La programmazione ad oggetti in java

UNITÀ 1

AREA DIGITALE

UNITÀ 0
Migrazione dal linguaggio C a Java

LA MAPPA DELLE COMPETENZE

COMPETENZE CHIAVE DI CITTADINANZA

4. Competenza digitale

Comprende l'alfabetizzazione informatica e digitale, la comunicazione e la collaborazione, l'alfabetizzazione mediatica, la creazione di contenuti digitali (inclusa la programmazione), la sicurezza, le questioni legate alla proprietà intellettuale, la risoluzione di problemi e il pensiero critico.

COMPETENZE INTERMEDIE VALUTABILI

Asse LOGICO-MATEMATICO

3 Individuare le strategie appropriate per la soluzione di problemi.

ABILITÀ

3.1 Individua dati e variabili in un problema e formalizza la relativa strategia risolutiva attraverso algoritmi.

3.3 Traduce dal linguaggio naturale al linguaggio algebrico e viceversa.

3.4 Progetta un percorso risolutivo strutturato in tappe.

3.5 Formalizza il percorso di soluzione di un problema attraverso modelli algebrici e grafici.

Asse SCIENTIFICO-TECNOLOGICO

4 Saper scegliere gli strumenti informatici in relazione all'analisi dei dati e alla modellizzazione di specifici problemi scientifici.

ABILITÀ

4.1 Analizza un oggetto o un sistema artificiale in termini di funzioni o di architettura.

4.2 Sa spiegare il principio di funzionamento e la struttura dei principali dispositivi fisici e software.

4.4 Progetta un algoritmo, implementa un programma strutturato in un linguaggio di programmazione per la risoluzione di una classe di problemi.

COMPETENZE DISCIPLINARI

- **Definire una classe con attributi e metodi**
- **Definire i costruttori e il distruttore di una classe**
- **Classificare le classi e relazioni tra di esse**
- **Applicare i concetti di incapsulamento e information hiding**
- **Riconoscere la gerarchia delle classi**
- **Rappresentare classi e oggetti mediante diagrammi UML**

CONOSCENZE

- Conoscere gli elementi teorici del paradigma a oggetti (OOP)
- Comprendere il concetto di astrazione
- Acquisire il concetto di costruttore e distruttore
- Comprendere le differenze tra overloading e overriding
- Conoscere una metodologia di documentazione delle classi (UML)
- Conoscere il significato di classe astratta

ABILITÀ

- Usare la progettazione orientata agli oggetti per programmi complessi
- Applicare il concetto di astrazione per modellare le classi
- Individuare la specializzazione e la generalizzazione di una classe
- Applicare i concetti di ereditarietà e polimorfismo
- Definire gerarchie di classi

CONTENUTI DISCIPLINARI DELL'UNITÀ

LEZIONE 1

1 OOP: evoluzione o rivoluzione?

IN QUESTA LEZIONE IMPAREREMO...

- il concetto di programmazione di sistema
- a comprendere le cause della crisi del software
- il concetto di classe, oggetto, incapsulamento, ereditarietà e polimorfismo
- il concetto di astrazione, implementazione, interfaccia

MAPPA CONCETTUALE

didattica inclusiva

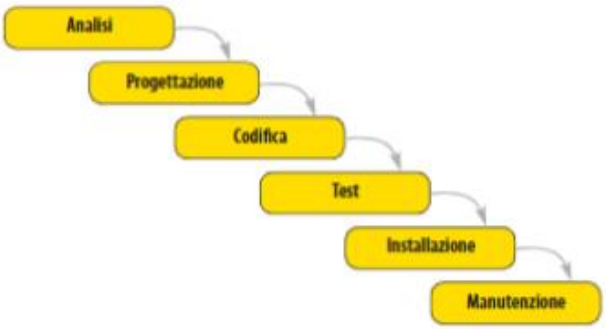
Introduzione

La programmazione a oggetti (OOP, *Object Oriented Programming*) rappresenta, senza dubbio, il modello di programmazione più diffuso e utilizzato nel nuovo millennio.

La OOP nasce alla fine degli anni Ottanta del secolo scorso come superamento della programmazione di tipo procedurale ma, oltre a essere una naturale evoluzione dei linguaggi di programmazione strutturata, è anche una vera e propria rivoluzione, in quanto modifica radicalmente il punto di vista dal quale si affronta il progetto di un'applicazione.

Come ogni rivoluzione, anche la "rivoluzione informatica" è conseguenza dell'insoddisfazione per l'inadeguatezza degli strumenti software a disposizione degli sviluppatori di fronte alla repentina evoluzione dei sistemi hardware degli anni Novanta.
Vediamo brevemente di comprenderne le cause, in modo da avvicinarci all'OOP seguendo le tappe che ne hanno consentito lo sviluppo e l'affermazione.

Le fasi di realizzazione di un programma, che vanno a formare quello che prende il nome di modello a cascata – in quanto ogni singola attività viene completata prima del passaggio alla successiva –, sono le seguenti:
- analisi del problema;
- progettazione (strategia);
- codifica (programmazione);
- test;
- installazione;
- manutenzione.



Negli anni Ottanta ci si trova di fronte a un insieme di problematiche che causano la prima vera "crisi del software": l'evoluzione esponenziale dell'hardware e la riduzione dei costi sia delle macchine sia dei sistemi operativi ha reso sproporzionati e preponderanti i costi di sviluppo e manutenzione del software applicativo, nella componente sia correttiva (per eliminare errori) sia adattativa (per rispondere a nuove esigenze).

Si passa allora dalla "programmazione di un programma" alla "programmazione di un sistema", attribuendo al concetto di base del termine "programmare" un nuovo significato.

Anche l'attività di pura codifica, intesa come scrittura di istruzioni, assume una nuova dimensione in quanto deve essere integrata al resto del progetto di sistema e quindi richiede ai progettisti nuove (e a volte maggiori) competenze, quali:
- la conoscenza di fondamenti teorici;
- la padronanza degli strumenti disponibili;
- la visione sistemistica del problema;
- la capacità di analisi dello spazio dei problemi;
- la capacità di ragionamento sul lavoro svolto;
- la capacità di comunicazione con gli utilizzatori;
- la capacità di rinnovarsi (aggiornarsi e modificare le proprie convinzioni).

Per progettare sistemi occorrono però linguaggi che offrano metafore e concetti sistemistici: infatti, in ogni linguaggio di programmazione, oltre alle regole sintattiche e semantiche e a strutture di dati, è intrinsecamente presente anche un insieme di metafore e concetti che agevolano, o meglio, "instradano", le scelte progettuali del programmatore che utilizza il linguaggio.

Ciascun linguaggio offre propri strumenti per migliorare l'espressività del programmatore (come l'organizzazione in funzioni, procedure, moduli ecc.), che lo caratterizzano o lo rendono diverso dagli altri, anche se appartenente allo stesso paradigma.

Le diversità delle metafore e dei concetti intrinseci nei diversi linguaggi sono probabilmente la vera ragione dell'esistenza di tanti linguaggi di programmazione, indipendentemente dal paradigma al quale fanno riferimento.

Le peculiarità insite in ciascun linguaggio fanno sì che, per risolvere applicazioni in un determinato ambito o settore, a volte anche per un solo specifico problema o parte di esso, alcuni di essi risultino più adatti di altri offrendo specifiche primitive o di alto alto livello che permettono di "risparmiare tempo e fatica" allo sviluppatore.

Crisi del software e OOP

AREA DIGITALE

Leggi catastrofiche

La "crisi del software" negli anni Ottanta ha rappresentato un serio problema, al punto da mettere in crisi più volte il "sistema informatica" nel suo insieme (spesso ridicolizzando i progettisti e gli operatori del settore).

Volendo ricercare le motivazioni della crisi del software, è possibile individuare tre cause fondamentali:
- crisi dimensionale;
- crisi gestionale;
- crisi qualitativa.

Esaminiamole più nel dettaglio.

La crisi dimensionale ha origini di natura essenzialmente tecnologica: è dovuta cioè alla crescita della dimensione dei sistemi informatici con la diffusione di un numero sempre maggiore di computer e della maggiore connessione degli stessi in reti locali e remote.
I nuovi sistemi non hanno solo subito un aumento delle "dimensioni fisiche", ma hanno provocato nuovi problemi e, di conseguenza, le entità delle astrazioni, dei modelli e degli strumenti per progettare si sono rivelati inadeguati.

La crisi gestionale è essenzialmente dovuta alla crescita sproporzionata dei costi per effettuare la manutenzione del software, sia correttiva sia adattativa, legata al variare delle dimensioni dei sistemi e quindi dei programmi. Nei sistemi di medie e grandi dimensioni trovare gli errori può essere molto difficile e oneroso e ogni modifica può coinvolgere tutto il team di sviluppo.

La crisi qualitativa è anch'essa legata sia alla crescita delle richieste sia al fatto che gli strumenti a disposizione dei programmatori sono risultati insufficienti per ottenere software di qualità e proprietà desiderate.
I linguaggi imperativi e la programmazione strutturata, con le loro strutture dati e le strutture di controllo, le funzioni e le procedure non si sono dimostrati strumenti efficaci. anche perché in essi non sono presenti modalità adatte a offrire supporto all'organizzazione del processo produttivo del software inteso come prodotto di qualità.

Il software di buona qualità deve invece essere protetto, modulare, riusabile, documentato, incrementabile ed estendibile: cambiando le dimensioni del problema, deve cambiare anche la dimensione del progetto, ma soprattutto la modalità con cui si deve effettuare l'approccio alla soluzione: queste nuove esigenze hanno portato alla nascita e allo sviluppo dell'ingegneria del software, con nuovi strumenti progettuali e nuovi linguaggi implementativi.

La programmazione a oggetti si propone come una nuova metodologia, con un approccio risolutivo completamente diverso: l'approccio a oggetti.

L'utilizzo di linguaggi orientati agli oggetti costringe il programmatore ad adottare nuove metodologie di programmazione, anche inconsapevolmente, sviluppando spesso in modo automatico le attitudini mentali adatte all'approccio sistemistico.

L'impiego di un linguaggio OOP impone l'utilizzo dei principi fondamentali della programmazione a oggetti, che si possono riassumere nei concetti di modularità, incapsulamento, protezione dell'informazione, cioè le caratteristiche richieste al software di buona qualità: il programmatore "automaticamente" deve utilizzare le tecniche fondamentali di organizzazione del software necessarie per superare le possibili cause di "crisi" prima descritte.

Il primo linguaggio di programmazione orientato agli oggetti fu il Simula (1967), seguito negli anni Settanta da Smalltalk proposto da Alan Kay che, di fatto, fu il primo linguaggio completamente ad oggetti (puro): a questi seguirono il linguaggio C++, Eiffel e Java.

Astrazione, oggetti e classi

Con la programmazione a oggetti è possibile modellare la realtà in modo più naturale e vicino all'uomo, offrendo nuovi strumenti con cui poter descrivere tutti i livelli di astrazione nei quali viene "trasformato" il sistema software nelle varie fasi della sua progettazione.
Di seguito, esaminiamo più nel dettaglio i principi e i concetti su cui essa si basa e i principali vantaggi offerti da questo tipo di programmazione.

Astrazione

L'astrazione è il primo importante strumento di lavoro nella fase di progettazione di un sistema software (oltre che essere un concetto importante in tutte le attività dell'ingegneria). Una sua possibile definizione è riportata di seguito.

L'astrazione è il risultato di un processo, detto appunto di astrazione, secondo il quale, assegnato un sistema, complesso quanto si voglia, si possono tenerne nascosti alcuni particolari evidenziando invece quelli che si ritengono essenziali ai fini della corretta comprensione del sistema.

Hoare, nel suo libro *Notes on Data Structuring*, sottolinea questo aspetto e associa il concetto di astrazione a quello di oggetto.

Nel processo di comprensione di fenomeni complessi, lo strumento più potente disponibile alla mente umana è l'astrazione. L'astrazione nasce dall'individuazione di proprietà simili tra certi oggetti, situazioni o processi del mondo reale e dalla decisione di concentrarsi su queste proprietà simili e di ignorare, temporaneamente, le loro differenze.
Gli oggetti simili hanno comportamenti uguali (metodi) e caratteristiche specifiche (attributi) che li differenziano tra loro: sono raccolti in classi, dove ogni classe è l'origine degli oggetti stessi e in essa vengono descritti i comportamenti e le proprietà degli oggetti.

Classe

La classe è l'elemento base della OOP e un programma a oggetti è costituito da un insieme di classi che generano oggetti che tra loro comunicano con scambi di messaggi.
Spesso le classi non hanno bisogno di essere scritte in quanto sono di dominio pubblico, cioè disponibili come moduli di libreria, e il programmatore deve solamente utilizzarle come veri e propri "mattoni software" con cui costruire il programma.

Ereditarietà

Se una classe non soddisfa appieno le esigenze specifiche di un caso particolare è possibile completarla, aggiungendole tutto quello che necessita per la nuova esigenza: si parla di classe ereditata e l'ereditarietà è la vera potenzialità della OOP, che permette di modellare e specializzare le classi già esistenti e quindi di non dover mai "partire da zero" ma sempre da solide fondamenta.

Incapsulamento

Le classi sono veri "circuiti integrati software" e il programmatore a oggetti a volte diventa un vero "assemblatore di software": naturalmente il sogno di ogni programmatore è quello di avere a disposizione tutti i componenti di cui ha bisogno ma, ovviamente, nella stragrande maggioranza dei casi questo non è possibile.
Nella OOP si realizza quello che comunemente prende il nome di *information hiding*, cioè l'incapsulamento, all'interno della classe, sia della rappresentazione dei dati sia delle elaborazioni che possono essere eseguite su di essi, fornendo all'utente un meccanismo di interfacciamento che garantisce il mascheramento del funzionamento interno dell'oggetto: solo attraverso l'interfaccia si interagisce con la classe e la classe stessa comunica all'esterno solo per "cosa fa" e non per "come lo fa".

Vantaggi della OOP

Se riprendiamo alcuni concetti fondamentali del progetto di un sistema informatico scopriamo che:
- è inutile riscrivere software che qualcuno ha già scritto (anche perché probabilmente lo ha scritto meglio di noi!);
- un programma deve essere in grado di adattarsi alla complessità del problema e del sistema senza dover intervenire radicalmente sul codice;
- un programma deve essere in grado di adattarsi alla tecnologia: per esempio, se viene modificata la rappresentazione in memoria occorre modificare solamente le operazioni specifiche che operano su quel componente e non l'intero programma.

> Dai principi fondamentali della la programmazione a oggetti ci accorgiamo che essa è la risposta a ogni esigenza.

Il grande vantaggio della OOP è che le classi disponibili per gli sviluppatori "funzionano", cioè il codice è testato e completamente esente da errori, e che sono "blindate", cioè non è possibile modificare il codice che è al loro interno ma solo utilizzarle o estenderle in nuove classi più complete con aggiunte personali.

Conclusione: che cos'è la programmazione a oggetti

Nella OOP, gli oggetti sono raccolti in classi, che definiscono campi e metodi comuni a tutti gli oggetti di un certo tipo; la OOP offre inoltre la possibilità di estendere queste classi (*nesting*) creando vere e proprie gerarchie mediante l'ereditarietà (che permette di propagare automaticamente attributi comuni a più oggetti di una stessa classe).

L'oggetto estende il concetto di dato astratto e per la definizione di una classe è necessario:
- definire tutte le operazioni che sono applicabili agli oggetti (istanze) del tipo di dato astratto;
- nascondere la rappresentazione dei dati all'utente;
- garantire che l'utente possa manipolare gli oggetti solo tramite operazioni specifiche per il tipo di dato astratto a cui gli oggetti appartengono (protezione).

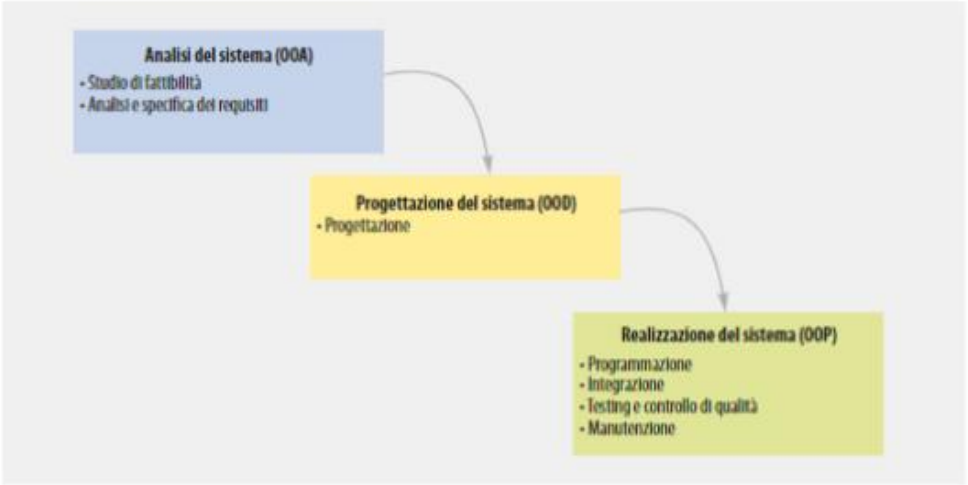
Mediante il procedimento di astrazione, gli oggetti fisici vengono virtualizzati in oggetti software che offrono le stesse informazioni e servizi degli oggetti reali: si suddivide il sistema reale preso in esame in classi di oggetti, ognuna delle quali possiede proprie variabili e funzioni, che assumono il nome di attributi e metodi.

L'incapsulamento, inoltre, obbliga il progettista a separare "che cosa c'è dentro" una classe da "che cosa si vede dal di fuori" durante tutta l'attività di realizzazione di un'applicazione:
- il sistema esterno è ciò che vede l'utente, quindi quello che deve soddisfare le aspettative definite dal contratto: al cliente non interessa che cosa c'è dentro, interessa che ciò che ha acquistato funzioni e come si utilizza, cioè come si interagisce con esso (interfaccia); il progettista deve quindi focalizzarsi sul funzionamento osservabile;
- il sistema interno non deve essere accessibile all'utente, per cui ci si deve focalizzare sull'implementazione per fare in modo che tutti i dettagli restino invisibili all'esterno del sistema stesso (incapsulamento).

Un'ulteriore caratteristica della OOP è quella che prende il nome di polimorfismo: si tratta della possibilità di applicare lo stesso operatore a classi diverse, cioè mandare un messaggio con un "medesimo significato" a classi diverse che però lo interpretano e lo eseguono in maniera differente, compatibile con la loro struttura.

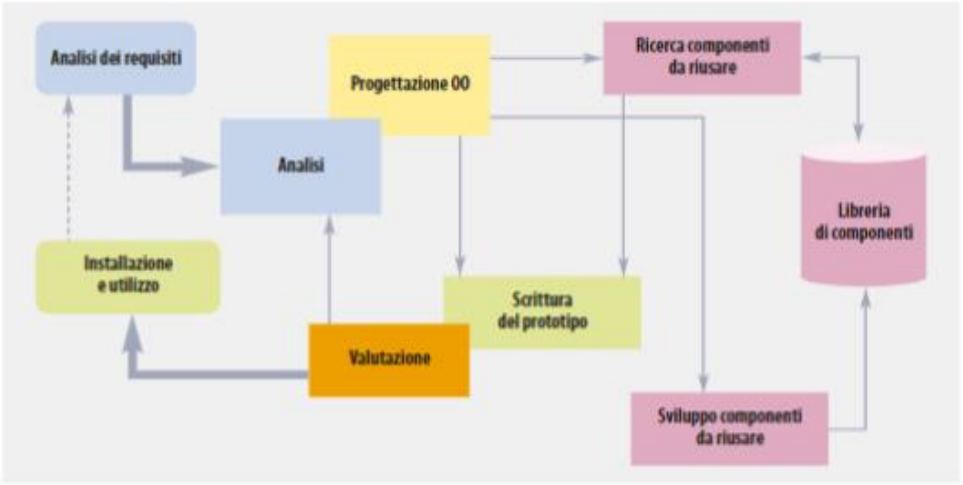
> Ereditarietà, incapsulamento e polimorfismo sono la "triade" di nuovi strumenti da comprendere e sfruttare per realizzare la programmazione di sistemi modulari, di buona qualità, con dimensioni scalabili, di facile manutenzione e riutilizzabili.

Le tecnologie OO (*Object Oriented*) permeano tutto il ciclo di sviluppo del software e in esso possiamo individuare tre distinte macrofasi, in analogia col modello a cascata.



Nella programmazione a oggetti tali fasi prendono il nome di:
- OOA: *Object Oriented Analysis*;
- OOD: *Object Oriented Design*;
- OOP: *Object-Oriented Programing*.

Il processo di sviluppo OO si può sintetizzare come nello schema di seguito proposto.



Si noti come è stato evidenziato il fatto che un obiettivo è quello di "cercare di riutilizzare" componenti già pronti, disponibili nelle librerie e quindi già "funzionanti".

Di norma vengono creati e raffinati prototipi funzionanti del progetto software o di sue parti secondo l'approccio incrementale-iterativo, cioè:
- si individuano sottoparti relativamente autonome;
- si realizza il prototipo di una di esse;
- si prosegue con altre parti;
- si aumenta progressivamente l'estensione e il dettaglio dei prototipi, tenendo conto delle altre parti interagenti.

I progettisti del modello a oggetti utilizzano quasi esclusivamente un unico linguaggio di modellazione di tipo grafico, l'UML (*Unified Modeling Language*), sviluppato verso la metà degli anni Novanta da Grady Booch, James Rambaugh, Ivar Jacobson.

L'UML è un linguaggio semiformale e grafico che utilizza specifici diagrammi per specificare, visualizzare, costruire e documentare gli artefatti di un sistema software: non è solo un linguaggio per la modellazione, ma un linguaggio per la modellazione orientata agli oggetti.

L'UML include quindi sia l'analisi che la progettazione orientata agli oggetti, cioè sia la OOA e OOD. L'analisi consente di capire cosa deve fare il sistema, senza occuparsi dei dettagli implementativi, mentre la progettazione permette di capire come il sistema raggiunge il suo scopo, come viene implementato.

VERIFICA... le conoscenze

SCELTA MULTIPLA

1 L'acronimo OOP significa:
- **a** Object Objective Programming
- **b** Objective Object Processing
- **c** Object Oriented Programming
- **d** Oriented Object Processing

2 Quali delle seguenti fasi del modello a cascata non sono nel corretto ordine? *(2 risposte)*
- **a** Analisi del problema
- **b** Progettazione
- **c** Codifica
- **d** Installazione
- **e** Test
- **f** Manutenzione

3 Quale tra le seguenti non è una caratteristica di un sistema software?
- **a** Ben organizzato
- **b** Modulare
- **c** Protetto
- **d** Semplice
- **e** Riusabile
- **f** Riconfigurabile
- **g** Flessibile

4 Quale tra i seguenti non è un paradigma di programmazione?
- **a** Imperativo
- **b** Funzionale
- **c** Matematico
- **d** Logico
- **e** A oggetti

5 Quale fu il primo linguaggio a oggetti?
- **a** Ada
- **b** Java
- **c** Simula 67
- **d** C++
- **e** SmallTalk

6 Le crisi del software furono di natura (indica quella errata):
- **a** dimensionale
- **b** gestionale
- **c** qualitativa
- **d** quantitativa

7 Quale tra le seguenti affermazioni è errata?
- **a** I metodi corrispondono ai programmi
- **b** Gli attributi corrispondono alle variabili
- **c** Un oggetto è un'istanza di una classe
- **d** Una classe è "un mattone software"

8 L'incapsulamento consiste:
- **a** nell'astrarre gli oggetti
- **b** nel fare in modo che i dettagli della realizzazione restino invisibili all'esterno
- **c** nel propagare automaticamente attributi comuni
- **d** nel realizzare il divide et impera

9 Quale tra i seguenti non è un concetto OOP? *(2 risposte)*
- **a** Incapsulamento
- **b** Automatismo
- **c** Polimorfismo
- **d** Ereditarietà
- **e** Ricorsione

VERO/FALSO

1 Il primo linguaggio a oggetti fu sviluppato negli anni Novanta. V F
2 L'astrazione ha un ruolo fondamentale nelle OOP. V F
3 SmallTalk fu introdotto negli anni '70 ed è il primo linguaggio "puro". V F
4 La classe è un insieme di oggetti. V F
5 L'oggetto è un'istanza di una classe V F
6 Nella OOP le variabili corrispondono agli attributi. V F
7 L'interfaccia serve per interagire con un oggetto. V F
8 Nella OOP l'information hiding viene realizzato mediante l'ereditarietà. V F
9 L'ereditarietà permette di propagare automaticamente attributi comuni. V F

LEZIONE 2

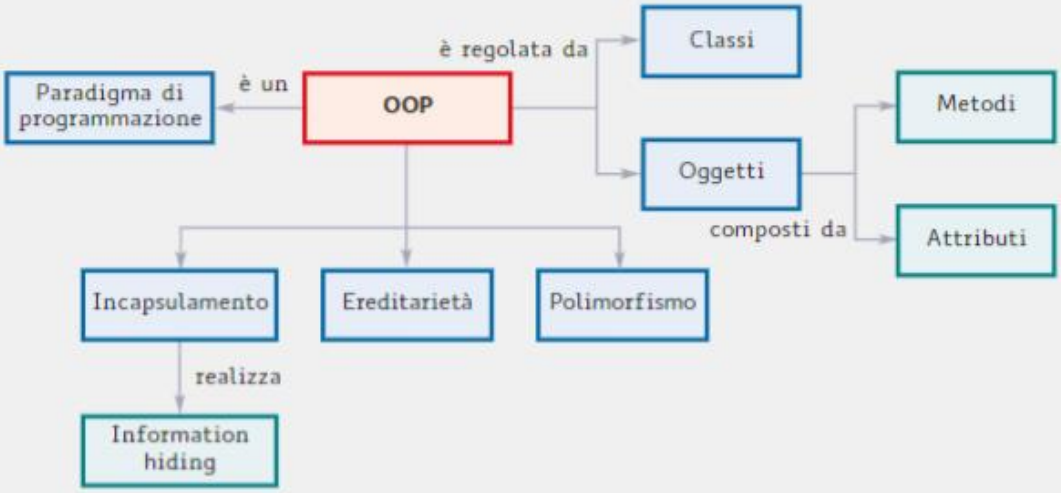
2 Classi e oggetti

IN QUESTA LEZIONE IMPAREREMO...

- a utilizzare il processo di astrazione per modellare le classi
- a rappresentare classi e oggetti
- a utilizzare classi e oggetti

MAPPA CONCETTUALE

didattica inclusiva



Programmazione modulare

Un programma OOP si basa essenzialmente sull'utilizzo di moduli software già preparati (magari da altri) senza la necessità di sapere come ciascuno di essi svolga il proprio compito, semplicemente assemblandoli in un contesto funzionale adeguato allo scopo.

⚠ Nella OOP gli oggetti software sono da considerarsi veri e propri "circuiti integrati software", che i progettisti possono utilizzare nelle loro applicazioni come elementi già completamente funzionanti, testati e "inscatolati" in moduli autonomi protetti e non modificabili.

Benché i concetti di modularità fossero già stati ampiamente "predicati" dal prof. Wirth negli anni Ottanta e quindi i concetti di scomposizione, atomizzazione e riutilizzo siano da considerarsi ben noti ai programmatori, la programmazione OOP va molto oltre, in quanto offre, nei linguaggi, strumenti che automatizzano la realizzazione di queste tecniche integrando il processo di *information hiding* senza che il programmatore debba occuparsene, consentendogli di focalizzarsi esclusivamente su come questi oggetti interagiscono tra loro.
Il progettista software si trova ad avere di fatto a disposizione una vera e propria "scatola nera", di cui non conosce il contenuto ma solamente le funzionalità e che può usare e riusare tutte le volte che vuole (concetto di incapsulamento).

Inoltre, la novità è che all'interno della scatola non ci sono solo le funzioni ma anche i dati: ogni elemento può quindi vivere di vita propria, indipendentemente dal contesto in cui viene utilizzato.

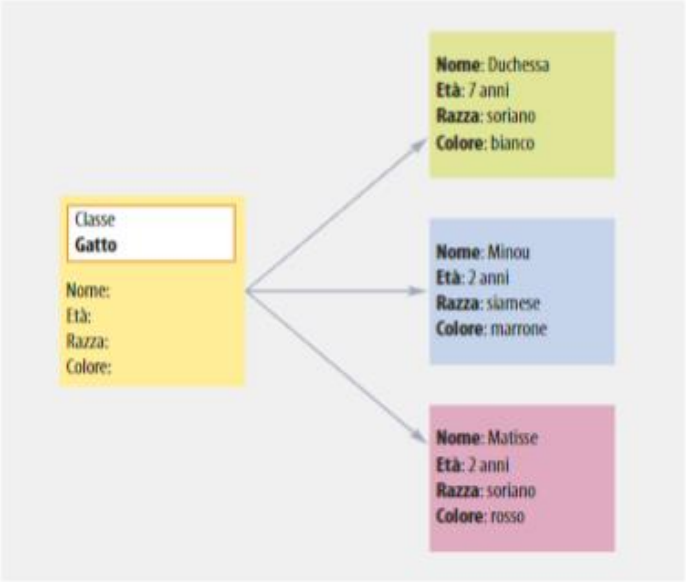
Gli oggetti e le classi

L'elemento fondamentale nella OOP è l'oggetto, che è appunto un "insieme di dati e di funzioni" che prende "vita" da una classe: questa è la vera rivoluzione per cui la OOP può considerarsi come un nuovo paradigma che riunisce codice e dati dopo che per anni la tendenza è stata quella di separarli completamente, come possiamo riscontrare nei sistemi a database.

Nella programmazione OOP l'oggetto è dunque un'istanza di una classe.

Tra classe e oggetto vi è la stessa relazione che sussiste fra un tipo e una variabile del paradigma imperativo.
Sottolineiamo inoltre anche due altri aspetti fondamentali di questa relazione:

- le classi definiscono dei prototipi per gli oggetti: sono una specie di "stampino" che può essere utilizzato per realizzare tanti oggetti dello stesso tipo;
- all'interno delle classi si incapsulano i dati e le funzioni, dove le funzioni sono strettamente correlate con i dati che manipolano.



Gli oggetti

Ogni oggetto, anche se appartiene alla stessa classe, è generalmente differente dagli altri, cioè ha proprietà (attributi) che lo caratterizzano, mentre condivide con gli altri suoi simili il comportamento.
Formuliamo adesso una prima definizione di oggetto.

➡ Un oggetto è un'entità astratta dotata di specifici attributi e in grado di cooperare con altri oggetti svolgendo specifiche azioni.

Un oggetto è quindi la rappresentazione di una qualsiasi cosa che ha uno stato e un comportamento: per meglio familiarizzare con il processo di astrazione e descrizione degli oggetti, consideriamo di seguito alcuni esempi tratti dalla vita quotidiana (compiti di realtà).

ESEMPIO

1 Oggetto Gatto

L'oggetto Gatto:
- ha caratteristiche specifiche: peso, colore del pelo, età
- compie azioni, come: correre, dormire e bere il latte

2 Oggetto Automobile

L'oggetto Automobile:
- ha caratteristiche specifiche: marca, modello, peso, colore, posti ecc.
- esegue azioni, come: avviamento, arresto, accelerazione, cambio marcia, frenata, suono del clacson, avvio tergicristalli ecc.

3 Oggetto Cellulare

L'oggetto Cellulare:
- ha caratteristiche specifiche: tipo del display, peso, colore, marca, modello ecc.;
- esegue azioni, come: invio di sms, composizione di un numero o accensione e spegnimento automatici.

Da questi esempi, possiamo innanzitutto notare che in ciascuno di essi è possibile riscontrare le caratteristiche proprie di quel particolare oggetto, cioè gli elementi che lo distinguono dagli oggetti a esso simili, i valori specifici che hanno gli attributi di quell'elemento particolare (istanza) e che sono i dati interni all'oggetto stesso distinguendolo dagli oggetti a lui "simili" (della stessa classe).
La seconda osservazione è che tutti gli oggetti "simili" hanno un medesimo comportamento, che è l'insieme delle azioni (operazioni) che sono in grado di eseguire (metodi).

⚠ Due oggetti Gatto si distinguono tra loro per il colore, la razza, l'età, il peso, il nome proprio ecc., cioè per tutte le caratteristiche fisiche individuali di ogni animale, ma tutti i gatti mangiano, corrono, bevono il latte ecc. allo stesso modo.
Così pure le Automobili: tutte sono costruite per trasportare le persone, si muovono su ruote, hanno bisogno di combustibile e ognuna ha un colore, una marca, una cilindrata, un proprietario, una targa che la distingue dalle altre.

Le classi

La classe costituisce la base della OOP: essa descrive la natura di un oggetto e ne definisce il suo comportamento.
Una classe è un "modello" per un insieme di oggetti "analoghi", caratterizzati:
– dalla stessa rappresentazione interna;
– dalle stesse operazioni con lo stesso funzionamento, che permettono di descrivere un insieme di oggetti che hanno gli stessi attributi (variabili) ed eseguono le stesse operazioni (funzioni o metodi).

La classe si può pertanto definire come lo schema di base dal quale è possibile creare i singoli oggetti.

TDA
In un linguaggio di programmazione tradizionale il concetto di TD, tipo di dato, è quello di insieme dei valori che può assumere un dato (una variabile): il tipo di dato astratto (TDA oppure ADT, *Abstract Data Type*) estende questa definizione includendo anche l'insieme di tutte e sole le operazioni possibili su dati di quel tipo.

La descrizione dei metodi e delle variabili è contenuta all'interno della classe e non negli oggetti, che sono singole istanze della classe stessa.
Tutti gli oggetti istanza di una stessa classe:
– condividono la stessa struttura interna e lo stesso funzionamento;
– hanno ciascuno la propria identità e il proprio stato.

Nella progettazione di una classe è necessario:
– definire (e nascondere) la rappresentazione dei dati all'utente;
– definire tutte le operazioni applicabili agli oggetti come elementi di un tipo di dato astratto;
– garantire che l'utente possa manipolare gli oggetti solo tramite operazioni del tipo di dato astratto (TDA) a cui gli oggetti appartengono (protezione).

Nei linguaggi di programmazione la classe viene dichiarata per mezzo della parola chiave class, con la seguente sintassi:

Linguaggio Java

```
public class Nome_classe{
  // area attributi
  private int xxx;
  private int yyy;
  public Nome_classe(){
    // da completare
  }
  // metodi vari
  public int calcolaxxx(){
    // codice del metodo
    ...
  }
  protected int calcolayyy(){
    // codice del metodo
    ...
  }
  private int calcolazzz(){
    // codice del metodo
    ...
  }
  ...
}
```

All'interno della classe è possibile specificare il tipo di visibilità di dati (e, come vedremo, dei metodi) e codice tramite i seguenti modificatori di accesso:
– public (publica): permette l'utilizzo di variabili e metodi dall'esterno;
– private (privata): nasconde metodi e codice che risulteranno non utilizzabili;
– protected (protetta): dati e codice nascosti a esterni ma visibili da chi eredita.

Il linguaggio C++ separa nella sua definizione le tre aree con i diversi livelli di visibilità, mentre nel linguaggio Java la differenziazione avviene semplicemente anteponendo la parola riservata corrispondente.

ESEMPIO

Iniziamo a scrivere il codice per la classe Rettangolo indicando la parte privata e pubblica.

Linguaggio Java

```
public class Rettangolo{
  private int base;
  private int altezza;
  // Metodi vari
  public int calcolaPerimetro(){
    return 2*(base + altezza);
  }
  public int calcolaArea(){
    return base * altezza;
  }
}
```

Per realizzare in pratica il concetto di incapsulamento, il valore di default da assegnare a tutti gli attributi della classe deve essere private.

Come secondo esempio, definiamo una classe che chiamiamo Frazione che ha due attributi di tipo intero, il numeratore e il denominatore.

Linguaggio Java

```
public class Frazione{
  private int numeratore;
  private int denominatore;
  public int stampa(
    ...
  }
  public boolean valida(){
    ...
  }
  public double calcola(){
    ...
  }
  public double semplifica(){
    ...
  }
}
```

Rappresentazione in UML

Nell'ambito dell'ingegneria del software è stato definito un apposito linguaggio per la descrizione grafica dei progetti, il linguaggio UML, nel quale viene utilizzato un simbolismo universalmente riconosciuto da tutti gli sviluppatori a oggetti per rappresentare le classi e gli oggetti.

Per le classi sono previsti due livelli di rappresentazione:
- sintetica, che utilizza un rettangolo con tre sezioni dove viene indicato solo il nome della classe;
- completa, che utilizza un rettangolo con indicati anche gli attributi e i metodi.

Anche per gli oggetti la notazione è simile, ma il rettangolo ha solo due sezioni:
– la rappresentazione sintetica utilizza un rettangolo con solo il nome;
– la rappresentazione completa utilizza un rettangolo con indicati anche i valori degli attributi.

Oggetto: classe

Oggetto: classe
Attributi: valore

ESEMPIO

Definiamo la classe Gatto e due istanze, l'oggetto Gatto Pippo e l'oggetto Gatto Malachia.

RAPPRESENTAZIONE SINTETICA

Classe

Gatto

Rappresentazione della classe

Oggetto

Pippo: Gatto

Oggetto

Malachia: Gatto

Rappresentazione degli oggetti

RAPPRESENTAZIONE COMPLETA

Classe

Gatto
pelo razza colore nome età cosaFa
mangia dorme corre faFusa gioca

Oggetto

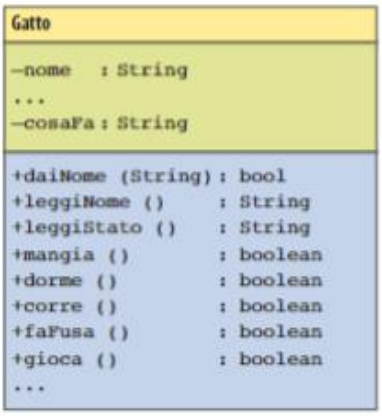
Pippo: Gatto
pelo : "corto" razza : "soriano" colore : "rosso" nome : "Pippo" età : "10-02-17" cosaFa : "ci guarda"

Oggetto

Malachia: Gatto
pelo : "lungo" razza : "di strada" colore : "bianconero" nome : "Malachia" età : "12-03-14" cosaFa : "sonnecchia"

Gli attributi permettono di memorizzare lo stato attuale di un oggetto, cioè l'insieme dei valori che descrivono l'oggetto; tali valori prendono il nome di variabili di esemplare (o variabili di istanza, *instance variables*).

La rappresentazione UML della visibilità viene effettuata semplicemente scrivendo i caratteri + e – davanti ai metodi e agli attributi:
– sta per private;
+ sta per public.



Completiamo quindi la classe Gatto aggiungendo la visibilità agli attributi. Per semplicità, ci limitiamo a considerare due attributi, nome e cosaFa: aggiungiamo quindi due metodi necessari per leggere e scrivere in essi.

Avendo infatti reso privati gli attributi, l'unico modo per poterli leggere e scrivere in un'applicazione è quello di utilizzare metodi pubblici.

AREA DIGITALE

Naming delle classi

Esistono molti programmi per realizzare graficamente i diagrammi UML: tra di essi forse il più utilizzato, sia per la sua semplicità sia perché è completamente gratuito , è StarUML: l'indirizzo del sito di riferimento è http://staruml.io/, dove è possibile effettuare il download del software per i diversi sistemi operativi.
Oltre a questo, altri due programmi molto utilizzati sono:
– ArgoUML, che è un prodotto gratuito e completo scaricabile all'indirizzo http://argouml.tigris.org/.
– yUML, che è un prodotto che offre la possibilità di creare diagrammi UML all'interno del proprio browser, esportarli come immagine o pubblicarli direttamente senza alcuna istallazione: l'indirizzo di riferimento è il seguente: www.yuml.me/diagram/scruffy/class/draw.

METTITI ALLA PROVA

→ • Definizione sintetica di una classe • Definizione completa di una classe

Definisci la classe Automobile e danne la rappresentazione sia sintetica sia completa istanziando successivamente due oggetti.
Scarica quindi il software StarUML e realizza tramite esso il diagramma UML.

VERIFICA... le conoscenze

SCELTA MULTIPLA

1 Un oggetto è:
- **a** una definizione di tipo
- **b** un elemento di un metodo
- **c** un'entità astratta
- **d** un insieme di metodi e attributi
- **e** una istanza di una classe

2 Per definire una classe è necessario (indica l'affermazione errata):
- **a** definire i tipi di dati
- **b** nascondere la rappresentazione dei dati all'utente
- **c** istanziare almeno un oggetto
- **d** definire tutte le operazioni applicabili agli oggetti
- **e** garantire che l'utente possa manipolare gli oggetti

3 Due classi sono uguali se: (*2 risposte*)
- **a** hanno gli stessi attributi
- **b** hanno gli stessi metodi
- **c** non ha senso avere due classi uguali
- **d** hanno gli stessi attributi e metodi

4 Quale di queste affermazioni riferite agli oggetti della stessa classe è falsa?
- **a** Due oggetti hanno i medesimi attributi
- **b** Due oggetti hanno il medesimo comportamento
- **c** Ogni oggetto compie delle azioni personalizzate
- **d** Ogni istanza ha propri valori per gli attributi

5 La visibilità degli attributi può essere: (*2 risposte*)
- **a** public
- **b** private
- **c** reserved
- **d** readonly

6 UML è l'acronimo di:
- **a** Unified Modeling Language
- **b** Uniform Moduling Language
- **c** Unified Moduling Language
- **d** Uniform Modeling Language

7 I metodi pubblici:
- **a** possono essere richiamati fuori dalla classe
- **b** possono essere richiamati solo con attributi pubblici
- **c** possono essere richiamati solo dentro la classe
- **d** nessuna delle affermazioni precedenti

8 Il processo di astrazione:
- **a** è unico per ogni classe
- **b** se corretto porta alla modellizzazione univoca
- **c** crea una classe in funzione delle esigenze specifiche
- **d** crea una classe per ogni oggetto diverso

VERO/FALSO

1 La descrizione dei metodi è contenuta all'interno della classe. V F
2 La descrizione degli attributi è contenuta all'interno degli oggetti. V F
3 Due oggetti della stessa classe hanno il medesimo comportamento. V F
4 Due oggetti della stessa classe possono avere attributi diversi. V F
5 Lo stato di un oggetto viene definito dai suoi attributi. V F
6 La classe rappresenta la parte statica della OOP, l'oggetto la parte dinamica. V F
7 È possibile leggere un attributo privato dall'esterno della classe. V F
8 La modellizzazione può portare a classi diverse per gli stessi oggetti. V F

VERIFICA... le competenze

Verifica di fine Lezione

ESERCIZI

AREA DIGITALE
Esercizi per l'approfondimento

Dopo aver progettato la classe realizza il diagramma UML utilizzando StarUML.

1. Definisci la classe Aereo e quindi due sue istanze, assegnando i valori per gli attributi e dando una rappresentazione sia sintetica sia completa.
2. Definisci la classe Televisione dando la rappresentazione sia sintetica sia completa anche di una sua istanza.
3. Definisci la classe Treno dando la rappresentazione sia sintetica sia completa anche di una sua istanza.
4. Definisci una classe che rappresenti un Rettangolo e successivamente crea due sue istanze, assegnando valori per la base e l'altezza.
5. Definisci la classe Telefono dando la rappresentazione sia sintetica sia completa anche di una sua istanza.
6. Crea la classe Massimi che contiene i metodi max() e min(), che calcolano rispettivamente il valore massimo e minimo di due numeri passati come parametri per i tipi di dati int, long, float e double.
7. Definisci una classe Convertitore, in grado di convertire valute diverse, per esempio Dollaro - Euro. Per il programma si richiede la scrittura di almeno un metodo per il calcolo, un metodo di stampa dei valori risultanti e la definizione di uno o più attributi privati ove memorizzare i dati.
8. Crea una classe Negozio e due oggetti, sapendo che ha almeno i seguenti attributi:
   - string proprietario;
   - string nomeNegozio.
9. Crea una classe ContoCorrente e due oggetti, sapendo che ha almeno i seguenti attributi:
   - string nome;
   - string codiceConto;
   - double saldo.
10. Definisci una classe Libro contenente i seguenti attributi: nome del libro, prezzo, numero di scaffale, numero di pagine, casa editrice.
    Inoltre definisci i metodi con i seguenti compiti:
    - inizializzare i campi dati dell'oggetto classe;
    - stampare tutti i dati dell'oggetto;
    - diminuire del 10% il prezzo del libro in oggetto.
11. Crea la classe Prodotto contenente i seguenti attributi che definiscono lo stato dell'oggetto:
    - int codice;
    - string descrizione;
    - double prezzo.
    - string categoria;
    - string settore;

LA SFIDA

LA CLASSE FATTURA

Crea la classe Fattura contenente i seguenti attributi che definiscono lo stato dell'oggetto:

- Cliente cliente
- int progressivo
- int numeroProdotti
- double imponibile
- double IVA
- double lordo

e gli attributi necessari per gestire le scadenze di pagamento, che possono essere dilazionate anche a 120 gg (RIBA a 30/60/90/120).
Definisci tutti i metodi e rappresenta la classe con StarUML.

LEZIONE 3

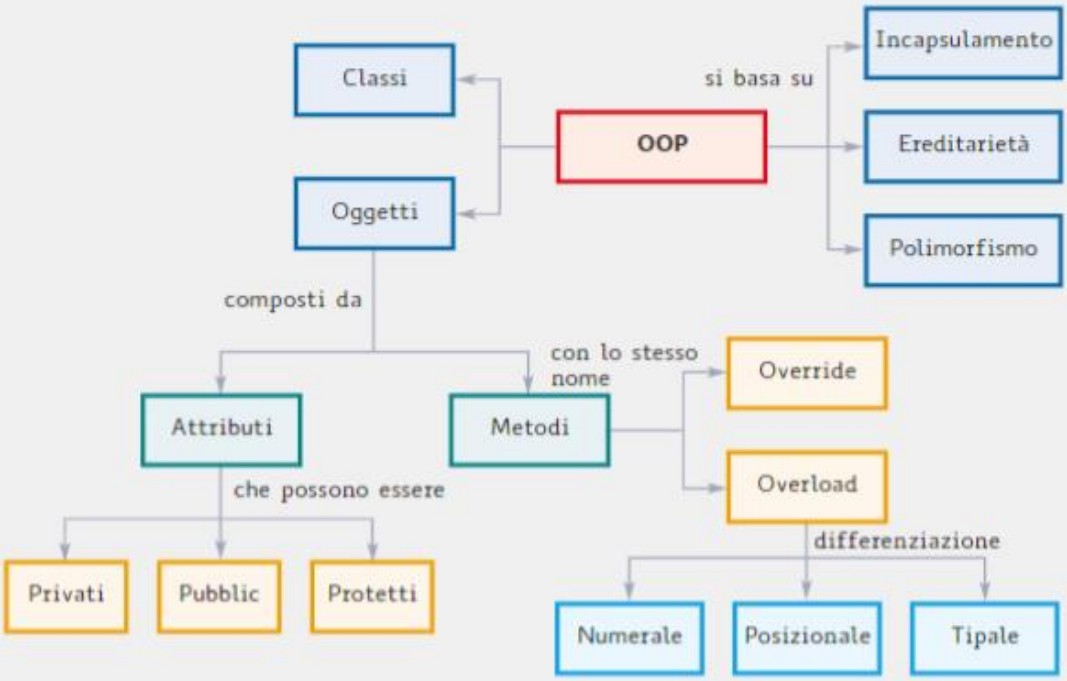
3 Metodi e creazioni di oggetti

IN QUESTA LEZIONE IMPAREREMO...

- a scrivere il codice dei metodi
- il concetto di costruttore e distruttore
- il concetto di overloading

MAPPA CONCETTUALE

didattica inclusiva



La scrittura dei metodi

Abbiamo detto, nella precedente lezione, che la classe è composta da attributi e metodi, dove i metodi non sono altro che le funzioni che possono modificare il valore degli attributi.
In base alle operazioni che eseguono, possiamo distinguere alcune categorie di metodi:
– costruttori/distruttori, che servono per creare l'oggetto in memoria, cioè allocare lo spazio necessario per memorizzare gli attributi e rilasciarlo al termine dell'utilizzo dell'oggetto;

– modificatori, chiamati anche metodi getter/setter, che servono per a potere leggere/scrivere negli attributi che, per rispettare l'incapsulamento, vengono dichiarati privati.

Affinché un metodo possa essere utilizzato al di fuori della classe deve essere dichiarato di tipo public: l'insieme dei metodi public costituisce l'interfaccia della classe che gli permette di interagire col mondo esterno.

Costruttori

La OOP mette a disposizione un tipo particolare di metodi che servono per la creazione dell'oggetto e l'eventuale inizializzazione dei suoi attributi: questi metodi prendono il nome di metodi costruttori.

Un metodo costruttore è un metodo particolare, che si distingue dagli altri per tre elementi:
- ha lo stesso nome della classe;
- non ha tipo di ritorno, nemmeno void;
- ha visibilità public.

Per la codifica delle istruzioni di controllo utilizziamo le stesse istruzioni finora studiate nella programmazione imperativa, con la medesima sintassi.
Vediamo un esempio.

ESEMPIO

Per aggiungere un metodo costruttore nella classe Frazione scriviamo un metodo con lo stesso nome della classe, come indicato di seguito.

Linguaggio Java

```
public class Frazione{
  // attibuti
  private int numeratore;
  private int denominatore;
  // metodi costruttore
  public Frazione(){
    // codice
  }
```

Il costruttore di default che appare nell'esempio è il costruttore che viene automaticamente creato quando si dichiara una classe e non contiene alcun codice: se si desidera far effettuare delle operazioni a tale costruttore, come l'inizializzazione degli attributi, è necessario completare il suo codice.

Oltre al costruttore di default, pertanto, nella classe si inseriscono generalmente altri costruttori che però hanno uno o più parametri in ingresso, in modo che, alla creazione dell'oggetto, l'utilizzatore possa contemporaneamente assegnare dei valori ai suoi attributi.

Riprendiamo quindi il precedente esempio e completiamolo.

ESEMPIO

Linguaggio Java

```
public class Frazione{
  // attributi
  private int numeratore;
  private int denominatore;
  // metodi costruttore
  public Frazione(){
    // codice
  }
  public Frazione(int num, int den){
    numeratore = num;
    denominatore = den;
    semplifica();
  }
```

Overloading

La possibilità offerta dai linguaggi a oggetti di consentire la coesistenza di più metodi con lo stesso nome prende il nome di overloading.

Una classe può contenere più costruttori e metodi con lo stesso nome purché distinti da intestazioni (signature) diverse, cioè:
- con un numero diverso di parametri;
- con numero uguale di parametri ma di tipo diverso;
- con numero uguale di parametri ma con ordine diverso.

Le condizioni a cui devono sottostare le liste dei parametri riguardano i loro tipi e non gli identificatori e sono necessarie per permettere l'univoca individuazione del metodo all'atto della chiamata da parte del sistema: ricordiamo che il prototipo è costituito dai parametri formali e nella chiamata vengono passati i valori attuali, quindi due metodi con lo stesso numero e tipo di parametri formali, anche se hanno identificatori diversi, non verrebbero distinti.

ESEMPIO

Prendiamo per esempio la classe Contatore e ipotizziamo di avere due costruttori come i seguenti:

Linguaggio Java

```
Contatore(int val, int passo)
Contatore(int passo, int val)
```

Questi sono solo apparentemente diversi, ma sono identici formalmente.

Infatti, se effettuiamo due ipotesi di chiamata come le seguenti:

Linguaggio Java

```
Contatore(0, 1)
Contatore(1, 0)
```

quale dei due costruttori viene eseguito? L'ambiguità è evidente!

È quindi necessario che siano diversi in numero oppure nel tipo dei parametri.

Non ci sono problemi, invece, in tutti i casi seguenti, anche quando abbiamo lo stesso numero di parametri, sia in numero che in tipo, dato che in questi casi hanno ordine diverso.

Linguaggio Java

```
Contatore()
Contatore(int val, int passo)
// stesso numero ma di tipo diverso
Contatore(int val)
Contatore(double passo)
// stessi parametri ma ordine diverso
Contatore(int val, double passo)
Contatore(double val, int passo)
```

Questi sei metodi costruttori possono coesistere in una classe, e il programmatore può scegliere quale utilizzare al momento della creazione degli oggetti, secondo la necessità dell'applicazione che sta realizzando

Lo stesso discorso vale anche per tutti gli altri metodi: ricapitolando, possiamo quindi avere tre possibili forme di scrittura di metodi che hanno lo stesso nome, ma con differenze di carattere:
– tipale: il numero dei parametri è lo stesso, ma i tipi sono diversi;

Linguaggio Java

```
somma(int n1, int n2)
somma(int n1, double n2)
somma(int n1, char n2)
```

– numerico: il numero dei parametri è diverso;

Linguaggio Java

```
somma(int n1)
somma(int n1, int n2)
somma(int n1, int n2, int n3)
```

– posizionale: il numero e il tipo dei parametri sono gli stessi, ma la loro posizione è scambiata.

Linguaggio Java

```
somma(double n1, int n2)
somma(int n1, double n2))
```

Distruzione degli oggetti

Analogo discorso può essere fatto anche per il metodo distruttore ma, poiché le occasioni del suo overloading sono estremamente rare, a questo punto della trattazione non riportiamo alcun esempio.
Ricordiamo solamente che il distruttore serve per effettuare tutte quelle azioni di "pulizia" che risultano necessarie prima della deallocazione dell'oggetto, ovvero:
– deallocare i dati dinamici che sono stati allocati da funzioni membro dell'oggetto;
– chiudere i file che sono stati aperti dalle funzioni membro dell'oggetto.

La necessità di scrivere e richiamare direttamente il distruttore esiste solo nel caso di progetti articolati o complessi, mentre nelle classi semplici esiste un distruttore standard, che viene automaticamente chiamato dal sistema ogniqualvolta un oggetto viene deallocato in una delle tre seguenti situazioni:
– l'oggetto è locale e si esce dal blocco in cui è dichiarato;
– l'oggetto è nella memoria dinamica (heap) e viene rimosso il riferimento a esso;
– l'oggetto è un membro di un altro oggetto e quest'ultimo viene deallocato.

Il metodo distruttore viene indicato con la seguente notazione:

Linguaggio Java

```
// richiamato in automatico
public void finalize(){
}
```

Generalmente non contiene codice: a volte viene inserita una istruzione di "echo", cioè una semplice istruzione di scrittura sullo schermo che visualizza lo stato per le fasi di debugging.

Linguaggio Java

```
// richiamato in automatico con personalizzazione
public void finalize(){
  System.out.println ("E' stato distrutto un oggetto ");
}
```

Metodi getter/setter

Il linguaggio Java non prevede particolari meccanismi per implementare i metodi getter/setter: per ogni attributo vengono scritte due funzioni che, a seconda delle situazioni, possono essere definite pubbliche o private, antecedendo al nome del metodo l'opportuno qualificatore (o modificatore).
Nel seguente esempio sono riportati i metodi per la gestione dell'attributo passo della classe Contatore.

ESEMPIO

Linguaggio Java

```
public setVal(int n1)                    // metodo setter
  val = n1;
return;
public getVal()                          // metodo getter
return val ;

public setPasso(int n1)                  // metodo setter
  passo = n1;
return;
public getPasso()                        // metodo getter
return passo;
```

Creazione di oggetti in Java

Una volta realizzata la classe, il linguaggio Java mette a disposizione una parola chiave per la creazione degli oggetti da aggiungere alla solita dichiarazione delle variabili: la parola new.

```
Contatore conta1 = new Contatore();        // crea un oggetti di tipo Contatore
```

Gli oggetti si comportano come normali variabili rispetto alla visibilità e al ciclo di vita: ogni volta che si istanzia una classe all'interno di una funzione, viene automaticamente invocato il costruttore dell'oggetto e all'uscita dalla funzione in cui è stato dichiarato avviene la sua distruzione, in perfetto accordo con il ciclo di vita delle variabili locali.

> Se non diversamente indicato dal programmatore, in entrambi i casi il compilatore invoca il costruttore e il distruttore di default.

Nell'esempio che segue dichiariamo due oggetti, dei quali il secondo richiama un costruttore personalizzato.

```
Contatore conta1 = new Contatore();      //costruttore di default
Contatore conta2 = new Contatore(10,2);  //costruttore personalizzato
```

Invocazione dei metodi

L'invocazione dei metodi avviene mediante la dot notation, con la seguente sintassi.

```
<oggetto>.<metodo>
```

Come per gli attributi, anche per i metodi sono previsti i modificatori di visibilità (descrittore): possiamo quindi avere metodi public, che possono essere invocati dall'esterno, e metodi private, che invece hanno visibilità solo all'interno della classe.

Come esempio di utilizzo dei metodi completiamo il codice che utilizza due oggetti della classe Frazione: dopo aver creato due oggetti, utilizziamo i metodi setter per assegnare il valore agli attributi: richiamiamo il metodo semplifica() che, al suo interno, richiama il metodo MCD(), una funzione che calcola il MCD con l'algoritmo di Euclide (del quale omettiamo la codifica) e, per ultimo, il metodo stampa(), che visualizza il valore della frazione.

Linguaggio Java

```
public static void main(String[] args)
{
  Frazione f1 = new Frazione();
  Frazione f2 = new Frazione(21, 14);
  f1.setNumeratore(12);
  f1.setDenominatore(3);
  f1.semplifica();
  f1.stampa();
}
```

Duplicazione di oggetti e costruttore di copia

In alcune situazioni risulta necessario effettuare la copia di oggetti già presenti e, mentre il linguaggio C++ mette a disposizione un apposito strumento, che prende il nome di costruttore di copia, nel linguaggio Java questo non è formalizzato a livello di linguaggio.
Quindi in Java il termine "costruttore di copia" è un'espressione un po' informale che descrive un costrutto/sintassi che può creare un oggetto partendo dai dati di un altro oggetto *dello stesso tipo*

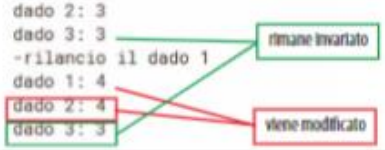
e, quindi, viene realizzato "semplicemente" con un costruttore che ... riceve un oggetto dello stesso tipo della classe!
Come esempio, definiamo una semplice classe Dado dove con la costruzione dell'oggetto assegniamo un valore casuale compreso tra 1 e 6 all'unico membro intero faccia.
Aggiungiamo nella classe un metodo lancia() che permette di effettuare un nuovo lancio del dado in modo da generare casualmente un nuova valore.
Nel metodo main() creiamo un oggetto dado1, ne facciamo una copia in un secondo oggetto dado2 direttamente copiando i puntatori ed una copia dado3 utilizzando il costruttore di copia.

```
public class Dado{
  private int faccia;
  public Dado() {
    lancia();
  };
  // costruttore  di copia
  // copia gli attributi in un altro oggetto
  public Dado(Dado altro) {
    this.faccia = altro.faccia;
  }
  // getter/setter
  int getFaccia() {
    return faccia;
  };
  void setFaccia(int num) {
    faccia = num;
  };
  // lancia il dado casualmente
  void lancia() {
    faccia = (int)(1 + Math.random() * 6);
  };
  // stampa i valori
  void stampa(char dado) {
    System.out.println("dado "+dado+": "+getFaccia());
  };
}
```

```
import java.util.Scanner;
public class ProvaDado {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.println("Copia di oggetti");
    // creazione di due oggetti
    Dado dado1 = new Dado();
    // creo dado 2 copiando direttamente
    Dado dado2 = dado1;
    // creo dado 3 col costruttore di copia
    Dado dado3 = new Dado(dado1);
    // visualizza i dadi
    System.out.println("-valori iniziali dei dadi");
    dado1.stampa('1');
    dado2.stampa('2');
    dado3.stampa('3');
    // lancio il dado1
    System.out.println("-rilancio il dado 1");
    dado1.lancia();
    // visualizza i dadi
    dado1.stampa('1');
    dado2.stampa('2');
    dado3.stampa('3');
  }
}
```

Osservando il risultato di un'esecuzione possiamo osservare come "tutto vada a buon fine" nel caso del dado3, cioè se effettuiamo un nuovo lancio del dado1 questo NON compromette/modifica i valori del dado3, mentre il dado2 "subisce le stesse sorti" del dado1.

```
 Copia di oggetti
-valori iniziali dei dadi
dado 1: 3
dado 2: 3
dado 3: 3
-rilancio il dado 1
dado 1: 4
dado 2: 4
dado 3: 3
```



Con la copia diretta, l'area dinamica risulta condivisa dai due oggetti, mentre noi desideriamo che dopo la copia ogni oggetto abbia vita propria, con tutti i membri autonomi in area riservata, e questa situazione si ottiene solo con il costrutttore di copia.

Uguaglianza fra oggetti copia

È doveroso effettuare un'osservazione anche per il confronto tra oggetti: è possibile effettuare una verifica *superficiale* dell'uguaglianza oppure una verifica *profonda*:

- l'uguaglianza *superficiale* dell'oggetto viene effettuata:
  - con l'operatore '==', che esegue un confronto fra i valori dei riferimenti, ovvero fra i due indirizzi di memoria in cui si trovano gli oggetti;
  - con la funzione equals(), disponibile per ogni classe che, se non ridefinita, si comporta come l'operatore '==';
- l'uguaglianza *profonda* fra oggetti viene effettuata riscrivendo la funzione equals().

Nel programma di prova dell'esempio precedente aggiungiamo il metodo ottenuto mediante overriding della funzione equals(), nella quale effettuiamo il confronto tra singoli attributi per "decretare" se i due oggetti sono uguali in profondità:

```
public boolean equals(Object o) {
  if (o != null && getClass().equals(o.getClass())) {
    Dado b = (Dado)o;
    return (b.faccia == faccia);
  }
  else return false;
}
```

```
System.out.println("-confronto dado1 == dado3");
if (dado1 == dado3)
  System.out.println("sono uguali!");
else
  System.out.println("sono diversi!");
System.out.println("-confronto dado1 -equals(dado3)");
if (dado1.equals(dado3))
  System.out.println("sono uguali!");
else
  System.out.println("sono diversi!");
```

Mandiamo in esecuzione il programma e confrontiamo dado1 con dado3 (naturalmente il confronto tra dado1 e dado2 è inutile, essendo sicuramente uguale sia il riferimento sia il valore degli attributi).

```
Copia di oggetti
-valori iniziali dei dadi
dado 1: 1
dado 3: 1
-rilancio il dado 1
dado 1: 1
dado 3: 1
-confronto dado1 == dado3
Diversi!
-confronto dado1 = equals(dado3)
Uguali!
```

uguaglianza dei riferimeti

uguaglianza dei valori

Otteniamo che i due oggetti NON sono uguali in una verifica superficiale, mentre lo sono in una verifica in profondità.

Un esempio completo

Scriviamo un esempio completo dove definiamo la classe Rettangolo che ci permette di calcolare perimetro ed area conoscendo la base e l'altezza e la utilizziamo in un programma di prova dove testiamo tutti i metodi.
Il codice della classe è riportato di seguito: definiamo la classe con i suoi attributi e due costruttori, quindi scriviamo i metodi setter/getter per tutti gli attributi (ne riportiamo solo due coppie a titolo di esempio).

Classe, attributi e costruttori

```
public class Rettangolo{
  private double base;
  private double altezza;
  private double perimetro;
  private double area;

  //Metodi vari
  public Rettangolo(){
    base = 0;
    altezza = 0;
  }
  public Rettangolo(double x1, double x2){
    base = x1;
    altezza = x2;
  }
```

Metodi getter(setter

```
  public void setBase(double x){
    base = x;
  }
  public void setAltezza(double x){
    altezza = x;
  }
  public double getBase(){
    return base;
  }
  public double getAltezza(){
    return altezza;
  }
```

Scriviamo poi i metodi che effettuano il calcolo del perimetro e dell'area.
Per collaudare la nostra classe nel main() scriviamo il codice che crea un oggetto di tipo Rettangolo, legge i valori dei lati e calcola area e perimetro utilizzando i metodi prima definiti.

Metodi

```
public void calcolaPerimetro() {
  perimetro = 2 * (base + altezza);
}
public void calcolaArea() {
  area = base * altezza;
}
public double calcolaDiagonale() {
  return Math.sqrt(Math.pow(base,2)+Math.pow(altezza,2));
}
```

main() di prova

```
public static void main(String[] args) throws Exception{
 Scanner in = new Scanner(System.in);
 double numero;
 Rettangolo tavolo = new Rettangolo();
 System.out.print("Inserisci la base del tavolo: ");
 numero = in.nextDouble();
 tavolo.setBase(numero);
 System.out.print("Inserisci la altezza del tavolo: ");
 numero = in.nextDouble();
 tavolo.setAltezza(numero);
 tavolo.calcolaPerimetro();
 tavolo.calcolaArea();
 tavolo.stampa();
}
```

L'esecuzione del programma produce l'output riportato a lato.

Il codice completo di questo programma lo puoi trovare nel file Rettangolo.java.

```
Inserisci la base del tavolo: 3
Inserisci la altezza del tavolo: 4
 base      : 3.0
 altezza   : 4.0
 perimetro: 14.0
 area      : 12.0
la diagonale e':5.0
```

Un esempio con la classe di prova

Nell'esempio precedente abbiamo inserito all'interno della classe Rettangolo il metodo main() con il codice che ne crea una istanza definendo un oggetto e richiamando su di esso i metodi per collaudarne la loro correttezza: in generale le classi non hanno al loro interno il metodo main() e l'avvio di un programma a oggetti viene effettuato mediante un'apposita classe generalmente chiamata **classe principale** (che equivale alla **classe di prova** utilizzata nelle operazioni di testing), dove sono presenti solo metodi statici.
Realizziamo un semplice esempio di un programma composto da due classi:
– la classe Moneta che definisce gli oggetti, senza il metodo main();
– la classe ProvaMoneta, con il metodo main(), che istanzia uno (o più oggetti) e ne testa i metodi.

In un'applicazione può essere presente solamente una classe che contiene il metodo main().

✔ PROBLEMA SVOLTO PASSO PASSO

✔ Il problema

Il giocatore sceglie "testa" oppure "croce" e il programma effettua 5 lanci di una moneta, visualizzando al termine del gioco il nome del giocatore che risulta vincitore, cioè colui che ha avuto il maggior numero di lanci a favore.

✔ L'analisi e strategia risolutiva

Scomponiamo il problema in due classi.

Per prima cosa realizziamo la classe Moneta che ha un solo attributo, la faccia, che può assumere due soli valori (TESTA = 0, CROCE = 1) legati alla casualità ottenuta in un metodo lancia() che sfrutta la funzione random() predisposta alla generazione di un numero binario.
Completa la classe un metodo che ritorna sotto forma di stringa lo stato della moneta (toString()).

La seconda classe, che chiamiamo ProvaMoneta, contiene esclusivamente il metodo main() che istanzia un oggetto, effettua i lanci previsti, visualizzando l'esito di ogni estrazione e contabilizzando quante volte viene estratto **testa** (oppure **croce**), e individua quale dei due giocatori vince il gioco mostrandone il suo nome sullo schermo.

✔ Il codice delle classi

Java ricerca automaticamente il codice di tutte le classi utilizzate all'interno della directory di lavoro corrente.

La classe Moneta

```
import java.util.Random;
public class Moneta{
  private final int TESTA = 0;
  private final int CROCE = 1;
  private int faccia;

  public Moneta(){
    lancia();
  }

  // lancio casuale della moneta
  public void lancia(){
    faccia = (int)(Math.random() * 2);
  }
  // ritorna VERO se il valore e' testa
  public boolean isTesta(){
    return (faccia == TESTA);
  }
  // ritorna nome della faccia corrente
  public String toString(){
    String nomeFaccia;
    if (faccia == TESTA)
      nomeFaccia = "Testa";
    else
      nomeFaccia = "Croce";
    return nomeFaccia;
  }
}
```

La classe di prova

```
import java.util.Scanner;
public class ProvaMoneta{
  public static void main()throws Exception{
    Scanner in = new Scanner(System.in);
    int sceltaGiocatore;
    int lanciTesta = 0;
    System.out.println("Gioco del testa o croce - 5 lanci");
    System.out.print("Scegli testa (0) oppure croce (1): ");
    sceltaGiocatore = in.nextInt();
    Moneta miaMoneta = new Moneta();
    for (int x = 0; x < 5; x++){
      miaMoneta.lancia();
      System.out.println(miaMoneta.toString());
      if (miaMoneta.isTesta())
        lanciTesta++;
    }
    if((sceltaGiocatore == 0) && (lanciTesta > 2))
      System.out.println( " -> hai vinto !");
    else
      System.out.println(" -> ha vinto il PC!");
  }
}
```

✔ L'esecuzione del programma

```
Gioco del testa o croce - 5 lanci
Scegli testa (0) oppure croce (1): 0
Testa
Croce
Croce
Croce
Croce
 -> ha vinto il PC!
```

METTITI ALLA PROVA

→ • Definizione di una classe • Esecuzione dei metodi

1 Progetta il diagramma UML della classe Triangolo contemplando i tipi di seguito elencati: equilatero, rettangolo, isoscele e scaleno.
Scrivi il codice della classe, compresi i metodi che individuano il tipo di triangolo in base al valore dei tre lati, i metodi che permettono di calcolare l'area, il perimetro e le diagonali nei casi in cui è possibile, e un metodo che visualizza i risultati sullo schermo.
Realizza una programma di prova che richiami tutti i metodi scritti e confronta la tua soluzione con quella riportata nel file Triangolo_solux.java.

2 Successivamente componi il programma in due classi, riportando in un file .h i prototipi e scrivendo la classe di prova ProvaTriangolo dove è presente solo il main() che istanzia i vari oggetti ed effettua su di essi le operazioni sopra elencate.

VERIFICA... le conoscenze

SCELTA MULTIPLA

1 La rappresentazione di una classe usando la notazione UML non richiede di:
- **a** indicare il nome della classe
- **b** riportare l'elenco dei metodi
- **c** riportare l'elenco degli attributi
- **d** riportare l'elenco degli oggetti

2 Un oggetto è:
- **a** una definizione di tipo
- **b** un elemento di un metodo
- **c** un'istanza di una classe
- **d** un insieme di metodi e attributi

3 Un metodo è costituito da (indica l'affermazione errata):
- **a** uno specificatore di accesso
- **b** il tipo di dati restituito dal metodo
- **c** il nome del metodo
- **d** un elenco di parametri di ingresso
- **e** un elenco di parametri di uscita

4 I metodi privati:
- **a** costituiscono l'interfaccia della classe
- **b** devono esser indicati con get/set
- **c** possono essere invocati solo da metodi della stessa classe
- **d** servono a inizializzare gli attributi

5 In una classe:
- **a** deve esserci un costruttore
- **b** possono esserci più costruttori con diverso prototipo
- **c** possono esserci costruttori che ritornano void
- **d** possono esserci più costruttori ma con nome diverso

6 L'overloading consiste:
- **a** nel sovraccaricare le classi
- **b** nel duplicare i metodi
- **c** nell'usare i metodi get/set
- **d** nell'avere metodi con lo stesso nome

7 I metodi possono essere (indica quelli erratl):
- **a** costruttori
- **b** modificatori
- **c** distruttori
- **d** descrittori
- **e** selettori

8 Il distruttore: (*2 risposte*)
- **a** deve sempre essere presente
- **b** serve per eliminare i metodi
- **c** serve per eliminare un oggetto
- **d** serve per eliminare la classe
- **e** non ha parametri

VERO/FALSO

1 Il qualificatore public permette di definire la visibilità degli attributi. V F
2 Il qualificatore public permette di definire la visibilità dei metodi. V F
3 La parte pubblica di una classe è costituita dall'insieme dei metodi e degli attributi private. V F
4 Il qualificatore protected permette di assegnare diritti sui metodi. V F
5 I metodi pubblici costituiscono l'interfaccia della classe. V F
6 Il termine overloading indica la possibilità di avere metodi/funzioni con lo stesso risultato. V F
7 Per rimuovere la classe dalla memoria si usa il distruttore. V F
8 Il distruttore chiude i file che sono stati aperti dalle funzioni membro dell'oggetto. V F

VERIFICA... le competenze

ESERCIZI

AREA DIGITALE
Esercizi per l'approfondimento

Progetta il diagramma UML della classe, quindi realizza in linguaggio di programmazione il codice della classe e un programma di prova dove richiami tutti i metodi che hai scritto.

1 Utilizzando la classe Frazione precedentemente descritta, aggiungi i metodi necessari per effettuare la somma, la differenza e il prodotto di frazioni.

2 Data la classe Rettangolo scrivi il codice di tutti i metodi e una classe di prova che legga i valori per creare due rettangoli e successivamente collaudi tutti i metodi.

3 Realizza la classe Impiegato con tre attributi: cognome, nome e reparto.
Quindi definisci due oggetti impiegati, istanzia tutti gli attributi e, dopo aver visualizzato il loro contenuto mediante il metodo stampa(), distruggi i due oggetti.

4 La seguente classe definisce il concetto di NumeroIntero come oggetto.

```
import java.util.Scanner;
public class Numero {
  private int numeroIntero;
  public NumeroIntero() { ... };
  public NumeroIntero(int n) { ... };
  public void setNumero(int n) { ... };
  public int getNumero() { ... };
  public void stampaNumero() { ... };
  ...
}
```

In essa vengono dichiarati una variabile e un metodo che stamperà la variabile stessa. Crea una classe di prova che istanzi almeno 2 oggetti della classe NumeroIntero e cambi il valore delle relative variabili verificando le assegnazioni, utilizzando il metodo stampaNumero().
Aggiungi un costruttore alla classe NumeroIntero che inizializzi l'attributo.

5 Definisci una classe Studente per rappresentare oggetti studente con il cognome, il nome, il codice fiscale, il numero di matricola e con opportuni metodi d'istanza tra cui un metodo del tipo String toString() per la sua descrizione.

LA SFIDA

LA CLASSE LAMPADINA
Definisci la classe Lampadina i cui oggetti rappresentano delle lampadine elettriche. Una lampadina può essere accesa, spenta o rotta, e mette a disposizione due sole operazioni: toString() che restituisce una stringa che indica lo stato corrente della lampadina e click() che ne cambia lo stato da accesa a spenta o da spenta a accesa o la rompe. Una lampadina si rompe dopo un certo numero di click deciso dal fabbricante.
Definisci gli attributi, i metodi e i costruttori che ritieni opportuni. Istanzia e usa la classe Lampadina in modo che ammetta un numero massimo di click deciso dall'utente e poi, in maniera iterativa, offra all'utente la possibilità di invocare una delle due funzionalità (visualizzando l'esito dell'operazione) o di terminare l'esecuzione.

LEZIONE 4

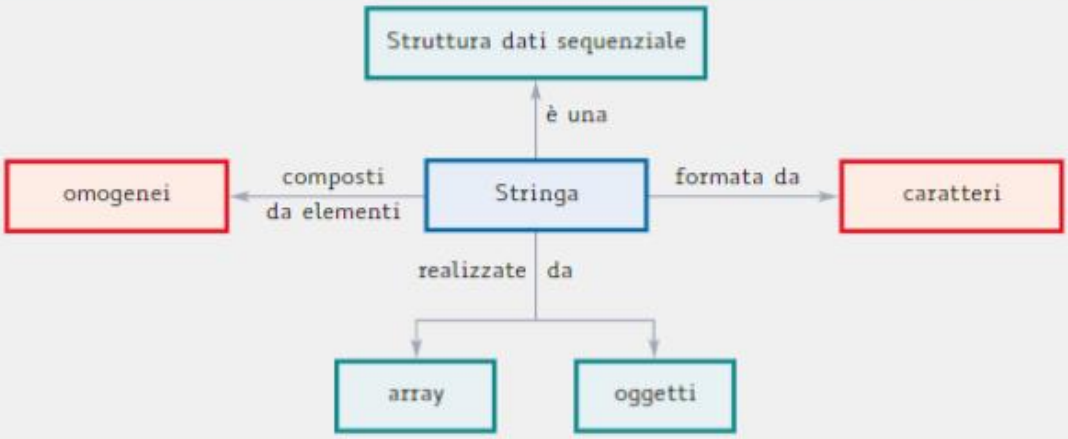
4 Un esempio di applicazione: la classe String

IN QUESTA LEZIONE IMPAREREMO...

- gli array di caratteri
- a operare con le stringhe
- a utilizzare gli oggetti String

MAPPA CONCETTUALE

didattica inclusiva



Introduzione alle stringhe

Si è detto che l'array è una struttura di tipo sequenziale in quanto rispecchia fedelmente l'organizzazione della memoria dell'elaboratore: infatti esso è costituito da un insieme contiguo di elementi dello stesso tipo, così come la memoria RAM è costituita da celle (elementi) fisicamente adiacenti. Un'altra struttura di dati sequenziale è la stringa: essa consiste in una sequenza finita di caratteri che generalmente viene racchiusa tra virgolette o apici (" ").
In questa trattazione le stringhe sono già state utilizzate molte volte, anche se non sono state definite esplicitamente: per esempio, tutte le operazioni di comunicazione con l'utente, eseguite con l'istruzione di output, sono state accompagnate da messaggi racchiusi tra virgolette che sono stringhe a tutti gli effetti.

ESEMPIO

La scritta "*inserisci un numero intero*" è una stringa, come è una stringa la singola parola "*spaghetti*" oppure la frase "*1.2.3... stella!*".

Le stringhe in Java

Nei linguaggi imperativi le stringhe sono generalmente realizzate mediante un array di caratteri; in Java, invece, sono oggetti della classe String, ma è comunque possibile utilizzare gli array per memorizzare le parole inserendo singolarmente un char per ogni posizione del vettore.

La classe String mette a disposizione molteplici metodi per analizzare e manipolare le stringhe: descriviamo i più utilizzati mediante alcuni esempi.

• Concatenazione di stringhe: operatore +

La concatenazione di due (o più) stringhe in una nuova stringa viene effettuata con il semplice operatore di somma oppure con l'apposito metodo concat(): il risultato è identico.

Codice	Output

`String s1,s2,s3, s4;`
`s1 = "Hello";`
`s2 = "world";`
`s3 = s1 + " " + s2;`
`s4 = s1.concat(s2);`
	`Hello`
	`world`
	`Hello world`
	`Helloworld`

• Confronto di stringhe: compareTo()

Le istanze di String possono essere confrontate con il metodo compareTo(): 0 se s1 è uguale a s2, <0 se s1<s2, >0 se s1>s2 sulla base dell'ordinamento alfabetico (ordinamento lessicografico, cioè quello dei dizionari).
Il risultato si ottiene dalla differenza dei valori del primo carattere differente:

```
(int)s1.charAt(i) – (int)s2.charAt(i)
```

Codice	Output

`s1 = "Anto";`
`s2 = "Antonio";`
`s3 = "Antoniol";s4 = "Pino";`
`System.out.println(s1.compareTo(s2));`
`System.out.println(s2.compareTo(s3));`
`System.out.println(s3.compareTo(s2));`
`System.out.println(s4.compareTo("Pino"));`
	`-3`
	`-1`
	`1`
	`0`

• Confronto tra stringhe: equals()

Una seconda possibilità di comparazione tra stringhe ci viene data dal metodo equals() che ci permette di effettuare anche il confronto tra due sottostringhe.

Codice	Output

`//posiz:0123456789*123456789*12345`
`s1 = "ciao mondo";`
`s2 = "mondo";`
`System.out.println(s1.equals(s2));`
`System.out.println(s2.equals(s1.substring(5)));`
	`false`
	`true`

• Lunghezza di una stringa: length()

Il metodo length() è particolarmente utile in quanto restituisce il numero di caratteri presenti nella stringa.

Codice	Output

`s1 = "ciao";`
`s2 = "ciao a tutti";`
`System.out.println(s1.length());`
`System.out.println(s2.length());`
	4
	12

• Sostituzione/rimozione di caratteri: replace

Per modificare il contenuto di una stringa si può utilizzare la funzione che permette di sostituire/rimuovere uno o più caratteri da una stringa.

Codice	Output

`s1 = "ciao mondo";`
`s2 = "buongiorno buonasera";`
`System.out.println(s1.replace("a", "x"));` //sostituisce
`System.out.println(s1.replace("a", "afa"));` //sostituisce
`System.out.println(s1.replace("o", "="));` //sostituisce
`System.out.println(s1.replace("o", ""));` //rimuove
`System.out.println(s2.replace("b", "B"));` //sostituisce
	cixo mondo
	ciafao mondo
	cia= m=nd=
	cia mnd
	Buongiorno Buonasera

• Accesso a una sottostringa: substring()

È possibile estrarre una parte di una stringa (sottostringa) e memorizzarla in una nuova variabile con il metodo substring(<pos_iniziale>) oppure substring(<pos_iniziale>,<pos_finale>).

Codice	Output

`//posiz:0123456789*123456789*12345`
`s1 = "ciao mondo";`
`s2 = "buongiorno buonasera";`
`s3 = "qui quo qua";`
`System.out.println(s1.substring(5));`
`System.out.println(s1.substring(0,5));`
`System.out.println(s2.substring(11));`
`System.out.println(s2.substring(11,16));`
`System.out.println(s3.substring(8));`
`System.out.println(s3.substring(0,3));`
	mondo
	ciao
	buonasera
	buona
	qua
	qui

• Ricerca di caratteri: indexOf()

È possibile trovare la posizione di un carattere (o di una sottostringa) all'interno di una stringa con il metodo indexOf().

Codice	Output

`s1 = "ciao mondo";`
`s2 = "mondo";`
`System.out.println(s1.indexOf(s2) );`
`System.out.println(s1.indexOf("do"));`
`System.out.println(s1.indexOf("zz"));`
	5
	8
	-1

• Conversione da stringa a numeri: Integer.parseInt() e Double.parseDouble()

Può essere utile trasformate una stringa in un numero intero oppure in un float: di seguito sono riportati due esempi di conversione.

Codice	Output

`s1 = "123";`
`s2 = "45.6";`
`s3 = "7.890";`
`System.out.println(Integer.parseInt(s1));`
`System.out.println(Double.parseDouble(s1));`
`System.out.println(Double.parseDouble(s2));`
`System.out.println(Double.parseDouble(s3));`
	123
	123.0
	45.6
	7.89

• Conversione da numeri a stringa: toString()

Questa funzione permette di effettuare la conversione da qualunque tipo di numero a stringa.

Codice	Output

`byte x = 12;`
`int y = 123;`
`double z = 5.12;`
`System.out.println(Integer.toString(x));`
`System.out.println(Integer.toString(y));`
`System.out.println(Double.toString(x));`
`System.out.println(Double.toString(y));`
`System.out.println(Double.toString(z));`
	12
	123
	12.0
	123.0
	5.12

L'elenco completo dei metodi della classe String è disponibile al seguente indirizzo:
http://www.falkhausen.de/Java-8/java.lang/String.html.

Vediamo un semplice esempio di applicazione delle funzioni appena descritte.

✔ PROBLEMA SVOLTO PASSO PASSO

✔ Il problema

Scrivi un programma che legge una parola sotto forma di stringa, la "converte" in un array di caratteri e successivamente la trasforma in minuscolo.

✔ L'analisi e la strategia risolutiva

Definiamo le sue variabili, una del tipo string e una come array di caratteri; leggiamo quindi la parola in ingresso e, tramite un ciclo a conteggio, estraiamo da essa un carattere alla volta inserendolo nel vettore utilizzando il metodo charAt(); la dimensione della stringa viene restituita dal metodo length() e sarà utilizzato come estremo di conteggio.

Successivamente passiamo a trasformare in minuscolo carattere per carattere tramite il metodo toLowerCase().

✔ La codifica in linguaggio di programmazione

Linguaggio Java

```
import java.util.Scanner;
public class Minuscolo {
  static final int TANTI = 30;
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    String parola;                                  // stringa come oggetto
    char[] minuscolo = new char[TANTI];             // stringa come array
    System.out.print("\nInserisci una parola  : ");
    parola = in.next();                             // legge una stringa
    // trasforma la stringa letta in un array di caratteri
    for (int x = 0; x < parola.length(); x++)
      minuscolo[x] = parola.charAt(x);              // primo carattere letto

    // trasforma la parola in minuscolo
    for (int x = 0; x < parola.length(); x++)
      minuscolo[x] = Character.toLowerCase(minuscolo[x]); // in minuscolo

    System.out.print("La parola in minuscolo (array)  : ");
    // visualizza la parola carattere per carattere
    for (int x = 0; x < minuscolo.length; x++)
      System.out.print(minuscolo[x]);

    // trasformo da array di caratterI a stringa
    String parolaMinu = new String(minuscolo);
    System.out.print("\nLa parola in minuscolo (oggetto): ");
    System.out.println(parolaMinu);
  }
}
```

✔ L'esecuzione del programma

```
Inserisci una parola  : AustraLopiTeco
La parola in minuscolo (array)  : australopiteco
La parola in minuscolo (oggetto): australopiteco
```

Il codice sorgente lo trovi nel file Minuscolo.java.

METTITI ALLA PROVA

→ • Definizione di stringhe • Elaborazione di stringhe

Scrivi un programma che legga una frase introdotta da tastiera. La frase è terminata dall'introduzione del carattere di invio. La frase contiene sia caratteri maiuscoli sia caratteri minuscoli, e complessivamente al più 100 caratteri.
Il programma dovrà stampare su schermo le seguenti informazioni:
– per ognuna delle lettere dell'alfabeto, il numero di volte che la lettera compare nella stringa;
– il numero di consonanti presenti nella stringa;
– il numero di vocali presenti nella stringa.

VERIFICA... le competenze

PROBLEMI

Progetta e realizza completamente in linguaggio di programmazione il codice che risolva i problemi proposti.

1. Scrivi un metodo che ritorna vero (0) se due stringhe sono identiche oppure un numero che indica quanti caratteri consecutivi sono uguali.
2. Scrivi un metodo che ritorna vero (0) se una stringa contiene almeno una vocale maiuscola.
3. Scrivi un metodo che ritorna vero (0) se una stringa contiene almeno tre consonati consecutive.
4. Scrivi un metodo che ritorna vero (0) se una stringa contiene almeno una doppia.
5. Scrivi un metodo che ritorna vero (0) se una stringa non contiene nessun segno di separazione, altrimenti visualizza la loro frequenza. Assumi che i separatori di parole siano lo spazio (blank) e gli usuali segni di punteggiatura (punto, virgola, punto e virgola, punto esclamativo, punto interrogativo, ...).
6. Scrivi un metodo che, presa in input una stringa strText e altre due stringhe strOld e strNew, sostituisca la prima occorrenza di strOld in strText con strNew.
7. Scrivi un metodo che, presa in input una stringa strText e altre due stringhe strOld e strNew, sostituisca tutte le occorrenze di strOld in strText con strNew.
8. Scrivi un programma che, partendo dalla stringa "oggi piove", stampi la stringa in maiuscolo, aggiunga alla stringa la frase "domani sarà bel tempo" e stampi la nuova frase, restituisca in output la lunghezza della stringa e, infine, trasformi tutte le vocali in 'i', visualizzando il risultato.
9. Scrivi un programma che legga una frase introdotta da tastiera che contenga sia caratteri maiuscoli sia caratteri minuscoli, e complessivamente al più 100 caratteri; successivamente il programma deve costruire una nuova frase tale che ogni lettera vocale presente nella frase di partenza sia seguita dalla lettera 'f' (se la vocale e minuscola) o dalla lettera 'F' (se la vocale e maiuscola).
   Per esempio, la frase VacAnze di NaTAle diventa VafcAFnzef dif NafTAFlef.
10. Scrivi un programma che legga una frase introdotta da tastiera che contenga complessivamente al più 100 caratteri. Il programma deve costruire una nuova frase in cui tutte le occorrenze del carattere '.' siano sostituite con il carattere di ritorno di linea '\n'.
11. Scrivi un programma che legga una frase introdotta da tastiera che contenga sia caratteri maiuscoli sia caratteri minuscoli, e complessivamente al più 100 caratteri. Il programma deve costruire una nuova frase in cui il primo carattere di ciascuna parola nella frase di partenza è stato reso maiuscolo e tutti gli altri caratteri devono essere resi minuscoli.
    Per esempio, la frase cHe bElLA glOrnaTa diventa Che Bella Giornata.

LA SFIDA

PUNTI NELLO SPAZIO

Un possibile formato per la rappresentazione in forma di testo di una sequenza di punti (in uno spazio tridimensionale) consiste nell'elencarne le terne di coordinate separate da uno (o più spazi). Le singole coordinate (x,y,z) sono invece separate da una virgola (carattere ',').
Per esempio, per la sequenza di tre punti P1(0.12,1.3,2) P2(1.2.,1.3,4) e P3(1.8.,2.5,1.6) la rappresentazione nel formato sopra definito sarebbe 0.12,1.3,2 1.2,1.3,4 1.8,2.5,1.6.
Osserva come dopo la virgola non ci sia alcuno spazio. Implementa un metodo parsePoints() che, preso in input un testo text rappresentato da un array di caratteri zero-terminato, restituisca nell'argomento di output points la sequenza di punti memorizzati nel testo.

LEZIONE 5

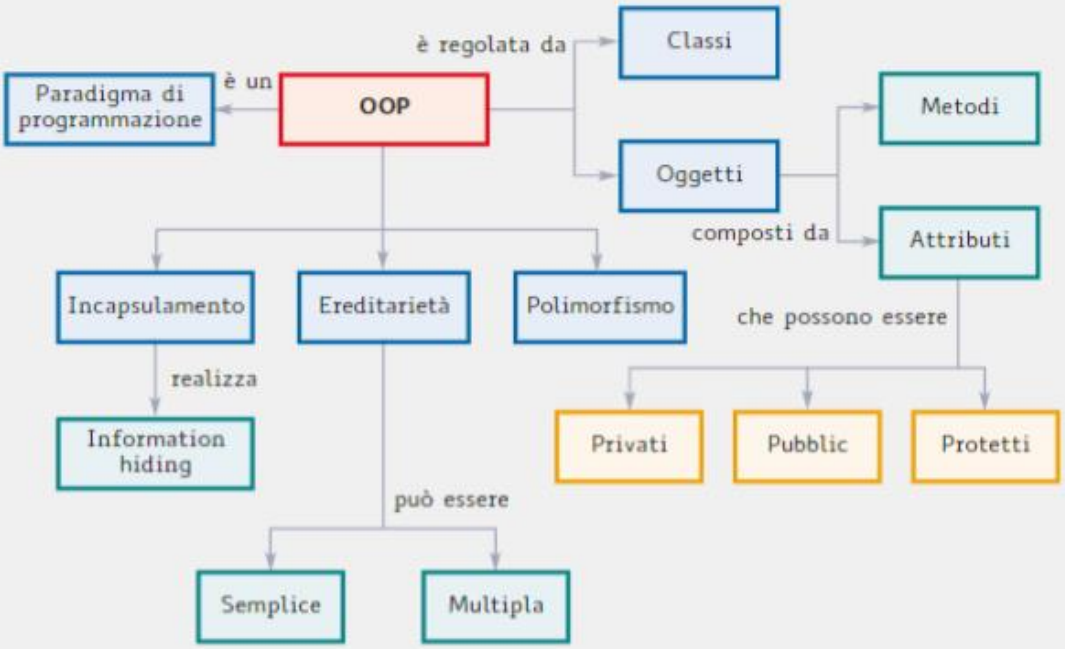
5 Ereditarietà, polimorfismo e relazioni tra le classi

IN QUESTA LEZIONE IMPAREREMO...

- a riconoscere e definire la gerarchia delle classi
- a individuare la specializzazione e la generalizzazione di una classe
- a riconoscere se una classe appartiene a una gerarchia

MAPPA CONCETTUALE

didattica inclusiva



Generalizzazione ed ereditarietà

L'ereditarietà è uno dei principi basilari della programmazione orientata agli oggetti e forse più degli altri caratterizza la OOP, costituendo lo strumento "per eccellenza" per raggiungere l'obiettivo di riusabilità del codice.
Il termine stesso richiama il concetto di "tramandare" qualcosa tra generazioni, cioè di trasferire ad altri beni o conoscenze: infatti, il concetto alla base di questo paradigma è utilizzare una

classe già "bella e pronta", che rappresenta un concetto più generale quando si deve realizzare una classe specifica.

L'ereditarietà è quindi un processo grazie al quale una classe acquisisce un bagaglio di proprietà da una classe più generica per ampliarla e specializzarla, in modo da permettere di descrivere una classificazione gerarchica di oggetti senza ridefinire ogni volta le singole caratteristiche comuni a ogni classe.

ESEMPIO

Supponiamo di avere a disposizione una generica classe Elettrodomestico e di dover modellare la classe Lavatrice: sicuramente nella modellizzazione della classe Lavatrice devono essere presenti tutti gli attributi e i metodi della classe Elettrodomestico, in quanto la lavatrice è un elettrodomestico; inoltre, ci saranno nuovi attributi e metodi specifici della lavatrice, come "il caricamento", "i kg di capacità", "i programmi di lavaggio" ecc.

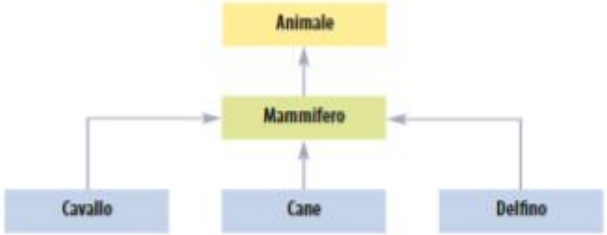
Lo stesso discorso vale nel caso in cui si voglia descrivere una Radio o un Televisore: sono entrambi elettrodomestici e in essi è compreso un sottoinsieme di attributi e metodi identici.
L'ereditarietà permette di non dover riscrivere all'interno della classe Lavatrice le caratteristiche della classe Elettrodomestico, ma di dire semplicemente che questa nuova classe è un Elettrodomestico con in più un nuovo set di attributi e metodi: l'enorme vantaggio è quello di utilizzare un "mattone" già solido (testato e collaudato!) e di doverlo solamente specializzare.

Gerarchie di classi

Questo discorso potrebbe essere esteso anche a più livelli di eredità, come all'interno di un albero gerarchico (nonno-padre-figlio).

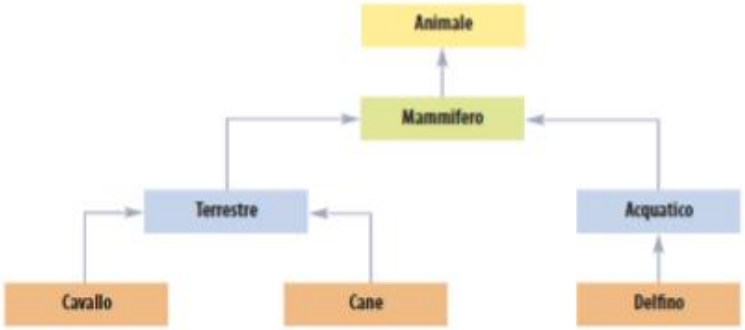
ESEMPIO

Supponiamo di avere a disposizione una generica classe Animale e di dover di descrivere tre nuove classi: la classe Cane, la classe Cavallo e la classe Delfino.
Tutti questi animali hanno una particolare caratteristica che li accomuna: sono tutti e tre mammiferi, quindi in tutte le tre classi dovranno essere presenti le caratteristiche dei mammiferi. Sarebbe opportuno quindi definire una classe intermedia Mammifero, con al suo interno attributi e metodi specifici.



Dunque il cane è un animale ma è anche un mammifero, o meglio un mammifero è un animale e un cane è un mammifero, e lo stesso discorso vale per il cavallo e il delfino: "il cavallo eredita dal mammifero che eredita dall'animale".

Questa classificazione sarebbe ulteriormente migliorabile introducendo una classe intermedia Terrestri, oppure una classe Quadrupedi, in modo da raggruppare ulteriormente le caratteristiche di cani e cavalli oppure Acquatico, per classificare le creature che vivono nelle acque.



In un secondo momento, in caso di necessità, è possibile ereditare dalla classe Animale altre classi, per esempio Rettile o Uccello, che andrebbero a completare la struttura gerarchica.
Non è necessario individuare la completa "organizzazione del regno animale" in una sola volta nel caso in cui serva la sola classe Cane: basta definire solamente il suo ramo gerarchico a partire dalla radice.

La classificazione gerarchica è quindi un elemento fondamentale nella costruzione delle classi, perché ci permette di definire livelli di dettaglio e di aggregazione utili per il riutilizzo da parte di altre classi: si sarebbero potute derivare le tre classi Cane, Cavallo e Delfino direttamente dalla classe Animali, con ovvia duplicazione e riscrittura inutile di metodi e attributi.

Nell'esempio Animale → Mammifero → Terrestre → Cane è molto facile individuare la classificazione gerarchica; è quindi utile generalizzare modelli per poi riutilizzarli: a volte però questa "catena" non è di così immediata individuazione e può portare a spiacevoli risultati.

Spesso infatti ci si basa solamente sull'individuazione di elementi (attributi) uguali e caratteristiche simili e... si commette un errore, come nel seguente esempio.

ESEMPIO

Supponiamo di avere la classe Triangolo con il diagramma delle classi (vedi figura di sinistra) e di aver creato la classe Quadrato: dal confronto dei diagrammi UML si vede come la classe Triangolo sia compresa interamente nel Quadrato, quindi si potrebbe pensare di ereditare dalla classe Quadrato la classe Triangolo, aggiungendo semplicemente un attributo (lunghezzaLato4).

Naturalmente si commetterebbe un errore, dato che un quadrato non è un triangolo.

Definizioni

Alla luce di quanto finora affermato, forniamo di seguito una prima definizione di ereditarietà.

Con il termine ereditarietà si indica un meccanismo per definire nuove classi in termini di classi esistenti già definite con un più alto livello di astrazione, dalle quali ereditare attributi e comportamenti.
L'ereditarietà permette di raggruppare classi correlate in modo che possano essere:
- gestite collettivamente;
- riusate.

Le classi vengono quindi organizzate secondo una classificazione gerarchica e viene determinato un concetto di antenato e discendente.

La classe antenato di una nuova classe (il padre) prende il nome di superclasse e la classe discendente di una classe (il figlio) esistente prende il nome di sottoclasse.
Ogni sottoclasse eredita le caratteristiche della sua superclasse e ogni istanza di una sottoclasse è (IS-A) un'istanza della sua superclasse. Vale inoltre la proprietà transitiva: ogni istanza di una sottoclasse è un'istanza di tutte le sue classi antenato.

La relazione IS-A è la relazione tra una classe (superclasse o classe base) e una o più versioni specializzate (sottoclassi, o classi derivate).
Prende il nome di regola di appartenenza o IS-A relationship ("relazione è-un").

La verifica della relazione di appartenenza avviene ponendo due semplici domande in merito a un oggetto per stabilire se esiste una relazione gerarchica:
- è un;
- è un tipo di.

ESEMPIO

Riprendendo il precedente esempio della classe Lavatrice procediamo alla seguente verifica:

"La lavatrice è un elettrodomestico?"	Risposta: Sì
"La lavatrice è un tipo di elettrodomestico?"	Risposta: Sì

quindi:
- la classe Lavatrice è una sottoclasse della classe Elettrodomestico;
- la classe Elettrodomestico è una superclasse della classe Lavatrice.

Generalizzazione e specializzazione

I processi che portano all'implementazione dell'ereditarietà sono la generalizzazione e la specializzazione.
Siamo in presenza di una specializzazione (o classificazione gerarchica) quando, partendo da una classe generica, si definiscono una o più sottoclassi allo scopo di ottenere oggetti più specia-

specifiche che li distinguono.

L'ereditarietà viene applicata seguendo una metodologia top-down.

ESEMPIO

Animali → Vertebrati → Mammiferi → Equini → Cavallo

Siamo invece in presenza di una generalizzazione se, a partire da un certo numero di classi, si definisce una superclasse che ne raccolga le caratteristiche comuni, individuando quindi una classe più generica.
L'ereditarietà viene applicata seguendo una metodologia bottom-up.

ESEMPIO

Formula1 → Automobile → MezzoAMotore → MezzoDiTrasporto

Funzioni della sottoclasse

a) La classe derivata (sottoclasse) può:
- aggiungere nuovi attributi e nuovi metodi a quelli della superclasse;
- ridefinire (sovrascrivere) alcuni dei metodi ereditati dalla superclasse, riscrivendone il codice utilizzando lo stesso nome e firma (override);
- ridefinire metodi della classe padre ma con firme diverse (overload);
- restringere in qualche modo la visibilità di una variabile o un metodo ereditato dalla superclasse.

b) La classe derivata (sottoclasse) non può:
- eliminare attributi o metodi della superclasse (comportamento monotono crescente: il numero di attributi aumenta sempre).

La ridefinizione di metodi nella sottoclasse è tuttavia possibile grazie al concetto di polimorfismo, che permette di avere comportamenti differenti richiamando la stessa funzione su due oggetti nella stessa gerarchia.

ESEMPIO

Se da una classe Stampanti deriviamo le classi InkJet e Plotter, sicuramente dovremo riscrivere il metodo print() che effettuerà operazioni diverse a seconda che venga richiamato su una documento Word oppure su un disegno di AutoCAD.

La stesso criterio viene ripetuto per le classi "nipoti", cioè per le classi "figlie di figlie" in una gerarchia con diversi livelli gerarchici di ereditarietà.

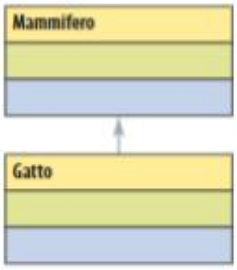
ESEMPIO

Un esempio tipico lo riscontriamo nel metodo toString() che inseriremo in ogni classe e che permetterà di visualizzare gli attributi significativi dell'oggetto: essendo presente in ogni classe della gerarchia esso verrà sovrascritto più volte.

Ereditarietà: modalità operative

Vediamo adesso come realizzare l'ereditarietà e come descriverla utilizzando i diagrammi UML.

La realizzazione di una classificazione gerarchica può essere effettuata mediante le seguenti due semplici regole operative:
- definire dapprima le classi che modellano i concetti più generali, quindi trattare i casi specifici (uses cases) come specializzazione;
- creare classi che modellano alcuni concetti, verificare che siano tra loro collegate ed estrarre le caratteristiche comuni in una o più superclassi.



Superclasse

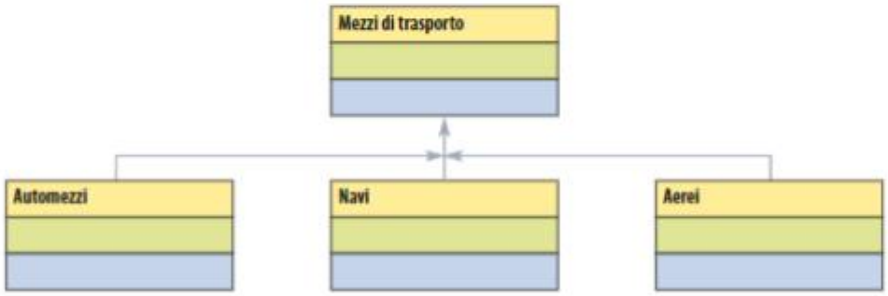
Eredità

Sottoclasse

La rappresentazione grafica UML è riportata nella figura a lato.

In un diagramma delle classi l'ereditarietà viene rappresentata mediante una freccia con la punta a triangolo vuoto diretta verso la superclasse.

Nel caso di più sottoclassi che hanno in comune la stessa superclasse, si utilizza la seguente rappresentazione grafica.



Riportiamo di seguito la definizione di una relazione gerarchica scritta con la sintassi dei linguaggi di programmazione da noi utilizzati.

Mammifero → Gatto

```
public class Mammifero {
  // attibuti
  private int m1;
  private String m2;
  // metodi
  private int metodo1();
}
public class Gatto extends Mammifero {
  // attibuti
  private int g1;
  private String g2;
  // metodi
  private int metodo1();
}
```

Genitore → Figlio

```
class Genitore{
  private int g1;
  public int g2;
}

class Figlio extends Genitore{
  private int f1;
  public int f2;
}
```

Visibilità degli attributi

Nel codice riportato a pagina precedente abbiamo dichiarato nella classe padre un attributo pubblico e uno privato: possiamo verificare come nella classe ereditata tali attributi abbiano il seguente comportamento:
- gli attributi privati della classe padre non sono accessibili dalla classe figlio;
- gli attributi pubblici della classe padre sono accessibili dalla classe figlio ma in Java rimangono pubblici.

Una classe figlio quindi ha accesso solo ai metodi e attributi della classe padre che non sono di tipo privato: `Figlio` eredita solo gli attributi e i metodi della classe `Padre` consentiti dagli specificatori di visibilità, cioè quelli che non sono stati dichiarati private, e cioè:

Linguaggio Java

```
public void esempioVisibilita()
{
  f1 = 1;
  f2 = 2;
  g2 = 10;
  g1 = 20;  // non si puo' fare
            // perche' e' privata nel padre
            // e non accessibile dal figlio
}
```

Sviluppiamo ulteriormente l'esempio creando due istanze delle classi prima definite per evidenziare la visibilità degli attributi.

Linguaggio Java

```
public class ProvaGenitore
{
  public static void main()throws Exception
  {
    Genitore adamo = new Genitore();
    Figlio caino = new Figlio();
    // attributi della classe padre
    adamo.g1 = 10;   // non corretta -> privato
    adamo.g2 = 20;   // corretta,  e' pubblico
    // attributi della classe figlio
    caino.f1 = 1;    // non corretta -> privato
    caino.f2 = 2;    // corretta, e' pubblico
    caino.g1 = 30;   // non corretta -> privato nel padre
    caino.g2 = 40;   // corretta, era pubblico
  }
}
```

Per rendere visibili gli attributi tra le classi ereditate si può utilizzare il qualificatore protected: si comporta come una "via di mezzo" tra privato e pubblico, in quanto gli attributi protected sono visti solamente nella classe dove sono definiti e anche nei metodi delle classi figlie, ma mai in altri punti del programma.

Quindi, dichiarare un attributo protected nella superclasse equivale a:
- dichiarare l'attributo public per le sottoclassi;
- dichiarare l'attributo private per tutte le classi esterne.

La tabella riporta un riepilogo della visibilità degli attributi in base alle classi e ai qualificatori.

ACCESSIBILE DA	PUBLIC	PROTECTED	PRIVATE
Stessa classe	SÌ	SÌ	SÌ
Sottoclasse	SÌ	SÌ	NO
Classe generica (non sottoclasse)	SÌ	NO	NO

Nella OOP i termini modificatore, qualificatore e specificatore sono spesso utilizzati come sinonimi e indicano il "livello" di visibilità di metodi e attributi.

Vediamo un esempio dove deriviamo una classe specializzandola a partire da un classe più generica.

✔ PROBLEMA SVOLTO PASSO PASSO

✔ Il problema

Scriviamo una classe ContatoreDoppio a partire dalla classe Contatore avente il seguente prototipo.

✔ L'analisi e metodologia risolutiva

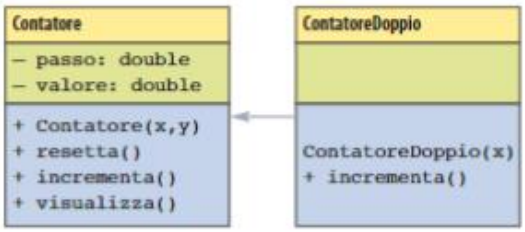
Scriviamo ora la classe ContatoreDoppio che eredita dalla classe Contatore.
Per semplicità, inseriamo un solo costruttore e un solo metodo:
- il costruttore non fa altro che invocare il costruttore della classe padre;
- il metodo, che ha lo stesso nome di quello presente nella classe padre, lo invoca due volte, in modo da effettuare un doppio incremento.

Linguaggio Java

```
class Contatore {
  private int valore;
  private int passo;

  public Contatore()..          // costruttori
  public Contatore(int x )..
  public Contatore(int val, int passo)..
  public void resetta()..       // modificatori
  public void incrementa()..
  public int getValore()..      // getter/setter
  public void setValore(int x)..
  public int getPasso()..
  public void setPasso(int x)..
}
```

✔ Il diagramma UML



✔ La codifica in linguaggio di programmazione

La classe

```
class ContatoreDoppio extends Contatore {
  // invocazione costruttore classe base
  ContatoreDoppio(int n){
    super (n);
  }
  // ridefinizione del metodo incrementa
  public void incrementa(){
    super.incrementa();
    super.incrementa();
  }
}
```

La classe di prova

```
public class ProvaContatore{
  public static void main(String[] args)throws Exception{
    Contatore c1 = new Contatore(10);
    ContatoreDoppio c2 = new ContatoreDoppio(10);
    System.out.println("Valore iniz. contatori: ");
    System.out.println("- singolo = " + c1.getValore());
    System.out.println("- doppio  = " + c2.getValore());
    // incremento dei due contatori
    c1.incrementa();
    c2.incrementa();
    System.out.println("Valore finale contatori: ");
    System.out.println("- singolo = " + c1.getValore());
    System.out.println("- doppio  = " + c2.getValore());
  }
}
```

⚠ Per accedere all'attributo valore dal main() è necessario invocare il meccanismo getter/setter, dato che nella classe padre questo attributo è definito privato.

✔ L'esecuzione del programma

Pur avendo chiamato lo "stesso metodo", otteniamo due risultati diversi: questa è una delle potenzialità dell'overriding.

Il codice completo di questo programma lo puoi trovare nel file ContatoreDoppio.java.

```
Valore iniz. contatori:
- singolo = 10
- doppio  = 10
Valore finale contatori:
- singolo = 11
- doppio  = 12
```

Indicazione esplicita @Override

Nelle gerarchie è possibile indicare in modo esplicito un metodo che viene sovrascritto: viene aggiunta la parola riservata @Override prima della sua scrittura.
Consigliamo di utilizzare sempre questa notazione, in quanto, con "poco sforzo", ci permette di avere due importanti vantaggi:

1 ci assicura che si sta effettivamente "scavalcando" un metodo e se si commette un errore, sia di digitazione del nome sia nella corrispondenza corretta dei parametri, si viene immediatamente avvisati;
2 ci semplifica la comprensione del codice perché risulta evidenziata la situazione di sovrascrittura.

L'esempio precedente verrebbe così modificato:

```
class ContatoreDoppio extends Contatore{
  // invocazione costruttore classe base
  ContatoreDoppio(int n){
    super(n);
  }
  // ridefinizione del metodo incrementa
  @Override
  public void incrementa(){
    super.incrementa();
    super.incrementa();
  }
}
```

Realizzazione di una gerarchia

Abbiamo visto che la realizzazione di una classificazione gerarchica può essere effettuata modellando dapprima le classi più generali per poi procedere nella specializzazione, prevedendo o concretizzando nuove funzionalità.

Si definisce pertanto gerarchia di specializzazione il legame logico che esiste tra classi e sottoclassi.

La regola pratica da seguire per la costruzione di un albero di ereditarietà dice che se un oggetto A è anche un oggetto B, allora la classe A eredita dalla classe B.

Vediamo un esempio con la definizione dei metodi e degli attributi.

✔ PROBLEMA SVOLTO PASSO PASSO

✔ Il problema

Data la classe Punto, che descrive i punti nel piano cartesiano, si modelli la classe Punto3D per rappresentare i punti nello spazio.

✔ L'analisi e metodologia risolutiva

Ai fini della nostra trattazione è sufficiente riportare solo parti delle caratteristiche della classe Punto, e cioè:

- come attributi:
  - un identificatore, che è un carattere che individua il punto (A, B, C ...)
  - la sua ascissa (coordinata X)
  - la sua ordinata (coordinata Y)
- come metodi:
  - un solo metodo, che calcola la distanza dall'origine mediante la formula:

$$d = \sqrt{x^2 + y^2}$$

Anche per la classe Punto3D teniamo in considerazione solo le corrispondenti caratteristiche della classe Punto:

- come attributi
  - tutti gli attributi delle classe Punto
  - un attributo aggiuntivo: la coordinata Z
- come metodi:
  - il metodo della classe Punto
  - propri metodi aggiuntivi (proiezioni)

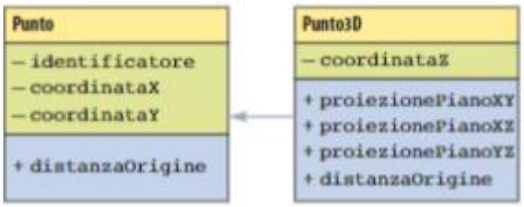
Il metodo distanzaOrigine() ereditato dalla superclasse non può tuttavia andare bene per la sotto-classe poiché la formula di calcolo è diversa, dato che comprende tutte le coordinate del punto: in questo caso, la formula corretta è la seguente:

$$d = \sqrt{x^2 + y^2 + z^2}$$

Si tratta quindi di sovrascrivere un metodo già esistente, che svolge la medesima operazione di quello che si desidera implementare, ma che deve essere specializzato per la nuova classe.

✔ Il diagramma UML

Una prima formulazione dei diagrammi UML delle due classi è la seguente.



✔ Verifica dei risultati

> ⚠ Quello che abbiamo appena fatto "sembra" essere corretto: tuttavia, pur avendo applicato il meccanismo dell'ereditarietà, non abbiamo verificato la regola dell'astrazione, che deve soddisfare la domanda di appartenenza (relazione IS-A).

Violando la regola dell'astrazione abbiamo potuto validare l'ereditarietà: ci siamo infatti chiesti "Un punto a tre dimensioni è un punto?" e abbiamo dato risposta affermativa, ma la classe Punto non è completamente astratta: avrebbe dovuto chiamarsi Punto2D, dato che non rappresenta il generico elemento geometrico "punto" ma un punto nel piano, quindi a due dimensioni.

Abbiamo dunque interpretato erroneamente l'identificatore e la nostra domanda di appartenenza avrebbe dovuto essere: "Un punto a tre dimensioni è un punto a due dimensioni?"; in questo caso, la risposta sarebbe stata negativa!

L'errore, o meglio, gli errori, sono da imputare a due cause:
– durante la modellizzazione non si è proceduto correttamente all'astrazione;
– durante l'implementazione non è stata analizzata nel dettaglio la relazione IS-A.

Nonostante la semplicità dell'esercizio sono stati commessi due errori: se apparentemente "tutto funziona lo stesso", sicuramente in un'applicazione che subisce molti incrementi gerarchici il "violare le regole" porta a conseguenze negative e in queste situazioni la "cosa migliore da farsi" è quella di "rianalizzare la situazione" in modo da riorganizzare la strutturazione gerarchica migliorando il processo di astrazione.

Vediamo allora di "ricostruire" la relazione gerarchica corretta per il nostro esempio: partiamo dal concetto geometrico di punto, cioè di una entità geometrica senza area identificata mediante una lettera maiuscola dell'alfabeto, che noi indicheremo come attributo nome.
Limitiamoci a considerare lo spazio a tre dimensioni: un punto nello spazio viene individuato mediante le sue tre coordinate cartesiane (oppure dalle tre coordinate polari più la sua distanza dall'origine), mentre un punto nel piano ha due sole coordinate.
Quindi domandiamoci: "Un punto nello spazio è anche un punto nel piano?"; oppure: "Un punto nel piano è anche un punto nello spazio?"
Utilizzando le coordinate del punto nello spazio possiamo definire i punti che appartengono a un piano, per esempio il piano xy avente come terza coordinata z = 0.

Quindi il piano è una parte dello spazio, cioè raggruppa un sottoinsieme di elementi dello spazio: in altre parole, "specializza" i punti che in comune hanno la particolarità di appartenere al medesimo piano avendo il valore di coordinata uguale per tutti.

Il diagramma UML riportato sotto è un esempio di rappresentazione di tale gerarchia.

Vediamo un secondo esempio.

✔ PROBLEMA SVOLTO PASSO PASSO

✔ Il problema

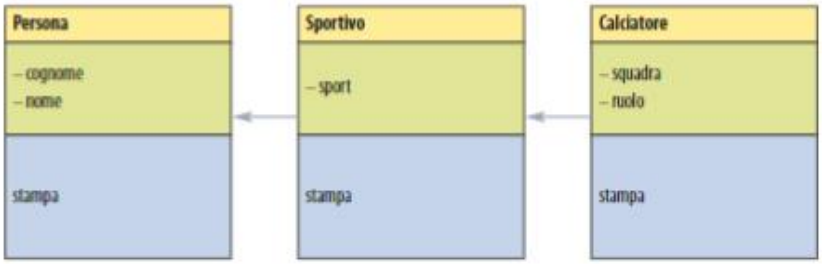
Progettiamo un insieme di classi in grado di descrivere le caratteristiche di alcune persone che possono essere sportivi e non e dove alcuni degli sportivi praticano il gioco del calcio.

✔ Il diagramma UML

Possiamo facilmente individuare la seguente gerarchia di classi:

Persona → Sportivo → Calciatore

Ci limiteremo a inserire gli attributi essenziali e come esempio di override sovrascriviamo in ciascuna il metodo stampa().



✔ L'analisi e metodologia risolutiva

Partiamo dalla classe Persona dove inseriamo solamente gli attributi ed i metodi principali, utili per questo esercizio. Il codice della classe Persona è il seguente.

Classe e costruttori

```
class Persona {
  private String cognome;
  private String nome;

  Persona() {
  }
  Persona(String co) {
    cognome = co;
  }
  Persona(String co, String no) {
    cognome = co;
    nome = no;
  }
}
```

Metodi getter/setter

```
void setNome(String no){
  nome = no;
}
String getNome(){
  return nome;
}
void setCognome(String co){
  cognome = co;
}
String getCognome(){
  return nome;
}
void stampa(){
  System.out.println("nome     : " + nome);
  System.out.println("cognome  : " + cognome);
}
```

Definiamo quindi la classe Sportivo partendo dalla classe Persona aggiungendo l'attributo sport dove andremo a inserire il nome dello sport che egli pratica: il costruttore richiama quello della classe padre per l'inizializzazione degli attributi nome e cognome.

Definiamo ora la classe Calciatore aggiungiamo due attributi, squadra e ruolo.

Classe Sportivo

```
class Sportivo extends Persona{
  String sport;
  Sportivo(){
  }
  Sportivo(String co, String no, String spo){
    super(co, no);
    sport = spo;
  }
  @Override public String toString(){
    String q;
    q = super.toString() + ("\nsport    : " + sport);
    return q;
  }
  @Override void stampa(){
    super.stampa();
    System.out.println("sport    : " + sport);
  }
}
```

Classe Calciatore

```
class Calciatore extends Sportivo{
  String squadra;
  String ruolo;
  Calciatore(){
  }
  Calciatore(String co, String no, String spo, String squ){
    super(co, no, spo);
    squadra = squ;
  }
  @Override public String toString(){
    String q;
    q = super.toString() + ("\nsquadra  : " + squadra);
    return q;
  }
  @Override void stampa(){
    super.stampa();
    System.out.println("squadra  : " + squadra);
  }
  void stampaSquadra(){
    System.out.println("squadra  : " + squadra);
  }
```

Osserviamo come nella classe Calciatore non sia necessario includere la classe Persona ma solo la classe Sportivo: infatti in essa è richiamata la classe "nonno", che è al vertice della gerarchia e, quindi, è visibile automaticamente.

✔ L'esecuzione del programma

Scriviamo adesso un programma di prova e mandiamolo in esecuzione ottenendo:

Linguaggio Java

```
public class ProvaPersone
{
  public static void main(String args[])
  {
    Persona p = new Persona("Sordi", "Alberto");
    System.out.println("\nPersona   \n--------" );
    p.stampa();

    Sportivo s = new Sportivo("Rossi", "Valentino", "motociclismo");
    System.out.println("\nSportivo   \n--------" );
    s.stampa();

    Calciatore c = new Calciatore("Maradona", "Diego", "calcio", "napoli");
    System.out.println("\nCalciatore \n--------" );
    c.stampa();
  }
}
```

```
Persona
-------
nome     : Sordi
cognome  : Alberto
Sportivo
-------
nome     : Valentino
cognome  : Rossi
sport    : motociclismo
Calciatore
-------
nome     : Maradona
cognome  : Diego
sport    : calcio
squadra  : napoli
```

Il codice completo di questo programma lo puoi trovare nel file ProvaPersone.java.

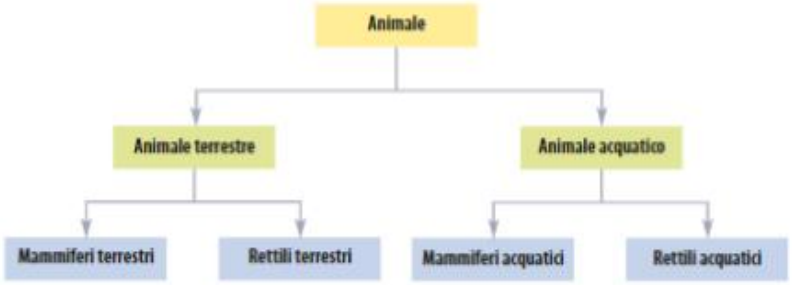
Ereditarietà multipla

Come è possibile avere più classi figlie da una classe padre è anche possibile che una classe figlia provenga da più "classi genitori" (ereditarietà multipla).
Riprendiamo l'esempio degli animali e consideriamo una classificazione secondo due criteri:
– ambiente in cui vivono (acquatici o terrestri);
– categoria zoologica (mammiferi, rettili ...).

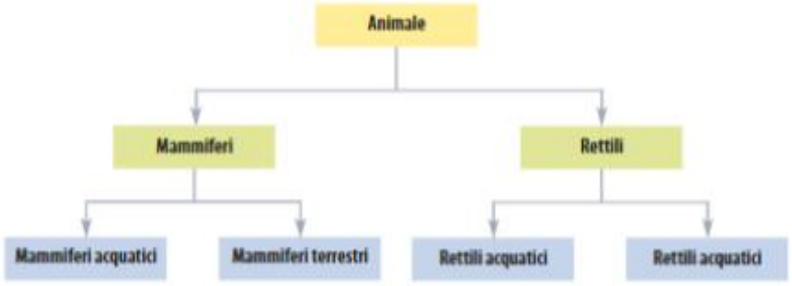
La rappresentazione di questa situazione in forma tabellare, è la seguente.

	ANIMALI ACQUATICI	ANIMALI TERRESTRI
Mammiferi		
Rettili		

Supponiamo di voler riportare questa classificazione in una tassonomia, definendo cioè una gerarchia di classi mediante l'ereditarietà. È necessario scegliere innanzitutto un criterio da usare per la creazione delle due sottoclassi "di primo livello": per esempio, possiamo definire dapprima gli animali in base all'ambiente in cui vivono e successivamente in base alla modalità di riproduzione.



Rispetto alla rappresentazione tabellare, però, qualcosa è andato perso: non esistono più infatti i concetti di "mammifero" e "rettile", che sono finiti dispersi nelle sottoclassi. Per recuperare il concetto di "mammifero", si può osservare che esistono "animali acquatici mammiferi" e "animali terrestri mammiferi", ognuno con proprie caratteristiche, ma non esiste una categoria di soli "mammiferi" con le peculiarità che la contraddistinguono.
Abbiamo quindi perso la possibilità di mantenere distinte le caratteristiche comuni a tutti (e soli) i mammiferi.
In alternativa, avremmo potuto stabilire che "mammiferi" e "rettili" fossero le classificazioni di primo livello: in questo caso, tuttavia, avremmo perso la rappresentazione pura di "animale acquatico" e "animale terrestre".



Attraverso il meccanismo di ereditarietà della rappresentazione tabellare precedente è infine possibile selezionare intere colonne della prima tabella esaminata, accedendo poi, tramite esse, alle singole celle ma non a intere righe. Questo avviene perché si è stabilita una gerarchia tra i due criteri di classificazione che in partenza erano ortogonali (sullo stesso piano) per poter utilizzare il meccanismo di ereditarietà singola per implementare la tassonomia.

L'ereditarietà singola è dunque uno strumento molto potente ma non adatto a modellare tutte le situazioni: sarebbe utile poter derivare le sottoclassi (mammiferi terrestri, rettili acquatici ecc.) a partire da più classi base, corrispondenti ciascuna all'applicazione di un criterio di classificazione.

In questo modo, i due criteri sarebbero sullo stesso piano e resterebbero ortogonali: quando è possibile derivare una classe facendole ereditare le proprietà di più classi base si parla di ereditarietà multipla.

L'ereditarietà multipla è un meccanismo molto potente ma molto discusso, perché introduce ambiguità non banali da risolvere.

Poiché la sottoclasse è un sottotipo di tutte le sue classi base (quindi un animale potrebbe essere contemporaneamente un mammifero terrestre e un rettile) e unisce in sé gli attributi e i metodi delle classi da cui eredita, ci si può porre alcune domande:
- che cosa succede se nelle classi base sono presenti campi dati o metodi omonimi? Se nella classe mammiferi terrestri è presente un metodo chiSei() presente anche nella classe dei rettili? Quale dei metodi viene ereditato: entrambi, uno solo? Da quale classe base?
- che cosa succede se una classe eredita direttamente o indirettamente (o per errore) più volte da una stessa classe base? Gli attributi vengono duplicati? I metodi devono essere considerati ambigui o no?

Se da un lato possiamo immaginare i vantaggi che si possono ottenere con l'ereditarietà multipla, e cioè la possibilità di comporre velocemente oggetti molto complessi, di aggregare funzionalità differenti in un'unica classe ottenendo soluzioni eleganti e di grande utilità, dall'altro dobbiamo tenere conto di una serie di svantaggi, come la notevole complicazione della codifica e la sua scarsa efficienza anche quando non viene usata; inoltre, si è in presenza di un rischio elevato di "name clash", cioè di "scontro tra metodi che hanno lo stesso nome", che deve essere gestito manualmente dal programmatore.

Per evitare le problematiche dovute al name clash, Java realizza l'ereditarietà multipla non in modo diretto ma mediante il meccanismo delle interfacce, che gli permette inoltre di effettuare un miglior controllo nella gerarchia delle classi.

METTITI ALLA PROVA

→ • Applicazione della ereditarietà • Visibilità dei metodi e attributi

Completa la classe Persona con i dati anagrafici e i rispettivi metodi getter/setter.
Da essa eredita la classe Professore, che la specializza aggiungendo la materia insegnata e la scuola di appartenenza.
Scrivi un programma di prova che definisce il tuo professore di teoria e il tuo ITP di informatica visualizzandone i dati.
Confronta la tua soluzione con quella presente nel file ProvaDocenti_solux.java.

VERIFICA... le competenze

ESERCIZI

AREA DIGITALE
Esercizi per l'approfondimento

Dopo aver individuato le classi e realizzato il diagramma UML, codificale in un linguaggio a piacere e scrivi un programma di prova che istanzia almeno due oggetti sui quali effettuare il collaudo di tutti i metodi definiti.

1 Classe Mucca
Implementa la classe Mucca come specializzazione di una catena di tre antenati.

2 Classe Cantante
Progetta la gerarchia di classi per definire cantanti, musicisti e compositori e i rispettivi metodi e attributi.

3 Classe Pianoforte
Progetta la gerarchia di classi per definire una classe Pianoforte a partire da una superclasse StrumentoMusicale.

4 Classe Cellulare
Progetta la gerarchia di classi per definire un telefono cellulare, metodi e attributi, a partire dalla classe Elettrodomestici.

5 Classe Automobile
Definisci una semplice gerarchia di classi per i veicoli a partire dalla classe MezzoDiTrasporto e modella la classe Automobile e CityCar.

6 Classe Geometria
Progetta un insieme di classi per la rappresentazione delle principali figure geometriche (Poligono, Ottagono, Rettangolo, Quadrato, Triangolo Isoscele). Di ognuna deve essere possibile calcolare l'area e il perimetro e confrontare questi valori con quelli di un'altra figura geometrica.

7 Classe Ascensore
Realizza un componente software che modelli un ascensore, che preveda come dati i piani di partenza, di arrivo e prenotati e permetta di eseguire le normali operazioni di un ascensore partendo da una classe Montacarichi.

8 Classe ComponenteElettronico
Realizza un componente software che modelli la gerarchia dei dispositivi elettronici discreti (bipoli, tripoli, resistori, condensatori, diodi ecc.).

9 Classe Bicicletta
Progetta la gerarchia di classi per definire una bicicletta, metodi e attributi, a partire dalla classe MezzoDiTrasporto.

10 Classe Autostrada
Progetta un programma che emetta lo scontrino al casello autostradale: gli automezzi sono classificati in categorie di pedaggio e i pedaggi sono differenziati nei giorni feriali e festivi.

LA SFIDA

ALBERO GERARCHICO SULLA SCACCHIERA
Nel gioco degli scacchi sono presenti sulla scacchiera 16 pezzi che hanno caratteristiche differenti. Realizza la gerarchia delle classi che permette di descrive le singole mosse effettuate e/o effettuabili durante una partita tra due giocatori.

VERIFICA... i saperi essenziali

SCELTA MULTIPLA

1 Indica le tre motivazioni della crisi del software. (*3 risposte*)
- **a** Crisi dimensionale
- **b** Crisi economica
- **c** Crisi gestionale
- **d** Crisi qualitativa
- **e** Crisi progettuale

2 L'acronimo OOD si ottiene da:
- **a** Object-Oriented Definition
- **b** Object-Oriented Design
- **c** Object-Oriented Directory
- **d** Object-Oriented Database3

3 I principi fondamentali della OOP sono: (*3 risposte*)
- **a** ereditarietà
- **b** semplicità
- **c** incapsulamento
- **d** ripetibilità
- **e** correttezza
- **f** polimorfismo
- **g** appartenenza

4 L'acronimo UML si ottiene da:
- **a** Uniform Modeling Language
- **b** Uniform Modify Language
- **c** Unified Modeling Language
- **d** Unified Modify Language

5 Un oggetto è:
- **a** una definizione di tipo
- **b** una istanza di una classe
- **c** un elemento di un metodo
- **d** un insieme di metodi e attributi
- **e** un'entità astratta
- **f** una definizione di tipo

6 Quale è il tipo di ritorno del costruttore?
- **a** Void
- **b** Nessuno
- **c** Object
- **d** Pointer

7 Quale è la visibilità del costruttore?
- **a** Public
- **b** Static
- **c** Protected
- **d** Private

8 Nell'overloading per i metodi abbiamo differenza di carattere:
- **a** formale
- **b** tipale
- **c** numerico
- **d** posizionale

9 La relazione IS-A è:
- **a** una relazione di appartenenza
- **b** una relazione di corrispondenza
- **c** una relazione di discendenza
- **d** una relazione di generalizzazione

10 Nell'override quali delle seguenti affermazione sono errate? (*2 risposte*)
- **a** Si può ridefinire metodi ereditati utilizzando lo stesso nome e firma
- **b** Si può ridefinire metodi ereditati utilizzando lo stesso nome ma firma differente
- **c** Non si può mai ridefinire metodi ereditati
- **d** Non si possono richiamare metodi della superclasse

VERIFICA... le competenze intermedie

AREA DIGITALE

Esercizi per l'approfondimento

ESERCIZI

Classi singole per applicazioni di matematica e geometria

1 Data la classe MyArray con una variabile d'istanza private mioArray[] si deve implementare un metodo cerca(n) che restituisce il valore true se e solo se il numero intero n appare all'interno di mioArray. Implementa inoltre un metodo single(), possibilmente ricorsivo, che restituisca un array ottenuto da mioArray eliminando tutti gli elementi duplicati. Per esempio:
– per l'array [4, 9, 7, 13, 9] il programma dovrà restituire l'array [4, 9, 7, 13] oppure l'array [9, 13, 7, 4];
– per l'array [7, 4, 13, 7, 13, 5] il programma dovrà restituire l'array [7, 4, 13, 5] oppure l'array [4, 7, 13, 5].

2 Data la classe MyArray con una variabile d'istanza private mioArray[] si deve implementare un metodo scramble() che scambi il primo (da sinistra verso destra) numero pari con l'ultimo, il secondo numero pari con il penultimo, e così via. Per esempio:
– l'array [4, 1, 2, 8, 6] viene trasformato in [6, 1, 8, 2, 4]
– l'array [12, 4, 7, 2, 8, 6] viene trasformato in [6, 8, 7, 2, 4, 12].

3 Un array di interi è detto alterno se è formato da elementi ordinati in modo crescente, seguiti da elementi ordinati in modo decrescente. Per esempio, l'array [1, 4, 7, 5, 3, 2] è alterno mentre l'array [5, 4, 7, 3, 2] non lo è. Implementa un metodo statico boolean che restituisce true se array è alterno, false altrimenti.

4 La potatura di un array si ottiene rimuovendo da esso le occorrenze consecutive di uno stesso numero e rimpiazzandole con un'unica occorrenza di quel numero.
Per esempio, la potatura dell'array [7, 7, 4, 4, 5, 4, 4, 4] è costituita dall'array [7, 4, 5, 4].
Implementa la classe MyArray con una variabile d'istanza private mioArray[] e un metodo ricorsivo public potatura() che restituisca il valore potato di array.

5 Definisci una classe GestMatrici per l'elaborazione di matrici (rettangolari o quadrate) di dimensione massima 10x10 composte da numeri interi. Tale classe deve includere (almeno) i seguenti metodi:
- lettura(): riceve una matrice di interi A e ne legge il contenuto da tastiera per righe, partendo dall'ultima riga sino alla prima; il metodo restituisce la matrice letta;
- scrittura(): riceve una matrice di interi A e ne scrive il contenuto su output;
- cePalindroma(): riceve una matrice di interi A e restituisce true se il contenuto di A è palindromo, false altrimenti. Una matrice è da intendersi palindroma quando la prima riga è uguale all'ultima, la seconda alla penultima ecc., come quella riportata nell'esempio della figura a lato;

1	2	3	4	5
–5	10	20	30	–2
9	8	7	6	0
–5	10	20	30	–2
1	2	3	4	5

- prodottoScalare(): riceve una matrice di interi A e due indici di riga r1 e r2 e calcola e ritorna il prodotto scalare tra la riga r1 e la riga r2;
- main(): legge da tastiera le dimensioni di una matrice di interi *m* e provvede a inizializzarne il contenuto con la lettura da input. Successivamente il programma visualizza la matrice, scrive se essa è palindroma o meno e, infine, stampa il prodotto scalare tra la prima e l'ultima riga della matrice stessa.

6 Dati tre array di triangoli, rettangoli e quadrati si vuole costruire un nuovo array di elementi Poligoni che sia ordinato per aree crescenti e si vuole stampare l'elenco ordinato.
Progetta le classi opportune tali da utilizzare un programma di prova simile al seguente:

```
class TestPoligoni()
  public main()
    T = 3, R = 4, Q = 2
    poligoni = creaPoligoni(T,R,Q)
    ordinaPoligoni(poligoni)
    visualizzaElenco(poligoni)
  endMain
endClass
```

VERIFICA... le competenze avanzate

AREA DIGITALE

Esercizi per l'approfondimento

ESERCIZI/PROBLEMI

Gerarchie di classi per applicazioni gestionali

1 Definisci una classe Auto per rappresentare oggetti Automobile con il nome della marca, il nome del modello, la targa e l'anno di immatricolazione e con opportuni metodi d'istanza tra cui un metodo stampa() per la sua descrizione sullo schermo. Quindi definisci le classi Automezzo e Camion, indicando la corretta gerarchia e riscrivendo opportunamente i metodi.

2 Definisci una classe Studente per rappresentare oggetti Studente con il cognome, il nome, il codice fiscale, il numero di matricola e con opportuni metodi d'istanza tra cui un metodo stampa() per la sua descrizione sullo schermo. Quindi definisci le classi Bidello e Professore definendo la corretta gerarchia e riscrivendo opportunamente i metodi.

3 Un'azienda sanitaria desidera creare un archivio elettronico per la gestione dei propri medici di base e delle liste dei relativi pazienti. Definisci una classe Medico, avente una variabile d'istanza nominativo (stringa), e una classe Paziente, avente come variabili d'istanza codice (intero) e medico, che lo "connette" proprio medico curante.

4 Un'assicurazione desidera creare un archivio elettronico in grado di raccogliere informazioni sulle automobili e sui loro proprietari. Definisci una classe Cliente, avente il nominativo (stringa) come variabile d'istanza, una classe Automobile, avente come variabili d'istanza il numero di targa della vettura (intero) e un riferimento al proprietario della classe Cliente. Implementa con esse la classe Archivio.

5 Definisci una classe Libro contenente i seguenti attributi: nome del libro, prezzo, numero di scaffale, numero di pagine, casa editrice.
Definisci inoltre i metodi inizializza(), stampa(), applicaSconto(), che hanno i seguenti compiti:
- inizializzare i campi dati dell'oggetto classe;
- stampare tutti i dati dell'oggetto;
- diminuire del x% il prezzo del libro in oggetto.

6 Data la classe Libro per rappresentare oggetti Libro con il nome dell'autore, il titolo e il numero di pagine e con i relativi metodi d'istanza, scrivi i seguenti metodi:
- un metodo che, ricevendo come parametro un array di oggetti Libro, calcola e restituisce una struttura collegata con gli stessi oggetti Libro nello stesso ordine;
- un metodo che, ricevendo come parametro una struttura collegata di oggetti Libro, e il nome di un autore, verifica se nell'elenco esista almeno un libro dell'autore dato;
- un metodo che, ricevendo come parametro un codice ISBN, elimina il libro corrispondente dall'elenco;
- un metodo che ordina i libri rispetto al numero crescente di pagine.

7 Definisci una classe Prenotazione (di un campo da tennis) contenente il nome del cliente e l'ora della sua prenotazione; a tal fine implementa una classe Campo in cui ci siano i seguenti metodi:
- public boolean addPren(int inizio, int fine, String unNomeCliente), per prenotare il campo, che controlla se i dati inseriti sono giusti e se il campo è disponibile e, quindi, salva la prenotazione e restituisce true se il campo è stato prenotato;
- public boolean removePren (int inizio, int fine, String unNomeCliente), per cancellare una prenotazione, che controlla se il campo è stato prenotato dal cliente che vuole cancellare la prenotazione e, quindi, la cancella e restituisce true se la prenotazione è stata cancellata;
- public double utilizzo(), per trovare la percentuale di utilizzo del campo.

Simulazione guidata di compito in classe

Situazione

Definisci la classe Segmento i cui oggetti rappresentano dei segmenti orizzontali di lunghezza intera sul piano cartesiano a partire dalla classe Punto che descrive un punto in un piano.

Traccia per la soluzione

Dopo aver dato la definizione geometrica di segmento, individua gli elementi che costituiscono il segmento, classificandoli in base alla loro natura.

Passi operativi

Definisci la classe Punto che utilizzerai per istanziare un oggetto con le coordinate del punto di applicazione (o iniziale) del segmento.
Definisci inoltre i seguenti costruttori:
- un costruttore con due parametri, left di tipo Punto e lunghezza di tipo intero, che crei un segmento;
- con vertice sinistro di coordinate left e lunghezza del segmento;
- un costruttore con un parametro intero che crei un segmento con vertice destro (0,0) e lunghezza uguale al parametro se maggiore di zero, altrimenti di lunghezza unitaria;
- un costruttore con due parametri formali di tipo Punto (left e right), che rappresentano i vertici sinistro e destro del segmento se allineati, altrimenti con vertice sinistro left e lunghezza unitaria.

Definisci poi i seguenti metodi:
- getLeft(), che restituisce il vertice sinistro del segmento;
- getRight(), che restituisce il vertice destro del segmento;
- getLength(), che restituisce la lunghezza del segmento;
- inclusoIn(mioSegmento) di tipo boolean con un singolo parametro di tipo Segmento, che restituisce true se e solo se il segmento su cui il metodo viene invocato è contenuto nel segmento passato come parametro.

Proposte di compito in classe

1 Realizza un programma di gestione delle fatture prevedendo due tipi diversi di righe per la descrizione di un singolo articolo: un tipo per descrivere i prodotti che vengono acquistati in una determinata quantità numerica (come, per esempio, "tre tostapane") e un altro per descrivere un addebito fisso (come "spedizione: euro 5,00", oppure "imballo euro 2,00").
Genera un diagramma UML dell'implementazione delle classi.

2 Un contratto telefonico per un telefono fisso e un telefono mobile hanno in comune alcune caratteristiche riassunte dalla classe ContrattoTelefonico.
La sottoclasse ContrattoFisso riscrive un metodo della superclasse ed eredita gli altri. Analogamente, la sottoclasse ContrattoMobile eredita tutti i metodi di ContrattoTelefonico, salvo riscriverne uno per il calcolo della tariffa. La classe TestContrattoTelefonico testa le classi precedenti.

3 Realizza una classe Televisore e una classe Monitor applicando l'ereditarietà a partire da una classe ApparecchiaturaElettrica.
Dopo aver definito la corretta relazione gerarchica, evidenzia quali metodi rispettano l'overloading e quali l'overriding. Scrivi infine una classe di prova che istanzia un oggetto per ciascuna classe.