzando ancora una volta un esempio del mondo reale, si pensi al diverso comportamento che assumono un uomo, una scimmia e un canguro quando eseguono l’azione del camminare. L’uomo camminerà in modo eretto, la scimmia in maniera decisamente più goffa e curva mentre il canguro interpreterà tale azione saltellando.

Riferendoci ad un sistema software ad oggetti, il polimorfismo indicherà l’attitudine di un oggetto a mostrare più implementazioni per una singola funzionalità.

Per esempio, supponiamo di voler costruire un sistema software in grado di disegnare delle figure geometriche. Per tale sistema avremo definito una classe padre chiamata FiguraGeometrica dalla quale avremo fatto derivare tutte le classi che si occupano della gestione di una figura geometrica ben precisa. Per maggiore chiarezza riportiamo quanto detto nel diagramma seguente:

**Figura 1. Esempio di polimorfismo**  


Quando l’utente desidera rappresentare una di tali figure, sia essa una linea, un cerchio o un rettangolo, egli eseguirà una determinata azione che produrrà l’invio al sistema di un messaggio che, a sua volta, scatenerà l’invocazione del metodo disegnaFigura della classe FiguraGeometrica..

Con l’utilizzo del polimorfismo, il sistema è in grado di capire autonomamente quale figura geometrica debba essere disegnata ed invocarne direttamente il metodo disegnaFigura appartenente alla classe figlia coinvolta.

In un sistema non ad oggetti (e, quindi, senza la possibilità di utilizzare il polimorfismo) un simile comportamento necessiterebbe, dal punto di vista del codice, di un costrutto tipo switch – case come il seguente, tutto implementato all’interno di un’unica classe:

function disegnaFigura()  
{  
  CASE FiguraGeometrica.Tipo  
    Case ‘Cerchio’  
      FiguraGeometrica.DisegnaCerchio()  
    Case ‘Rettangolo’  
      FiguraGeometrica.DisegnaRettangolo()  
    Case ‘Linea’  
      FiguraGeometrica.DisegnaLinea()  
  END CASE  
}

Al contrario, con l’utilizzo del polimorfismo, il tutto si riconduce ad una semplice chiamata del tipo:

function Disegna(  
{  
  FiguraGeometrica.disegnaFigura()  
}

Dietro una semplice codifica di questo tipo si nasconde tutta la potenza del polimorfismo: il sistema chiama in modo automatico il metodo disegnaFigura() dell’oggetto che è stato selezionato dall’utente, senza che ci si debba preoccupare se si tratti di cerchio, rettangolo o linea. In altre parole, potremmo dire che il polimorfismo consente ad oggetti differenti (ma collegati tra loro) la flessibilità di rispondere in modo differente allo stesso tipo di messaggio.

Uno dei maggiori benefici del polimorfismo, come in effetti di un po’ tutti gli altri principi della programmazione ad oggetti, è la facilità di manutenzione del codice. Per rendere l’idea, basta domandarsi cosa accadrebbe se l’applicazione precedente volesse implementare anche la funzionalità di disegnare un triangolo.

Nel caso non polimorfico, bisognerebbe aggiungere una nuova funzione DisegnaTriangolo all’oggetto FiguraGeometrica e poi modificare la funzione DisegnaFigura globale, esaminata in precedenza, aggiungendo ad essa due nuove righe per gestire il caso della nuova tipologia di figura da inserire.

Con il polimorfismo invece, molto più semplicemente, basterà creare l’oggetto Triangolo e implementare in esso il metodo disegnaFigura(). L’invocazione di quest’ultimo avverrà, come per le altre figure geometriche già definite, in modo del tutto trasparente e automatico.

### Implementare il polimorfismo

class Albero

{

public virtual void cresce()

{

Console.WriteLine("Metodo cresce della classe Albero");

}

}

class Abete : Albero

{

public override void cresce()

{

Console.WriteLine("Metodo cresce della classe Abete");

}

}

class Output

{

static void Main(string[] args)

{

Albero alb;

Abete ab;

alb = new Albero();

ab = new Abete();

alb.cresce(); // output --> "Metodo cresce della classe Albero"

ab.cresce(); // output --> "Metodo cresce della classe Abete"

alb = new Abete();

alb.cresce(); output --> "Metodo cresce della classe Abete"

}

}

**Classi, oggetti, ereditarietà**

* [Classi e oggetti](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Classes)
  + [Membri di classe](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Members)

[Proprietà e campi](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Properties)

[Metodi](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Methods)

[Costruttori](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Constructors)

[Distruttori](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Destructors)

[Eventi](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Events)

[Classi annidate](http://msdn.microsoft.com/it-it/library/dd460654.aspx#NestedClasses)

* + [Modificatori di accesso e livelli di accesso](http://msdn.microsoft.com/it-it/library/dd460654.aspx#AccessModifiers)
  + [Creazione di istanze di classi](http://msdn.microsoft.com/it-it/library/dd460654.aspx#InstantiatingClasses)
  + [Classi e membri statici (condivisi)](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Static)
* [Ereditarietà](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Inheritance)
  + [Override di membri](http://msdn.microsoft.com/it-it/library/dd460654.aspx#Overriding)

[**Classi e oggetti**](javascript:void(0))

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

{

}

Sia Visual Basic che C# forniscono 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:

public class SampleClass

{

public string sampleField;

}

class Program

{

static void Main()

{

SampleClass sc = new SampleClass();

Console.WriteLine("sc.sampleField " +sc.sampleField);

sc.sampleField = "Valore";

Console.WriteLine("sc.sampleField " + sc.sampleField);

Console.ReadKey();

}

}

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.

Esempio di proprietà implementata automaticamente e non…

Di seguito è riportato un esempio di una funzione di accesso get in una proprietà denominata Seconds:

class TimePeriod

{

private double \_seconds;

public double Seconds

{

get { return \_seconds; }

set { \_seconds = value; }

}

}

Nell'esempio seguente viene mostrata una funzione di accesso get in una proprietà implementata automaticamente:

class TimePeriod2

{

public double Hours { get; set; }

}

#### 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) {}

#### 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.

public class SampleClass

{

public SampleClass()

{

// Add code here

}

}

Quando [classe](http://msdn.microsoft.com/it-it/library/0b0thckt.aspx) o [struttura](http://msdn.microsoft.com/it-it/library/ah19swz4.aspx) viene creato, il costruttore viene chiamato. I costruttori hanno lo stesso nome della classe o della struttura e in genere inizializzano i membri dati del nuovo oggetto.

Nell'esempio seguente viene definita una classe denominata Taxi con un costruttore semplice. Viene quindi creata un'istanza di questa classe con l'operatore [new](http://msdn.microsoft.com/it-it/library/51y09td4.aspx). Il costruttore Taxi viene richiamato dall'operatore new immediatamente dopo l'allocazione della memoria per il nuovo oggetto.

public class Taxi

{

public bool isInitialized;

public Taxi()

{

isInitialized = true;

}

}

class TestTaxi

{

static void Main()

{

Taxi t = new Taxi();

Console.WriteLine(t.isInitialized);

}

}

I costruttori predefiniti, ossia che non accettano parametri, vengono richiamati ogni volta che viene creata un'istanza di un oggetto utilizzando l'operatore new e non vengono forniti argomenti a new.

Le classi possono definire costruttori che accettano parametri. I costruttori che accettano parametri devono essere chiamati tramite un'istruzione new. Le classi possono inoltre definire più costruttori e non devono necessariamente definire un costruttore predefinito. Ad esempio:

public class Employee

{

public int salary;

public Employee(int annualSalary)

{

salary = annualSalary;

}

public Employee(int weeklySalary, int numberOfWeeks)

{

salary = weeklySalary \* numberOfWeeks;

}

}

Per creare questa classe, è possibile utilizzare una delle seguenti istruzioni:

Employee e1 = new Employee(30000);

Employee e2 = new Employee(500, 52);

#### 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.

[Note](javascript:void(0))

* I distruttori non possono essere definiti nelle strutture,ma solamente nelle classi.
* Per una classe è possibile utilizzare un solo distruttore.
* I distruttori non possono essere ereditati e non è possibile eseguirne l'overload.
* I distruttori non possono essere chiamati.Vengono richiamati automaticamente.
* Un distruttore non accetta modificatori né parametri.

Di seguito è riportato un esempio di dichiarazione di un distruttore per la classe Car:

class Car

{

~Car() // destructor

{

// cleanup statements...

}

}

#### 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()

### 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" };

### 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. |

Esempio con public:

public class SampleClass

{

public string sampleField;

}

class Program

{

static void Main()

{

SampleClass sc = new SampleClass();

sc.sampleField = "Valore";

Console.WriteLine("sc.sampleField " + sc.sampleField);

Console.ReadKey();

}

}

// Output:

sc.sampleField Valore

Esempio con protected:

Errata:

public class SampleClass

{

protected string sampleField;

}

class Program

{

static void Main()

{

SampleClass sc = new SampleClass();

sc.sampleField = "Valore";

Console.WriteLine("sc.sampleField " + sc.sampleField);

Console.ReadKey();

}

}

// Messaggio di errore:

ConsoleApplication1.SampleClass.sampleField' è inaccessibile a causa del livello di protezione

Corretta:

public class SampleClass

{

protected string sampleField;

public void Stampa()

{

sampleField = "Valore";

Console.WriteLine("sampleField " + sampleField);

}

}

class Program

{

static void Main()

{

SampleClass sc = new SampleClass();

sc.Stampa();

Console.ReadKey();

}

}

// Output:

sampleField Valore

Con classe derivata:

public class SampleClass

{

protected string sampleField;

}

class SampleClassD : SampleClass

{

static void Main()

{

//L'istruzione sc.sampleField = "Valore" genera un errore

//perché utilizzata all'interno del metodo Main statico

//e non in un'istanza della classe SampleClassD.

//Errore 1: Impossibile accedere al membro protetto

//'ConsoleApplication1.SampleClass.sampleField'

//tramite un qualificatore di tipo

//'ConsoleApplication1.SampleClass'.

//Il qualificatore deve essere di tipo

//'ConsoleApplication1.SampleClassD' o derivato da esso.

//

SampleClass sc = new SampleClass();

sc.sampleField = "Valore";

Console.WriteLine("sc.sampleField " + sc.sampleField);

//L'istruzione scD.sampleField = "Valore" è corretta perchè

//accede ai membri protetti della classe base

//direttamente dalla classe derivata.

SampleClassD scD = new SampleClassD();

scD.sampleField = "Valore";

Console.WriteLine("scD.sampleField " + scD.sampleField);

Console.ReadKey();

}

}

Esempio con private:

Errata:

public class SampleClass

{

private string sampleField;

}

class Program

{

static void Main()

{

SampleClass sc = new SampleClass();

sc.sampleField = "Valore";

Console.WriteLine("sc.sampleField " + sc.sampleField);

Console.ReadKey();

}

}

// Messaggio di errore:

'ConsoleApplication1.SampleClass.sampleField' è inaccessibile a causa del livello di protezione

Con classe derivata:

Errata:

public class SampleClass

{

private string sampleField;

}

class SampleClassD : SampleClass

{

static void Main()

{

SampleClassD scD = new SampleClassD();

scD.sampleField = "Valore";

Console.WriteLine("scD.sampleField " + scD.sampleField);

Console.ReadKey();

}

}

// Messaggio di errore:

'ConsoleApplication1.SampleClass.sampleField' è inaccessibile a causa del livello di protezione

Corretta:

public class SampleClass

{

protected string sampleField;

}

class SampleClassD : SampleClass

{

static void Main()

{

SampleClassD scD = new SampleClassD();

scD.sampleField = "Valore";

Console.WriteLine("scD.sampleField " + scD.sampleField);

Console.ReadKey();

}

}

### 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";

}

Non è possibile fare riferimento a un membro static mediante un'istanza. 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.

Utilizzare il modificatore static per dichiarare un membro statico che appartiene al tipo stesso anziché a un oggetto specifico.Il modificatore static può essere utilizzato con classi, campi, metodi, proprietà, operatori, eventi e costruttori, ma non con indicizzatori, distruttori o tipi che non sono classi.

La classe seguente viene dichiarata come static e contiene soltanto metodi static:

static class CompanyEmployee

{

public static void DoSomething() { /\*...\*/ }

public static void DoSomethingElse() { /\*...\*/ }

}

**Nota:** Se la parola chiave static viene applicata a una classe, tutti i membri della classe devono essere statici.

In questo esempio si leggono il nome e l'identificatore di un nuovo dipendente; il contatore di dipendenti viene quindi incrementato di un'unità e vengono visualizzate le informazioni relative al nuovo dipendente e al numero di dipendenti aggiornato.

public class Employee4

{

public string id;

public string name;

public Employee4()

{

}

public Employee4(string name, string id)

{

this.name = name;

this.id = id;

}

public static int employeeCounter;

public static int AddEmployee()

{ return ++employeeCounter; }

}

class MainClass : Employee4

{

static void Main()

{

Console.Write("Enter the employee's name: ");

string name = Console.ReadLine();

Console.Write("Enter the employee's ID: ");

string id = Console.ReadLine();

// Create and configure the employee object:

Employee4 e = new Employee4(name, id);

Console.Write("Enter the current number of employees: ");

string n = Console.ReadLine();

Employee4.employeeCounter = Int32.Parse(n);

Employee4.AddEmployee();

// Display the new information:

Console.WriteLine("Name: {0}", e.name);

Console.WriteLine("ID: {0}", e.id);

Console.WriteLine("New Number of Employees: {0}",

Employee4.employeeCounter);

}

}

Input:

Matthias Berndt

AF643G

15

\*

Sample Output:

Enter the employee's name: Matthias Berndt

Enter the employee's ID: AF643G

Enter the current number of employees: 15

Name: Matthias Berndt

ID: AF643G

New Number of Employees: 16

In questo esempio viene mostrato che, sebbene sia possibile inizializzare un campo statico tramite un altro campo statico non ancora dichiarato, i risultati non saranno definiti finché non verrà assegnato un valore al campo statico in modo esplicito.

class Test

{

static int x = y;

static int y = 5;

static void Main()

{

Console.WriteLine(Test.x);

Console.WriteLine(Test.y);

Test.x = 99;

Console.WriteLine(Test.x);

}

}

/\*

Output:

0

5

99

\*/

[Ereditarietà](javascript:void(0))

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.

|  |
| --- |
| **NotaNota:** I linguaggi gestiti in .NET Framework non supportano l'ereditarietà multipla, ad esempio 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.

[Esempio](javascript:void(0))

In questo esempio, la classe Square deve fornire un'implementazione sottoposta a override dell'oggetto Area poiché quest'ultimo viene ereditato dall'oggetto ShapesClass astratto.

abstract class ShapesClass

{

abstract public int Area();

}

class Square : ShapesClass

{

int side = 0;

public Square(int n)

{

side = n;

}

// Area method is required to avoid

// a compile-time error.

public override int Area()

{

return side \* side;

}

static void Main()

{

Square sq = new Square(12);

Console.WriteLine("Area of the square = {0}", sq.Area());

}

interface I

{

void M();

}

abstract class C : I

{

public abstract void M();

}

}

// Output: Area of the square = 144

Un metodo override fornisce una nuova implementazione di un membro ereditato da una classe base. Il metodo che viene sottoposto a override mediante una dichiarazione override viene definito metodo di base sottoposto a override. Il metodo di base sottoposto a override deve avere la stessa firma del metodo di override. Non è possibile eseguire l'override di un metodo non virtuale o static. Il metodo di base sottoposto a override deve essere virtual, abstract o override.

Le dichiarazioni override non possono modificare l'accessibilità del metodo virtual.

Una dichiarazione di proprietà di override deve indicare esattamente lo stesso modificatore di accesso, lo stesso tipo e lo stesso nome della proprietà ereditata. Inoltre, la proprietà sottoposta a override deve essere virtual, abstract o override.

Nell'esempio seguente vengono definite una classe base denominata Employee e una classe derivata denominata SalesEmployee. La classe SalesEmployee comprende una proprietà aggiuntiva, salesbonus, ed esegue l'override del metodo CalculatePay per prenderla in considerazione.

class TestOverride

{

public class Employee

{

public string name;

// Basepay is defined as protected, so that it may be

// accessed only by this class and derrived classes.

protected decimal basepay;

// Constructor to set the name and basepay values.

public Employee(string name, decimal basepay)

{

this.name = name;

this.basepay = basepay;

}

// Declared virtual so it can be overridden.

public virtual decimal CalculatePay()

{ return basepay; }

}

// Derive a new class from Employee.

public class SalesEmployee : Employee

{

// New field that will affect the base pay.

private decimal salesbonus;

// The constructor calls the base-class version, and

// initializes the salesbonus field.

public SalesEmployee(string name, decimal basepay,

decimal salesbonus) : base(name, basepay)

{

this.salesbonus = salesbonus;

}

// Override the CalculatePay method

// to take bonus into account.

public override decimal CalculatePay()

{

return basepay + salesbonus;

}

}

static void Main()

{

// Create some new employees.

SalesEmployee employee1 = new SalesEmployee("Alice",

1000, 500);

Employee employee2 = new Employee("Bob", 1200);

Console.WriteLine("Employee4 " + employee1.name +

" earned: " + employee1.CalculatePay());

Console.WriteLine("Employee4 " + employee2.name +

" earned: " + employee2.CalculatePay());

}

}

Output:

Employee4 Alice earned: 1500

Employee4 Bob earned: 1200

**Esempio**

Nell'esempio riportato di seguito viene illustrata la dichiarazione di costruttori, metodi e campi di classi.Viene inoltre descritta la creazione di istanze di oggetti e la stampa di dati di un'istanza.In questo esempio vengono dichiarate due classi, la classe Child, che contiene due campi privati (name e age) e due metodi pubblici.La seconda classe, StringTest, viene utilizzata per contenere Main.

class Child

{

private int age;

private string name;

// Default constructor:

public Child()

{

name = "N/A";

}

// Constructor:

public Child(string name, int age)

{

this.name = name;

this.age = age;

}

// Printing method:

public void PrintChild()

{

Console.WriteLine("{0}, {1} years old.", name, age);

}

}

class StringTest

{

static void Main()

{

// Create objects by using the new operator:

Child child1 = new Child("Craig", 11);

Child child2 = new Child("Sally", 10);

// Create an object using the default constructor:

Child child3 = new Child();

// Display results:

Console.Write("Child #1: ");

child1.PrintChild();

Console.Write("Child #2: ");

child2.PrintChild();

Console.Write("Child #3: ");

child3.PrintChild();

}

}

/\* Output:

Child #1: Craig, 11 years old.

Child #2: Sally, 10 years old.

Child #3: N/A, 0 years old.

\*/

**Commenti**

Si noti, nell'esempio precedente, che è possibile accedere ai campi privati (name e age) solo tramite i metodi pubblici della classe Child.Non è quindi possibile stampare il nome del bambino dal metodo Main utilizzando un'istruzione come quella riportata di seguito.

Console.Write(child1.name); // Error

L'accesso ai membri privati di Child da Main è possibile solo se Main è un membro della classe.

Tipi una classe interna dichiarata senza un'impostazione predefinita al modificatore di accesso a private, i membri dati in questo esempio rimane sempre private se la parola chiave è stata rimossa.

Si noti infine che per l'oggetto creato mediante il costruttore predefinito (child3), il campo dell'età è stato inizializzato su zero per impostazione predefinita.

**Esempio (Ereditarietà)**

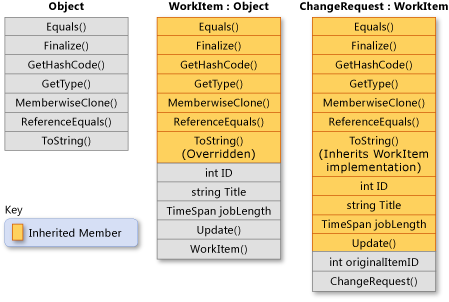
L’ereditarietà, insieme con l’incapsulamento e il polimorfismo, rappresenta una delle tre principali caratteristiche (o "pilastri") della programmazione orientata a oggetti. L'ereditarietà permette di creare nuove classi che riutilizzano, estendono e modificano il comportamento definito in altre classi. La classe i cui membri vengono ereditati è denominata classe base, mentre la classe che eredita tali membri è denominata classe derivata.

Concettualmente, una classe derivata rappresenta una specializzazione della classe base. Ad esempio, avendo una classe base Animal, è possibile definire una classe derivata denominata Mammal e un’altra classe derivata denominata Reptile. Un oggetto Mammal è anche un oggettoAnimal e un oggetto Reptile è anche un Animal, ma ciascuna classe derivata rappresenta una diversa specializzazione della 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.

Nella figura riportata di seguito viene mostrata una classe WorkItem che rappresenta un elemento di lavoro in un qualche processo aziendale. Come per tutte le classi, è derivata da [System.Object](http://msdn.microsoft.com/it-it/library/system.object%28v=vs.90%29.aspx) ed eredita tutti i metodi di tale classe. La classe WorkItem aggiunge cinque membri propri. Fra questi è incluso un costruttore, in quanto i costruttori non vengono ereditati. La classe ChangeRequest eredita dalla classe WorkItem e rappresenta un particolare tipo di elemento di lavoro. La classe ChangeRequest aggiunge due ulteriori membri all’elenco di membri ereditati da WorkItem e da [Object](http://msdn.microsoft.com/it-it/library/system.object%28v=vs.90%29.aspx). La classe deve aggiungere un proprio costruttore e aggiunge inoltre un membro che consentirà l’associazione tra l’oggetto ChangeRequest e l’oggetto WorkItem originale al quale si applica la modifica.

Ereditarietà delle classi



Nell'esempio seguente viene illustrato come le relazioni tra le classi mostrate nella precedente illustrazione vengono espresse in C#. Nell'esempio viene inoltre descritto come la classe WorkItem implementa l’override del metodo virtuale [Object.ToString](http://msdn.microsoft.com/it-it/library/system.object.tostring%28v=vs.90%29.aspx) e come la classe ChangeRequest eredita l’implementazione del metodo definito dalla classe WorkItem.

// WorkItem implicitly inherits from Object class

public class WorkItem

{

private static int nextID;

protected int ID { get; set; }

protected TimeSpan jobLength { get; set; }

protected string Title { get; set; }

protected string Description { get; set; }

// Default constructor

public WorkItem()

{

ID = 0;

Title = "Default title";

Description = "Default description.";

jobLength = new TimeSpan();

}

// Static constructor for static member.

static WorkItem()

{

nextID = 0;

}

// Instance constructor.

public WorkItem( string title, string desc, TimeSpan joblen)

{

this.ID = GetNextID();

this.Title = title;

this.Description = desc;

this.jobLength = joblen;

}

protected int GetNextID()

{

return ++nextID;

}

public void Update(string title, TimeSpan joblen)

{

this.Title = title;

this.jobLength = joblen;

}

// Virtual method override.

public override string ToString()

{

return String.Format("{0} - {1}", this.ID, this.Title);

}

}

// ChangeRequest derives from WorkItem and adds two of its own members.

public class ChangeRequest : WorkItem

{

protected int originalItemID {get; set;}

public ChangeRequest() { }

public ChangeRequest(string title, string desc, TimeSpan jobLen, int originalID)

{

this.ID = GetNextID();

this.Title = title;

this.Description = desc;

this.jobLength = jobLen;

this.originalItemID = originalID;

}

}

class Program

{

static void Main()

{

WorkItem item = new WorkItem(

"Fix Bugs",

"Fix all bugs in my source code branch",

new TimeSpan(3, 4, 0, 0));

ChangeRequest change = new ChangeRequest("Change design of base class",

"Add members to base class",

new TimeSpan(4, 0, 0),

1);

Console.WriteLine(item.ToString());

// ChangeRequest inherits WorkItem's override of ToString

Console.WriteLine(change.ToString());

// Keep the console open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

/\* Output:

1 - Fix Bugs

2 - Change design of base class

\*/