Principi della programmazione orientata agli oggetti

I principi della programmazione ad oggetti

Autore: Prof. Rio Chierego

Marzo 2016

Roadmap

- **Introduzione alle tecniche di programmazione**
- **Principi base del paradigma OO**
  - Astrazione
  - Classi ed Oggetti
  - Incapsulamento e Information Hiding
  - Ereditarietà
  - Polimorfismo
  - Modularità
- **Il linguaggio C++ e l'OOP**

Principi OOP: introduzione

Tecniche di programmazione a confronto:

a) Programmazione Non Strutturata

b) Programmazione Procedurale

c) Programmazione Modulare

d) Programmazione ad Oggetti

Introduzione: tecniche di programmazione a confronto

a) Programmazione Non Strutturata

1) Il programma è costituito da **un unico blocco di codice detto "main"** dentro il quale vengono manipolati i dati in maniera totalmente sequenziale

2) Tutti i dati sono rappresentati soltanto da **variabili e/o costanti e/o tipi** contenuti in un **ambiente globale** di visibilità

Svantaggi

a) È 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

b) Al crescere della complessità dei programmi da sviluppare, rend più complesse e problematiche le operazioni di **modifica** e di **testing** del codice prodotto

Introduzione: tecniche di programmazione a confronto

a) Programmazione Non Strutturata

Figura 1. Schema della programmazione non strutturata

Introduzione: tecniche di programmazione a confronto

b) Programmazione Procedurale

1) **Nascono le Procedure** ossia si raggruppano i pezzi di programma ripetuti in porzioni di codice utilizzabili e richiamabili ogni volta che se ne presenti l'esigenza

2) Una **Procedura** è un sottoprogramma che svolge una ben determinata **funzionalità** (ad esempio, il calcolo della radice quadrata di un numero) e che è visibile e richiamabile dal resto del codice

3) Ogni **Procedura** ha la capacità di poter utilizzare uno o più **parametri** che ne rappresentano l'interfaccia ed è **attivata** attraverso l'apposita **invocazione** nel **main**

4) Il **main** continua ad esistere ma al suo interno appaiono soltanto le invocazioni alle procedure definite nel programma e quando una procedura ha terminato il suo compito il controllo ritorna nuovamente al main

Introduzione: tecniche di programmazione a confronto

b) Programmazione Procedurale

Vantaggi

1) Al crescere della complessità dei programmi da sviluppare, consente di rendere più semplici ed efficaci le operazioni di **modifica** e di **testing** del codice prodotto

2) Permette la **suddivisione del lavoro tra vari programmatori**

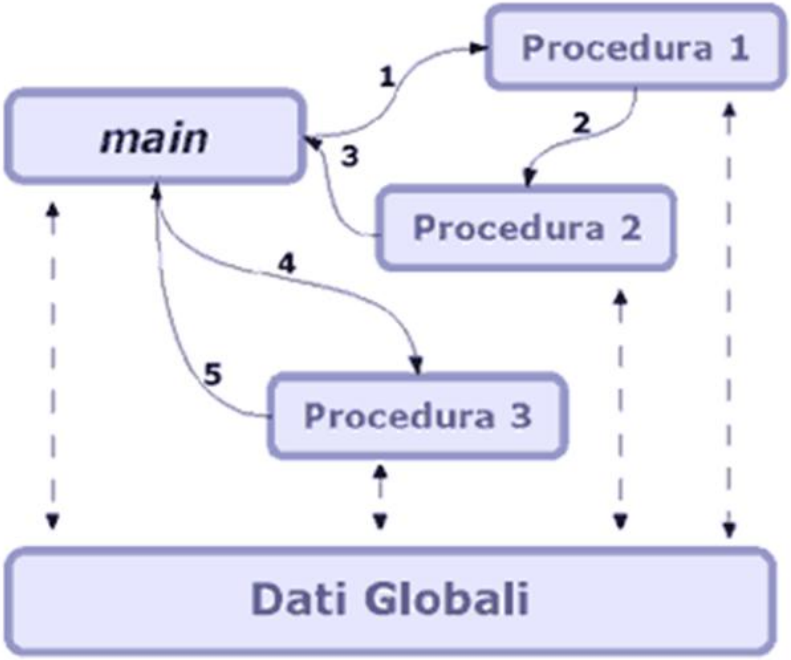
3) Semplificazione del **riuso**

4) Semplificazione della **manutenzione**

Introduzione: tecniche di programmazione a confronto

b) Programmazione Procedurale

Figura 2. Flusso di un programma con programmazione procedurale



Introduzione: tecniche di programmazione a confronto

c) Programmazione Modulare

1) E' possibile poter **riutilizzare le Procedure** messe a disposizione da un programma in modo che anche altri programmi ne possono trarre vantaggio

2) E' possibile raggruppare le **Procedure aventi un dominio comune** (ad esempio, procedure che eseguissero operazioni matematiche) in moduli separati (nascono le **librerie di sistema e/o utente**)

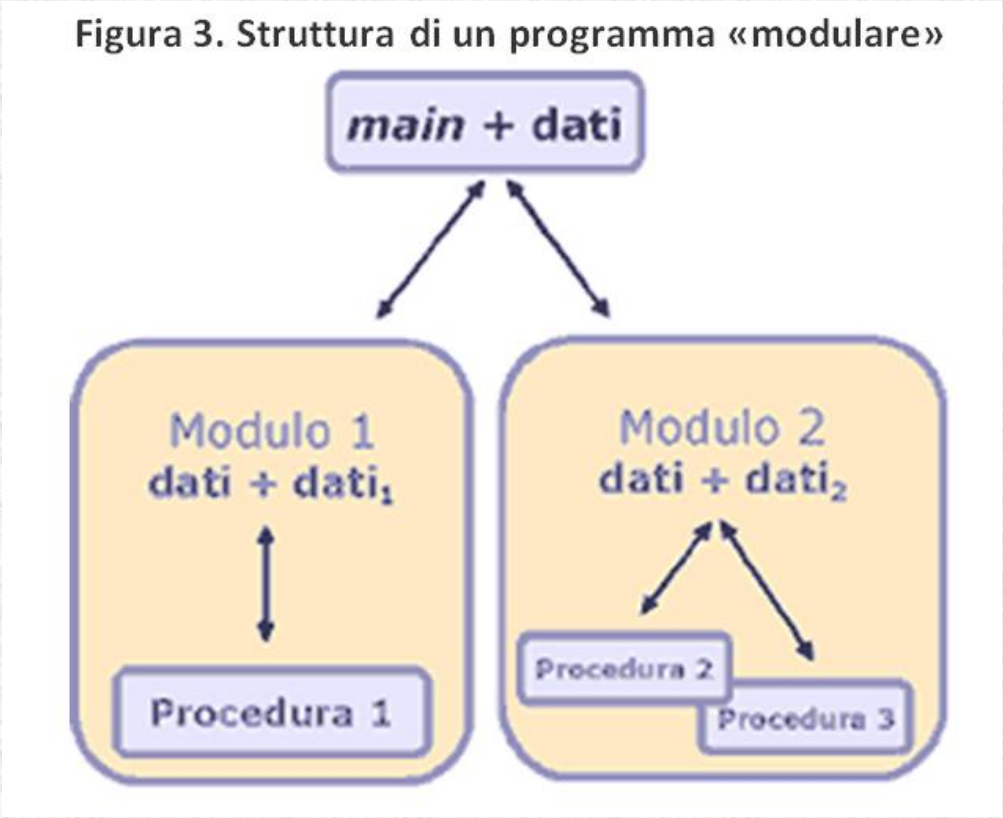
3) Un singolo programma non è più costituito da un solo file (in cui è presente il **main** e tutte le **Procedure**) ma da diversi **moduli** (uno per il main e tanti altri quanti sono i moduli a cui il programma fa riferimento).

4) I singoli **moduli** possono contenere anche dei dati propri (**variabili e/o costanti contenuti in un ambiente locale** di visibilità) che, in congiunzione ai dati del **main**, vengono utilizzati all'interno delle procedure in essi contenute

Introduzione: tecniche di programmazione a confronto

c) Programmazione Modulare

Figura 3. Struttura di un programma «modulare»



Introduzione: tecniche di programmazione a confronto

d) Programmazione ad Oggetti (Object-Oriented programming)

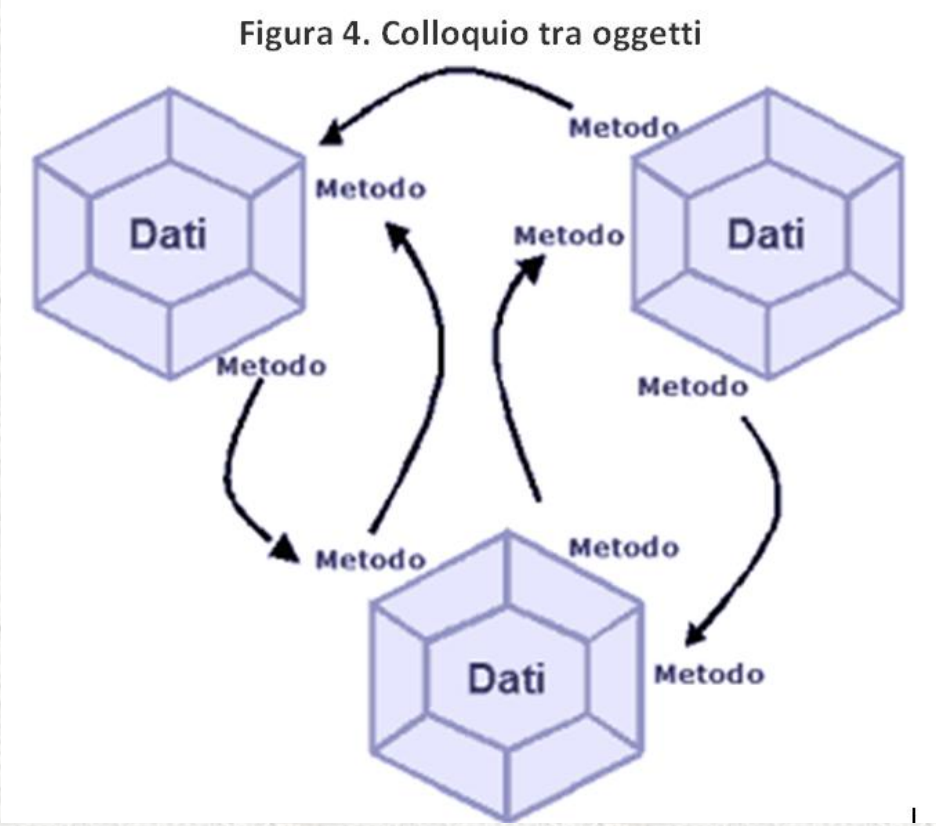
1) Secondo il paradigma OOP è possibile pensare ad un **programma** come un insieme di **oggetti** appartenenti **alla stessa classe o a classi diverse** che interagiscono tra loro scambiandosi **messaggi** ma mantenendo ognuno il proprio **stato** ed i propri **dati**

2) Nei linguaggi di programmazione OOP esiste un **nuovo tipo di dato** (astratto) la **classe** che serve a modellare **un insieme di oggetti** con le stesse **caratteristiche** in grado di compiere le stesse **azioni**

3) La **programmazione ad oggetti**, naturalmente, pur cambiando radicalmente l'approccio mentale all'analisi progettuale non ha fatto a meno dei vantaggi derivanti dall'uso dei **moduli** ulteriormente affinandola

Introduzione: tecniche di programmazione a confronto

d) Programmazione ad Oggetti



Principi OOP: concetti basilari

La programmazione ad oggetti si basa su alcuni concetti basilari

a) Astrazione

b) Incapsulamento delle informazioni ed Information hiding

c) Ereditarietà

d) Polimorfismo

e) Modularità

Principi OOP : a) Astrazione (le classi)

L'**idea principale** che sta dietro la Programmazione ad Oggetti risiede, in buona parte, nella possibilità di **modellare** la **realtà di interesse** in un **sistema software** in modo più naturale e vicino all'uomo attraverso **l'astrazione**

L'**astrazione** è un procedimento mentale che permette di evidenziare alcune proprietà, ritenute significative, relative ad un determinato fenomeno osservato escludendone altre considerate non rilevanti per la sua comprensione

"Qualsiasi modello include gli aspetti più importanti o essenziali di qualcosa mentre ignora i dettagli meno importanti, immateriali. Il risultato è di rimuovere le differenze ed enfatizzare gli aspetti comuni" [Dizionario di Object Technology – Firesmith, Eykholt 1995]

L'uso dell'**astrazione** comporta evidentemente la creazione di **modelli**
(nel caso dell'OOP teli modelli sono rappresentati dalle **classi)**

Principi OOP : a) Astrazione (le classi e gli oggetti)

Una **classe** è un modello astratto generico per una famiglia di oggetti con caratteristiche e funzionalità comuni.

Seguendo il principio OO dell'astrazione:

– Enfatizza le caratteristiche rilevanti

– Sopprime le altre caratteristiche non influenti

Una **oggetto** (o istanza) è una rappresentazione concreta e specifica di una **classe**

Per **processo di istanziazione** si intende la possibilità che una classe permetta la generazione di oggetti (o istanze) in modo che ciascuno possa contenere informazioni specifiche che lo differenziano dagli altri.

Principi OOP : a) le classi e gli oggetti

1) Esempio di ASTRAZIONE la **classe Automobile**

Un automobile in generale può essere caratterizzata in questo modo:

Proprietà (o Attributi)	**Funzionalità (o Metodi)**
Marca	Accelera
Modello	Frena
Potenza	Cambia marcia
Marce	Cambia direzione
Peso	
Cilindrata	

E' possibile interagire con un'automobile per determinarne il suo **comportamento** (**stato**) attraverso la sua **interfaccia** che permette di effettuare le sole operazioni consentite (**il pedale del freno,il pedale dell'acceleratore, il volante, la leva del cambio)**

Principi OOP : a) le classi e gli oggetti

Esempio di **oggetti (istanze)** della **classe Automobile**

Hyunday
I10 Sound Edition
67 CV
5 marce
550 kg
1000 cm3

Opel
Mokka
136 CV
6 marce
850 kg
1600 cm3

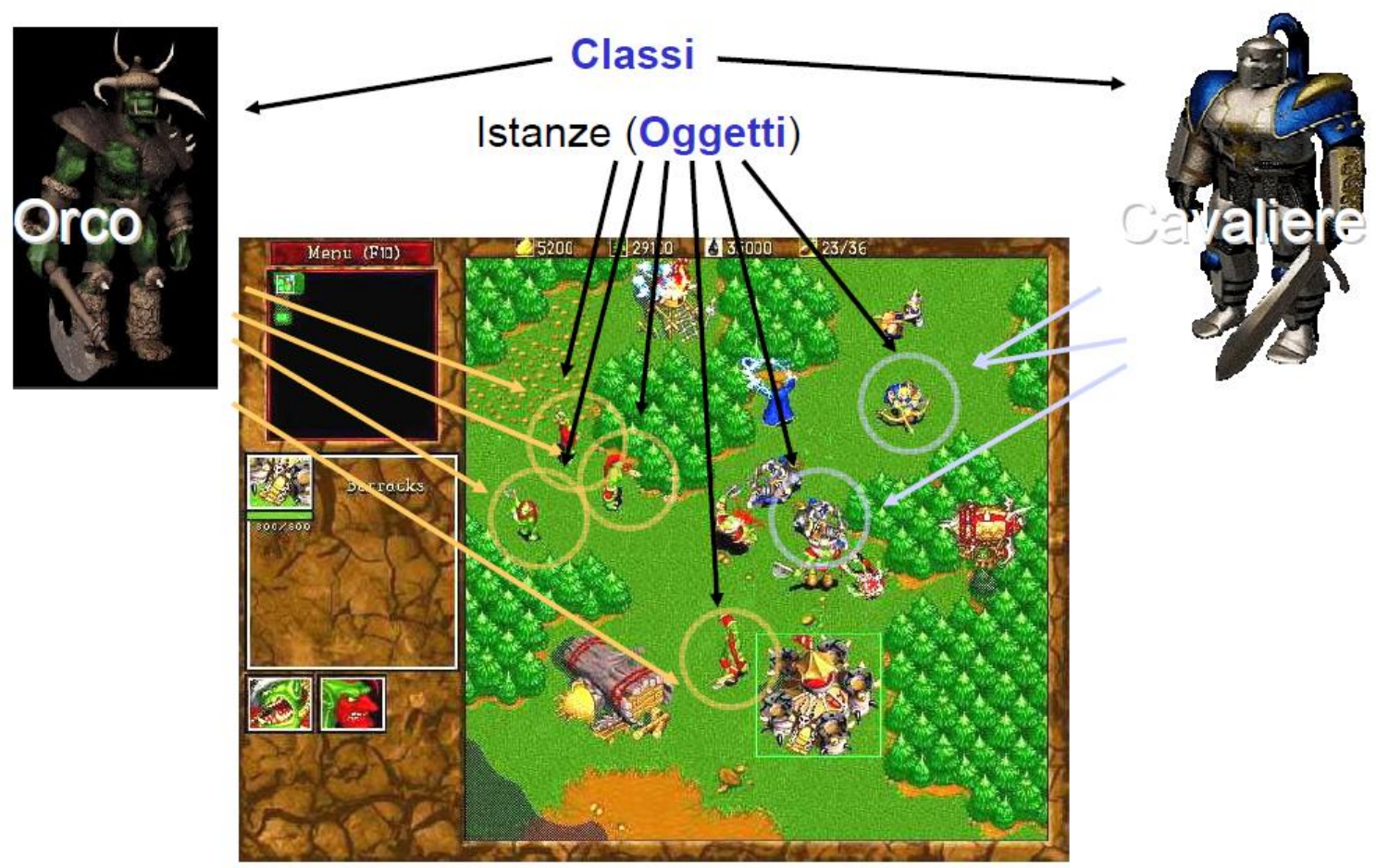
Ferrari
GTC 4
507 CV
7 marce
1790 kg
6262 cm3

Principi OOP : a) le classi e gli oggetti

2) Esempio di utilizzo di **classi** e di **oggetti:** un videogioco

Principi OOP : a) le classi e gli oggetti

2) Esempio di utilizzo di **classi** e di **oggetti:** un videogioco



Principi OOP : a) le classi e gli oggetti

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 di dato astratto** per un determinato oggetto ad essa appartenente

I **metodi (**o **funzioni)** costituiscono le **azioni** che possono essere compiute da un oggetto appartenente a quella classe

Gli **attributi** (o **dati**) rappresentano le **proprietà** caratteristiche di un **oggetto** di una determinata classe, ovvero le informazioni su cui i **metodi** possono eseguire le loro elaborazioni.

Un particolare oggetto che appartiene ad una classe costituisce un'**istanza della classe** stessa

Principi OOP : a) le classi e gli oggetti

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) o interfaccia

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

Un oggetto rappresenta un'entità a sé stante, ben definita che, nel corso dell'elaborazione, è soggetta ad una **creazione**, ad un suo utilizzo e, infine, alla sua **distruzione**

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.

Principi OOP : a) le classi e gli oggetti

I messaggi

I metodi rappresentano le azioni che un oggetto è in grado di eseguire che vengono scatenate ed eseguite rispondendo alle **"sollecitazioni" provenienti da altri oggetti**

Tali sollecitazioni costituiscono quelli che, in un programma che utilizza il paradigma OOP, vengono definiti **messaggi**

Si è soliti suddividere i **messaggi** nelle seguenti categorie:

Costruttori

I Costruttori costituiscono il momento in cui viene creato un oggetto allocandolo in memoria.

Essi devono essere richiamati ogni volta che si vuole creare una nuova istanza di un oggetto appartenente ad una classe e, solitamente, svolgono al loro interno funzioni di inizializzazione.

Distruttori

I Distruttori svolgono la funzione inversa dei costruttori: distruggono un oggetto ovvero ne eliminano la allocazione dalla memoria

Accessori (Accessors)

I messaggi di tipo "Accessors" vengono utilizzati per esaminare il contenuto di una proprietà di una classe (metodi Getter e Setter). Solitamente si utilizzano questi metodi per accedere alle variabili dichiarate con visibilità Private (vedremo il significato di questo termine in seguito)

Modificatori (Mutators)

I Modificatori rappresentano tutti i messaggi che provocano una modifica nello stato di un oggetto

Principi OOP : il linguaggio di modeling UML

Lo sviluppo di un sistema orientato agli oggetti, avviene seguendo le seguenti fasi:

1° fase : OOA: Object Oriented Analysis

2° fase: OOD: Object Oriented Design

3° fase: OOP: Object Oriented Programming

La fase di analisi (**OOA**) consiste nel creare un primo modello concettuale del problema che a noi interessa (ad esempio utilizzando il linguaggio di modeling **UML - *Unified Modelling Language***)

Durante la fase di progettazione (**OOD**), si cerca una strategia con cui risolvere i problemi relativi al fatto che il tale modellazione dovrà trovare spazio su un calcolatore (es. allocazione fisica dei dati, l'interfacciamento tra oggetti, le prestazioni etc.). Alla fine di tale processo gli oggetti, verranno raggruppati in **package** (moduli), e si stabiliranno le interconnessioni fra moduli differenti del programma

Nell'ultima fase (**OOP**), si effettuerà la traduzione da **UML** ad un linguaggio di programmazione orientato agli oggetti, **come il C++ o Java**, seguendo le informazioni forniteci dai modelli che abbiamo prodotto durante le precedenti fasi

Principi OOP : il linguaggio di modeling UML

L'UML o *Unified Modelling Language* è un linguaggio di modeling che ci aiuta a a rappresentare, soprattutto graficamente, **cosa** che il nostro programma dovrà fare e non il **come** esso sarà in grado di farlo.

Lo **scopo di UML** è quello di costruire delle specifiche formali per sistemi software complessi. Per fare ciò UML si avvale principalmente di uno strumento: i diagrammi.

Tra i **diagrammi standard** di UML vi sono:

Diagramma dei casi d'uso
Diagramma delle classi
Diagramma di sequenza
Diagramma di collaborazione
Diagramma di stato
Diagramma di attività
Diagramma dei componenti
Diagramma di allocazione

Principi OOP : il linguaggio di modeling UML

Nel linguaggio di modeling **UML** una **classe** viene rappresentata come segue:

Specificare più o meno accuratamente, i metodi e gli attributi delle classi in gioco, dipende dalla fase di progettazione in cui ci troviamo attualmente, sia essa di analisi (**OOA**) o di progettazione (**OOD**)

La visibilità di ogni attributo o metodo è specificata in UML attraverso l'uso di vari simboli (vedi **information hiding**)

- visibilità privata: l'attributo è accessibile solo dall'interno della classe usando i propri metodi

+ visibilità pubblica: l'attributo o il metodo è accessibile anche dall'esterno della classe

 # visibilità protetta: l'attributo o il metodo viene *ereditato* da tutte le classi da questa derivate.

Principi OOP : a) le relazioni tra le classi

Relazione tra classi

Premessa : una classe che non si interfacci con altre classi è sicuramente poco significativa nell'OOP.

Abbiamo visto che gli **oggetti**, in un **programma Object Oriented**, interagiscono tra loro utilizzando lo scambio di **messaggi** per richiedere l'esecuzione di un particolare **metodo**

Tale **comunicazione** consente di identificare all'interno del programma una serie di **relazioni tra le classi** in gioco la cui documentazione risulta essere assai utile in fase di disegno e di analisi.

Principi OOP : a) le classi e gli oggetti

Relazione tra classi ed oggetti

Le più comuni relazioni tra classi , in un programma ad Oggetti sono catalogabili in tre tipologie:

1) Associazioni (Use Relationship)

2) Aggregazioni (Containment Relationship)

3) Specializzazioni-Generalizzazioni (Inheritance Relationship)

Principi OOP : a) le classi e gli oggetti

1) **Associazioni** (Use Relationship)

Le **associazioni** sono fatti del mondo reale con proprietà omogenee ai fini della loro applicazione che mettono in corrispondenza tra loro **le istanze di due o più classi**

Un'associazione che mette in corrispondenza le istanze di due sole classi si dice **associazione binaria**, altrimenti si dice **associazione ennaria**

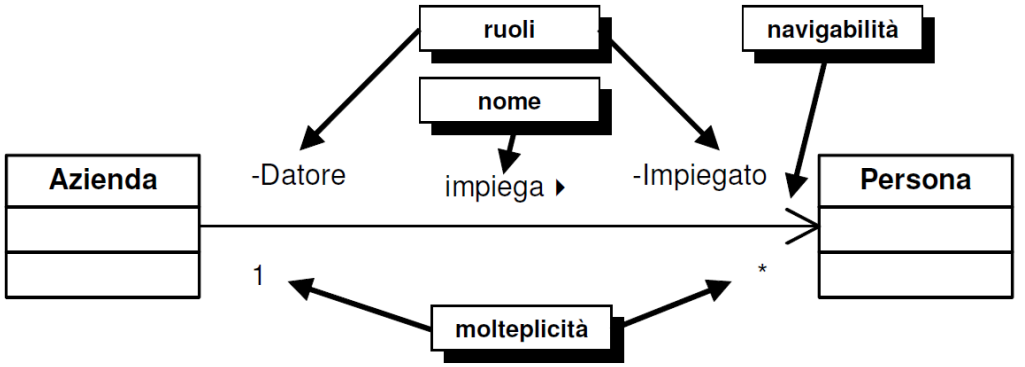
Un'associazione è caratterizzata da:

un **nome**

un **ruolo**

una **molteplicità**

una **direzione o navigabilità**



La sintassi grafica da usare è la seguente:

I numeri posti nei pressi della linea dell'associazione indicano per ciascuna istanza della prima classe il numero di istanze dell'altra che sono in corrispondenza

Il fatto che l'associazione sia realizzata con un linea senza freccia, indica che è bidirezionale, in caso contrario il verso della freccia indica il senso di lettura dell'associazione

Principi OOP : a) le classi e gli oggetti

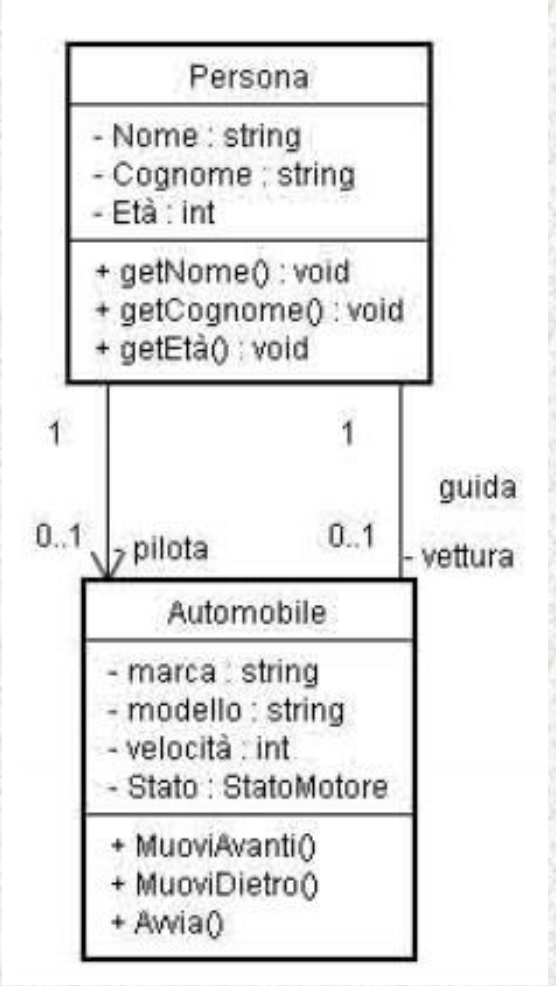
1) **Associazioni** (Use Relationship): esempio UML

La molteplicità può essere espressa:

- **Con un solo simbolo (0 1 *)**
- **Con un intervallo (0..2 1..*)**
- **Con una lista di simboli o intervalli separati da virgole (0..3,5,6..9)**

Esempi:

- **0..1** **Zero o uno**
- **1** **Esattamente uno**
- **0..*** **Zero o più**
- ***** **Zero o più (attenzione!)**
- **1..*** **Uno o più**
- **1..6** **Da uno a 6**
- **1..3,7,19..*** **Da 1 a 3, oppure 7 oppure da 19 in su**



OOP Programming: a) le classi e gli oggetti

2) Aggregazioni (Containment Relationship)

Una classe ne aggrega un'altra se esiste tra le due classi una relazione di tipo "intero-parte"

L'aggregazione consente di poter formare *aggregati* di oggetti di classi diverse, così come avviene nel mondo reale, ovvero se vediamo un oggetto di una determinata classe composto da più parti le rappresenteremo in termini di queste ultime e non dell'intero oggetto, in quanto è meglio gestire piccole classi che una sola grande classe.

Esistono due tipi di aggregazione

* una **lasca** (detta anche **aggregazione)**
* una **stretta** (detta anche **composizione)**

L'aggregazione lasca indica che l'oggetto "contenuto" ha vita propria anche senza l'oggetto "contenitore" (rombo vuoto rivolto verso la classe contenitore)

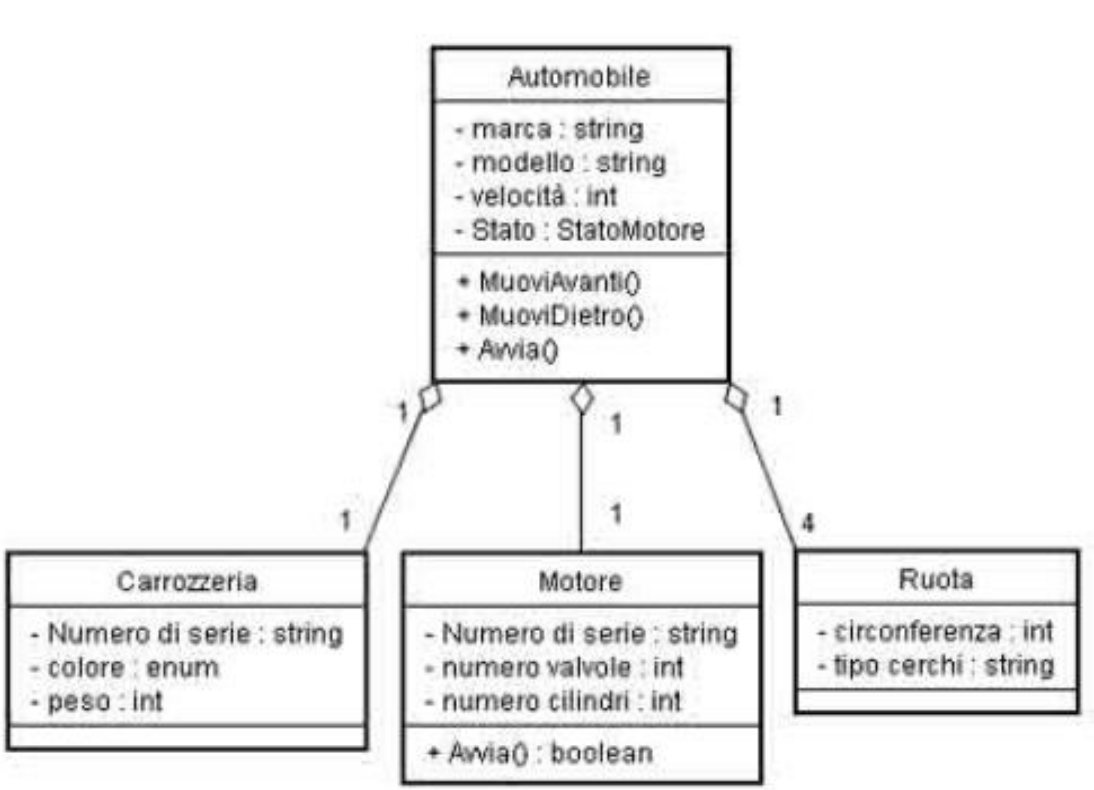
L'aggregazione stretta invece indica che l'oggetto "contenuto" non ha vita propria, quindi deve essere distrutto assieme al "contenitore" (rombo pieno rivolto verso la classe contenitore)

N.B. Anche in questo caso è possibile la specifica delle molteplicità

Principi OOP : a) le classi e gli oggetti

2) Aggregazioni (Containment Relationship): esempio **UML Aggregazione lasca**

Esempio di una **Automobile** (classe) la quale è facilmente decomponibile in più parti, tali parti saranno tutte "contenute" nella classe "contenitore" Automobile:

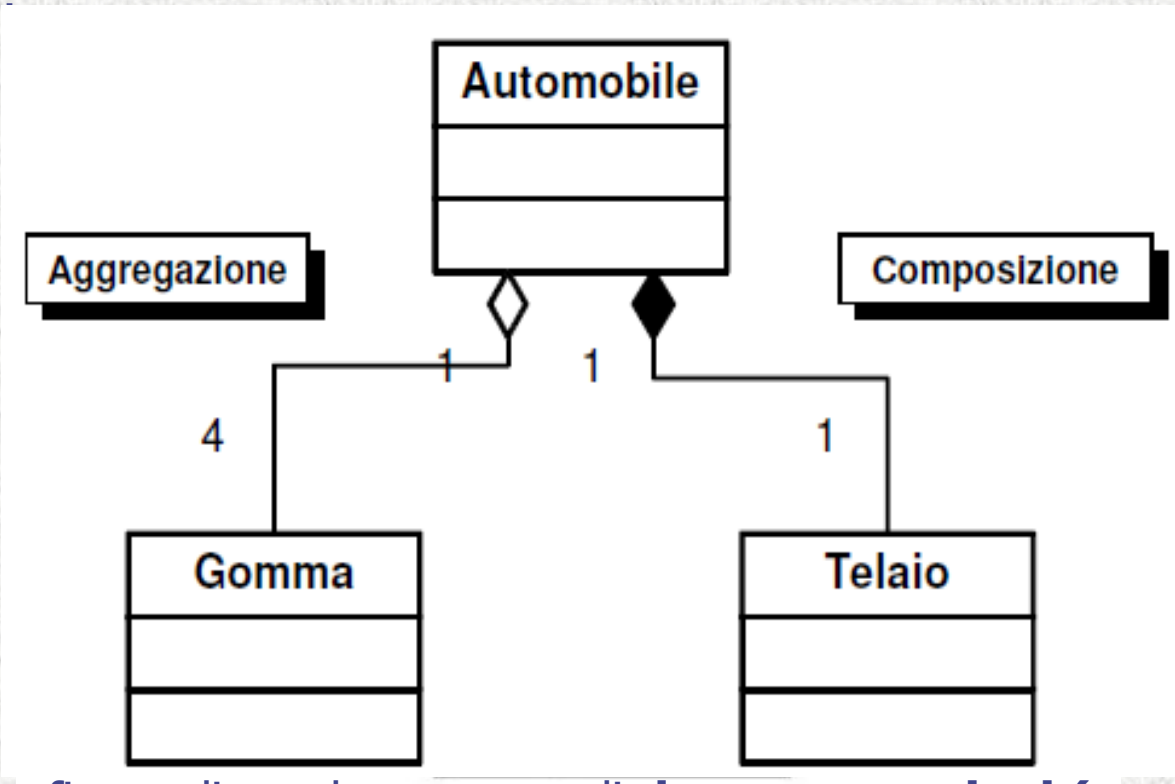


In questo grafico vediamo la presenza di **tre aggregazioni lasche**, in quanto la ruota, il motore e la carrozzeria hanno vita propria anche se esse non sono unite a formare un'automobile.

Principi OOP : a) le classi e gli oggetti

2) Aggregazioni (Containment Relationship): esempio **UML Aggregazione stretta**

Esempio di una **Automobile** (classe) la quale è facilmente decomponibile in più parti, tali parti saranno tutte "contenute" nella classe "contenitore" Automobile:



In questo grafico vediamo la presenza di **due aggregazioni (una lasca ed una stretta**, in quanto la ruota, il motore e la carrozzeria hanno vita propria anche se esse non sono unite a formare un'automobile.

Principi OOP : a) le relazioni tra le classi

3) Generalizzazioni-Specializzazioni- (Inheritance Relationship)

Le relazioni di tipo generalizzazione-specializzazione (gen-spec), servono per poter garantire al progettista un buon livello di astrazione

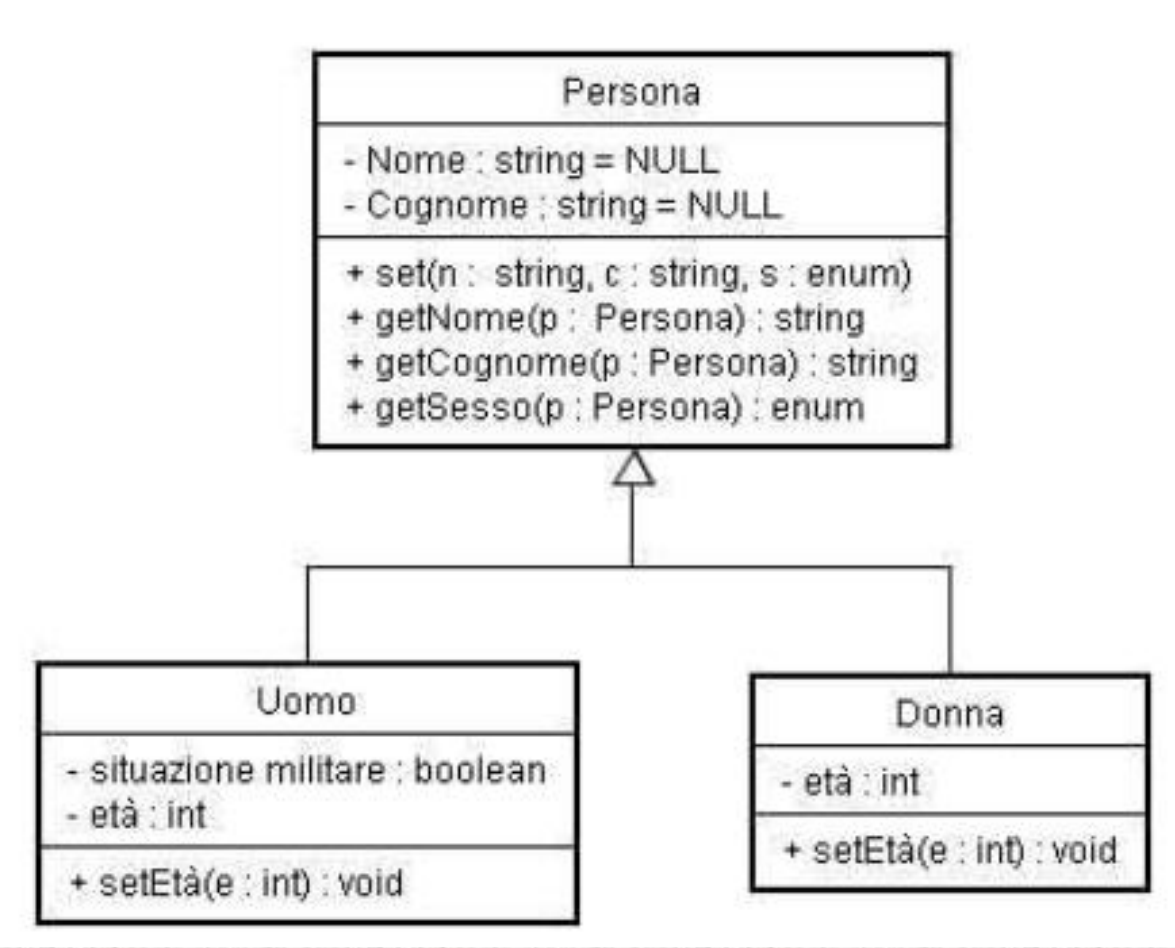
Utilizzando tali relazioni si può infatti creare in precedenza una **classe padre** (detta anche **super classe o classe base o classe di generalizzazione)** da cui far derivare in seguito **classi figlie** (dette **classi derivate o classi di specializzazione**)

Evitando di specializzare da subito tutti gli attributi di una classe, si ha il vantaggio di poter inquadrare sin dall'inizio il dominio di applicazione

La relazione di tipo Specializzazione si basa sul concetto di **ereditarietà** che verrà affrontato in seguito grazie al quale, **le classi figlie erediteranno dal padre attributi e metodi**

Principi OOP : a) le relazioni tra le classi

3) Generalizzazioni-Specializzazioni (Inheritance Relationship) esempio **UML**



Principi OOP : b) incapsulamento delle informazioni

Abbiamo visto che due oggetti dialogano tra loro tramite uno **scambio di messaggi** ed abbiamo supposto finora che tutti i metodi e gli attributi di un oggetto possano essere visibili agli altri oggetti

In realtà ogni oggetto **mittente non è obbligato a conoscere tutti i metodi dell'oggetto destinatario** poiché sono sufficienti solo quelli che il destinatario ritiene opportuno

Con **interfaccia con l'esterno** o semplicemente **interfaccia** indichiamo la lista di **proprietà** e **metodi** (per i metodi solo la **segnatura** ossia il *nome*, l'eventuale *tipo* del valore di ritorno e la *numero* e *tipo* dei suoi *parametri*) che l'oggetto rende noti all'esterno e che sono utilizzabili per interagire con esso

In questo modo **l'oggetto chiamante** conosce solo il modo di interagire con **l'oggetto chiamato** anche **se ignora completamente** *i dettagli implementativi* di quest'ultimo

Principi OOP : b) incapsulamento delle informazioni

L'information hiding è il principio teorico su cui si basa la tecnica dell'incapsulamento

- **Information Hiding**

– I dettagli implementativi di una classe - o di un costrutto di altro tipo (oggetto, modulo, ecc) - sono nascosti all'utente (vedi clausole di visibilità **private** e **public**)

– Le scelte interne di design e gli effetti che eventuali cambiamenti di tali decisioni comportano sono nascoste ai possibili utilizzatori (l'implementazione dei metodi - sia pubblici sia privati - non è comunque visibile all'esterno

- **Incapsulamento**

– Un oggetto contiene ("incapsula") al suo interno i dati (proprietà) e i metodi (funzioni) che agiscono su di essi

N.B. L' incapsulamento è una facility del linguaggio mentre l'information hiding è un principio di design

Principi OOP : b) incapsulamento delle informazioni

Principi OOP : c) ereditarietà

Molto spesso non è indispensabile o utile creare una classe dal nulla, ma è possibile utilizzare una classe già esistente (**ereditarietà semplice**) o più di una (**ereditarietà multipla**) dalla quale partire per la sua creazione, specificando solo le differenze con quest'ultima

Nella programmazione ad oggetti è possibile fare questo utilizzando il concetto di **ereditarietà** implicito nelle relazioni del tipo Generalizzazione-Specializzazione molto presenti nel mondo reale (Esempi: Persona, Poligono, Mammifero, etc.)

L'**ereditarietà** è un meccanismo che, in fase di definizione di classe, permette di specificare solo le differenze rispetto ad una classe (o più) già esistente, detta **superclasse**.

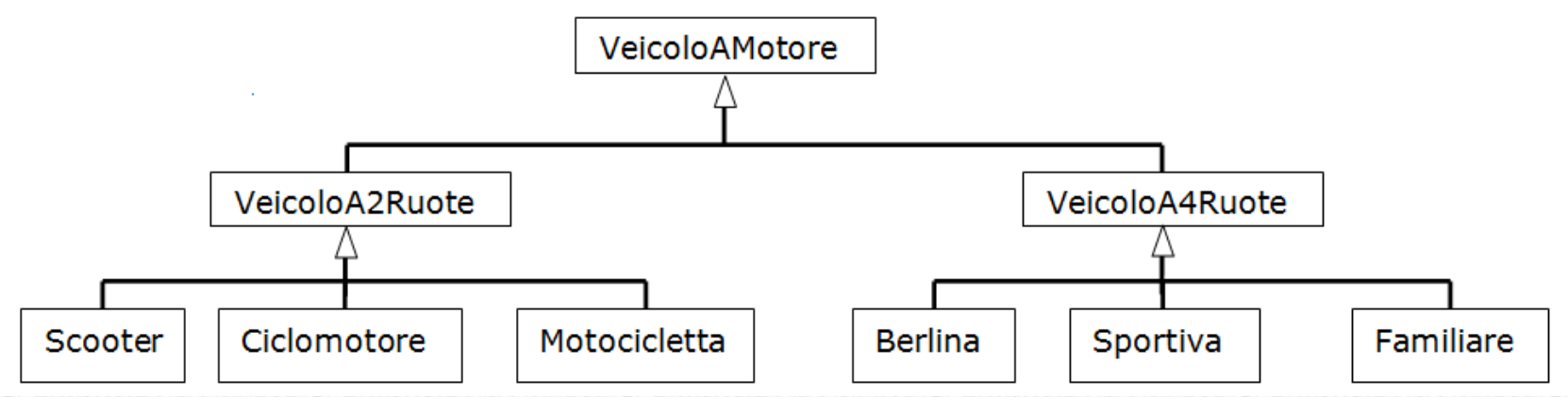
Tutte le altre caratteristiche ed i comportamenti della classe chiamata **sottoclasse** che si sta definendo **saranno gli stessi della superclasse**

Con questo concetto di ereditarietà si introduce quello di **gerarchia delle classi:**

- Ad ogni classe possono essere associate una o più classi che la *precedono* immediatamente sopra nella gerarchia (le sue **superclassi**).
- Ad ogni classe possono essere associate una o più classi che la *seguono* immediatamente sotto nella gerarchia (le sue **sottoclassi**).

Principi OOP : c) ereditarietà

Una classe **eredita** dalle classi superiori nella gerarchia tutti i loro **metodi** e tutte le loro **proprietà** *(ovviamente non definite private)*



la classe *Motocicletta* è una **sottoclasse** della classe *VeicoloA2Ruote*
la classe *VeicoloA2Ruote* è **superclasse** della classe *Motocicletta* e **sottoclasse** della classe *VeicoloAMotore*

Se la superclasse definisce già un metodo o una proprietà richiesto dalla classe non occorre ridefinire quest'ultimo o copiarne il codice, in quanto l**a classe ottiene automaticamente i metodi e le proprietà della sua superclasse** e così via (**riuso del software**)

La **classe** diventa una *combinazione* dei *metodi* e delle *proprietà* di **tutte le superclassi** che la *precedono* nella **gerarchia delle classi** ai quali vanno aggiunti i *metodi* e le *proprietà* specifici di quella classe

Principi OOP : d) Polimorfismo

I Linguaggi procedurali (esempio Pascal, Cobolo, C, etc.) basati su idea che procedure e funzioni, e i loro operandi, hanno un unico tipo. Tali linguaggi sono detti *monomorphic* (dal greco *una forma*), cioè ogni funzione, valore e variabile può avere uno ed un solo tipo

I Linguaggi OOP (come il C++, Java, C#, etc.) sono detti *polymorphic* (dal greco *più forme*), cioè ogni funzione, valore e variabile può avere più di un tipo.

Il **polimorfismo** è la capacità espressa dai *metodi ridefiniti* di assumere forme (*implementazioni*) diverse all'interno di **una gerarchia di classi** o all'interno **di una stessa classe**

In altre parole il **polimorfismo** indica la possibilità dei metodi di possedere diverse **implementazioni**

Quando un oggetto richiama un metodo di un altro oggetto appartenente ad una certa classe, esso verrà cercato dapprima in quella stessa classe. Se viene trovato (*stessa segnatura della chiamata*) sarà eseguito, altrimenti verrà ricercato risalendo nell'albero della gerarchia di classe (*tra le sue superclassi*).

Principi OOP : d) Polimorfismo: l'OVERLOAD dei metodi

Con il termine **overload** si intende la scrittura di più metodi identificati dallo stesso nome che però hanno, in ingresso, parametri di tipo e numero diverso. Può avvenire sovraccaricando uno o più metodi della stessa classe oppure uno o più metodi ereditati da una superclasse.

Dunque l'overload NON IMPLICA L'EREDITARIETA'

Una classe può ospitare un metodo con lo **stesso nome** ma del quale vengono fornite diverse implementazioni con segnature differenti che il programmatore potrà utilizzare rispettandole all'atto della chiamata

Quando una **sottoclasse** possiede un **metodo** con lo **stesso nome** della **superclasse** ma con ***numero e/o tipo di parametri differente e/o tipo ritornto*** allora non viene più ereditato il metodo della superclasse

Si dice che la sottoclasse *espressamente* **sovraccarica** (**overloading**) il metodo della superclasse

Principi OOP : d) Polimorfismo ed ereditarietà

Con il termine **override** si intende una vera e propria riscrittura di un metodo all'interno di una classe **mantenendo intatta la sua segnatura** che abbiamo ereditato dalla superclasse
Stessa segnatura significa che in entrambi i metodi sono perfettamente uguali il *nome*, l'eventuale *tipo* del valore di ritorno e *numero* e *tipo* dei *parametri*

Dunque l'override IMPLICA NECESSARIAMENTE L'EREDITARIETA'

Quando una **sottoclasse** possiede un **metodo** con la **stessa segnatura** (ossia) della **superclasse** ma con ***codice*** (ossia implementazione) **differente** allora non viene più ereditato il metodo della superclasse

Si dice che la sottoclasse *espressamente* **ridefinisce** (**overriding**) il metodo della superclasse

Principi OOP : d) Polimorfismo ed ereditarietà

Quando una **sottoclasse** aggiunge nuovi metodi e proprietà non presenti nella superclasse si dice che la sottoclasse **estende metodi e proprietà.**

Una nuova classe può dunque differenziarsi dalla superclasse o **per ridefinizione** (overriding e/o overloading) **o per estensione.**

In sintesi una classe può:

-**ereditare** metodi e proprietà dalla superclasse (*ereditarietà*);

-**estendere** la superclasse **aggiungendo** nuovi metodi e proprietà (*estensione*);

-**ridefinire** metodi della superclasse (*overriding*);

- **ridefinire** metodi propri e/o della superclasse (*overloading*)

Principi OOP : d) Polimorfismo e binding dinamico

In informatica il **binding** (in italiano *collegamento*) è il processo tramite cui viene effettuato il collegamento fra una entità di un software ed il suo corrispettivo valore

Uno dei più importanti discriminanti è il momento in cui il binding stesso viene stabilito, cioè il **binding time**

Nella **programmazione imperativa** quando il compilatore incontra una chiamata ad un sottoprogramma (funzione o procedura) con i relativi parametri attuali, realizza un legame tra tali parametri ed il sottoprogramma che deve essere chiamato. (**binding statico o early binding**)

Quindi il legame tra parametri e sottoprogramma è già noto e perfettamente specificato **a tempo di compilazione.**

Il **compilatore** nei linguaggi imperativi ad ogni chiamata di sottoprogramma fissa quale parte del codice deve essere eseguita e su quali dati farlo.
Si realizza un **legame statico** o **binding statico** o **associazione anticipata** tra il dato (parametro attuale) ed il sottoprogramma chiamato

Principi OOP : d) Polimorfismo e binding dinamico

Nella **programmazione ad oggetti** proprio a causa del polimorfismo vi è **l'impossibilità** di stabilire **a tempo di compilazione** il legame tra la *chiamata* di un *metodo* e la sua *definizione*

Il concetto di **polimorfismo** è strettamente collegato a quello di **binding dinamico** o **associazione posticipata** o ancora **collegamento ritardato** o **late binding**

Il legame tra nome del metodo ed il suo codice sarà *posticipato* **a tempo di esecuzione (run-time)** e non potrà essere risolto *a tempo di compilazione*

In questo caso il compilatore non genera una volta per tutte, all'atto della compilazione, il codice per l'assegnazione dei valori delle variabili in funzione delle chiamate dei metodi, o il codice per calcolare quale metodo chiamare in funzione delle informazioni provenienti dall'oggetto - come nel binding statico - **ma invece genera un codice che verrà utilizzato per calcolare quale metodo richiamare di volta in volta**.

Principi OOP : e) Modularità

• Un **modulo** può essere definito come un componente di un più vasto sistema, che opera in quel sistema indipendentemente dalle operazioni di altri componenti
  – per gestire la complessità, si può suddividere qualcosa che è complesso in pezzi più piccoli e quindi più maneggevoli

• La **modularità** viene utilizzato su diversi livelli di astrazione
  – frazionamento di un programma non solo in funzioni, ma a livello superiore in oggetti

• Gli oggetti possono essere raggruppati in **package**, **componenti** e **subsystem**
  – package, componenti e subsystem possono essere usati come blocchi per altri sistemi

Principi OOP : e) Modularità

- Il paradigma O.O. modella la **modularità** con **package**, **components** e **subsystem**
- Un **package** è un **namespace** che organizza un insieme di classi ed interfacce corrispondenti
- Concettualmente un **package** è simile ad una directory nel file system. Possiamo quindi memorizzare file HTML in una directory, immagini in un'altra, e script in un'altra ancora
- Tipicamente, sistemi complessi consistono di migliaia di classi, ha quindi senso organizzare le classi e le interfacce in package