1 Classi e oggetti

- **Cos'è una classe?**

  Una classe è un modello che permette di descrivere un concetto, sia astratto che concreto. Una classe permette di rappresentare sia le proprietà (o lo stato) che il comportamento tipico del concetto che rappresenta. Le proprietà sono descritte tramite delle variabili che prendono il nome di attributi, mentre i comportamenti sono definiti da funzioni dette metodi.

- **Cos'è un oggetto?**

  Un oggetto è il risultato dell'istanziazione di una classe. Ovvero, è la rappresentazione di una specifica entità modellata a partire dalla classe che è stata istanziata. Ogni oggetto è indipendente dagli altri oggetti della stessa classe in quanto ogni oggetto ha un proprio stato.

- **Che differenza c'è tra classe e oggetto?**

  Una classe fornisce un modello per la definizione di oggetti, mentre un oggetto è uno specifico esemplare, definito sulla base della classe, ma indipendente da essa e da tutti gli altri oggetti creati a partire dalla stessa.

- **Che cos'è l'identificatore di un oggetto?**

  Gli identificatori sono i nomi assegnati alle variabili. L'identificatore di un oggetto è quindi il nome assegnato al riferimento di quell'oggetto.

- **Che differenza c'è tra le classi in Java e in C++?**

  Nelle classi in C++ la visibilità di attributi e metodi è organizzata tramite una sezione privata e una pubblica, mentre in Java la visibilità è gestita sul singolo attributo/metodo. In Java inoltre, non è necessario gestire le dealoccazioni degli oggetti in quanto vi è un *garbage collector* che si occupa di deallocare i blocchi di memoria non più referenziati. Infine, in Java si accede agli attributi e ai metodi di una classe (o oggetto) mediante notazione puntata anziché con `->` come in C++. Un'ultima importante differenza tra le classi in Java e C++ riguarda l'*ereditarietà multipla* che è concessa solo in C++.

- **In che senso "Java non ha i puntatori"?**

  "Java non ha i puntatori" significa che il linguaggio non mette a disposizione del programmatore una sintassi e degli operatori espliciti per la manipolazione dei puntatori. Cioè l'operatore di *deferenziazione* `*`, il comando `delete` e tutta l'algebra dei puntatori del C++ non esistono in Java. Viene mantenuto il comando `new` con l'unico significato di creare una nuova istanza di una classe. Inoltre, la dinamica del passaggio per riferimento o valore viene nascosta: tutti i valori, ad eccezione dei valori primitivi, vengono passati per riferimento senza che il programmatore debba segnarlo nella firma del metodo.

- **Com'è definita la firma di un metodo?**

  La firma di un metodo è composta dal nome del metodo e dalla lista ordinata dei tipi dei parametri.

- **Qual è il ruolo del `package`?**

  Un package è un contenitore che aggrega classi strettamente correlate tra loro. Un package può contenere classi private o pubbliche e può essere usato per importare in altri file classi (non solo) presenti al suo interno. Ovviamente, possono essere importate solo le classi pubbliche perché quelle private non sono visibili all'esterno. A livello di file system il package è mappato come una cartella e proprio per questo motivo è fondamentale che ogni package abbia un nome univoco, altrimenti si verificherebbero problemi di ambiguità.

- **Cos'è la variabile `this`?**

  La variabile `this` può essere usata solo all'interno di un metodo di una classe e identifica la specifica istanza sulla quale il metodo verrà poi chiamato. Tramite notazione puntata permette quindi di accedere a tutti gli attributi e tutti i metodo della classe. Il `this` può in realtà essere omesso, quando si vuole far riferimento a un attributo il cui nome non crea ambiguità all'interno dello scope in cui viene richiamato, cioè se il nome dell'attributo non è lo stesso di una variabile locale al metodo. Può sempre essere omesso quando si invocano metodi della classe. Usato all'interno di un costruttore permette di invocare un altro costruttore della stessa classe, ma deve essere la prima istruzione.

- **Come faccio a stampare un oggetto?**

  Per stampare un oggetto nel terminale, si fa uso del metodo `toString()` che restituisce una stringa rappresentativa dell'istanza da cui viene invocato. Questa stringa viene poi stampata a terminale dal metodo `System.out.println()` (o `.print()`).

2 Memoria

- **Quando (e come) viene usato lo stack?**

  Lo *stack* viene usato per memorizzare tutto ciò che non viene allocato dinamicamente dal programma. Contiene quindi, i record di attivazione delle funzioni, le variabili locali e le costanti. Lo *stack* funziona come una pila, cioè ogni blocco viene allocato "sopra" ai precedenti. Nel momento in cui si esce da una funzione o da uno scope, il relativo blocco viene rimosso dallo *stack*.

- **Quando (e come) viene usato lo heap?**

  Lo *heap* viene usato per memorizzare tutto ciò che viene allocato dinamicamente, cioè istanze di classi o array. Un oggetto può essere allocato nello *heap* tramite il comando `new`. È possibile accedere ai dati nello *heap* tramite accesso causale, ovvero conoscendo l'indirizzo di un blocco vi si può accedere indipendentemente dagli altri blocchi presenti.

- **Ci sono variabili che non stanno né in heap, né in stack?**

  Non esistono variabili che non stanno né nello *heap*, né nello *stack*.

- **Cos'è un memory leak?**

  Un *memory leak* è un problema che si verifica quando un'area di memoria allocata nello *heap* non è più raggiungibile da nessun punto del programma. Ciò è causa di inefficienza in quanto la memoria disponibile al programma non potrà più essere utilizzata fino al termine del programma stesso.

- **Cos'è il Garbage collector, come fa a sapere cosa può essere rimosso?**

  Il *Garbage collector* è un processo della *JVM* che si occupa di deallocare tutti i blocchi di memoria dello *heap* che non sono più accessibili dal programma. Il *Garbage collector* sa quando rimuovere un blocco perché ad ogni blocco di memoria allocato è associato un contatore che tiene traccia del numero di riferimenti attivi verso quell'area di memoria. Quando il contatore va a zero significa che non c'è più alcun riferimento attivo e dunque non è più possibile accedere ai dati in quel blocco che quindi può essere rimosso.

3 Classi e interfacce

- **Cos'è una classe astratta? Per cosa si usa?** Una *classe astratta* è una classe che non può essere istanziata in quanto non è completamente implementata. Ciò significa che una classe, definita astratta con la keyword `abstract`, può contenere al proprio interno definizioni di metodi,

anch'essi specificati facendo uso di `abstract`, privi di implementazione. L'utilità delle classi astratte si manifesta soltanto con l'utilizzo dell'*ereditarietà*, infatti questo tipo di classi vengono usate per modellare, ad un livello di astrazione più alto, concetti che saranno poi concretizzati da classi "concrete" che estendono una particolare classe astratta. Il compito di fornire un'implementazione, per tutti i metodi definiti astratti, viene quindi demandato alle classi figlie, le quali possono a loro volta essere definite astratte e quindi non implementare alcuni dei metodi ereditati oppure definire nuovi metodi astratti.

Al'interno di classi astratte possono comunque essere definiti attributi e metodi non astratti.

- **Cos'è un'interfaccia? Per cosa si usa?**

  Il concetto di *interfaccia* è molto simile al concetto di classe astratta. Un'interfaccia contiene soltanto il prototipo di metodi, la cui è implementazione è demandata alle classi che implementeranno quell'interfaccia, oppure valori costanti, definiti quindi, con la keyword `final`. Le interfacce non possono essere istanziate.

  Siccome una classe può implementare più di un'interfaccia, queste permettono di simulare il concetto di *ereditarietà multipla*. Inoltre, le interfacce stesse possono estendere più di un'interfaccia alla volta usando la solita keyword `extends`.

- **Che differenza c'è tra classi astratte e interfacce?**

  La principale differenza tra classi astratte e interfacce è che le classi astratte sono comunque delle classi, quindi possono avere attributi, costruttori e metodi non astratti, mentre le interfacce contengono solo prototipi e costanti. Poi, come detto in precedenza, le interfacce simulano l'*ereditarietà multipla*, mentre le classi astratte consentono solo l'*ereditarietà singola*.

4 Static

- **Che differenza c'è tra le variabili e i metodi statici e quelli "normali"?**

  Le variabili e i metodi statici prendono anche il nome di variabili (attributi) e metodi di classe anziché di istanza come quelli "normali". Ciò significa che per fare riferimento a variabili e metodi statici di una classe non è necessario istanziare quella classe. Questo implica che la variabili statiche sono condivise tra tutte le istanze di quella classe e, se visibili all'esterno della classe, sono accessibili mediante notazione puntata sulla classe. I metodi statici possono essere invocati dai metodi d'istanza, ma non vale il contrario, cioè tutti i metodi e le variabili d'istanza sono invisibili ai metodi statici. Questo è dovuto al fatto che il metodo non avrebbe modo di distinguere le variabili o i metodi tra le diverse istanze.

- **Perché l'uso di variabili statiche va limitato? Quando è lecito usarle?**

  L'uso di variabili statiche va limitato perché violano il principio di *incapsulazione*, perché potrebbero creare problemi nella programmazione *multithreaded* e in quanto il loro tempo di vita copre l'intera durata del programma. È lecito usarle nel caso in cui si tratti di costanti.

5 Visibilità

- **Quali sono i livelli di visibilità previsti in Java e qual è il loro significato?**

  In ordine crescente di ristrettività i modificatori di visibilità sono:

  1. **public**: la classe, l'attributo o il metodo su cui è usato è visibile da qualunque parte del programma anche all'esterno del package;
  2. **protected**: l'attributo o il metodo su cui è usato è visibile ovunque all'interno del package di definizione e all'interno di tutte le classi che estendo la classe in cui è definito;

3. **package**: la classe, l'attributo o il metodo su cui è usato è visibile ovunque all'interno del package in cui è definito. Questo è il modificatore di visibilità di default;
4. **private**: l'attributo o il metodo è visibile solo all'interno della classe in cui è definito;

- **Che differenza c'è tra dichiarare il livello di visibilità e dichiarare protected?**

  Un metodo o un attributo dichiarato *protected* è visibile sia nel package in cui è definito, come usando il modificatore *package*, sia in tutte le classi che estendono la classe in cui è definito.

6 Ereditarietà

- **Che differenze ci sono tra l'ereditarietà tra classi astratte e interfacce?**

  La differenza principale sta nel fatto che una classe può derivare solo da una classe astratta, mentre può implementare contemporaneamente più interfacce. Similmente, una classe definita astratta può derivare direttamente da una sola classe, mentre un'interfaccia può estendere più di un'interfaccia.

- **Cos'è una is-a? Cos'è una has-a?**

  Le relazioni `is-a` ed `has-a` sono entrambe relazioni che sussistono tra classi diverse. La prima è strettamente legata al concetto di *ereditarietà* ed è verificata da tutte le coppie di classi in cui la seconda è una delle super classi della prima. In particolare, prese due classi `A` e `B`, la relazione `A is-a B` è verificata se `A = B`, `A` deriva direttamente da `B`, oppure se risalendo la gerarchia delle classi partendo da `A` prima o poi si arriva a `B`.

  La relazione `has-a` è invece una relazione di appartenenza. Nello specifico, una relazione `A has-a B` è verificata se tra gli attributi di `A` vi è un'istanza di `B`, o di una classe che soddisfa una relazione `is-a B`.

- **Che differenza c'è tra overriding e overloading?**

  In una gerarchia di classi si definisce *overriding* la reimplementazione di un metodo, già implementato in una delle super classi, da parte di una classe figlia. In particolare, quando quel metodo viene invocato, l'implementazione scelta è quella della classe che corrisponde al tipo di dato deciso a runtime, cioè se il metodo viene invocato da un'istanza di una classe che soddisfa una relazione `is-a` con la classe che ha reimplementato il metodo, viene scelte la nuova implementazione altrimenti, quella vecchia.

  L'*overloading* invece, si ha quando esistono due o più metodi con lo stesso nome, ma firme diverse. In particolare, per realizzare l'overloading, i metodi devono condividere il nome, ma differire nella lista ordinata dei tipi dei parametri. Nulla vieta ai metodi di differire anche per il tipi di valore restituito, la lista delle eccezioni lanciate o per il modificatore di visibilità.

- **Cosa si intende per dynamic binding?**

  Il *dynamic binding* è un meccanismo in base al quale il tipo di una variabile o un parametro formale viene stabilito a runtime.

- **Come viene deciso quale metodo invocare in presenza di polimorfismo?**

  La decisione sfrutta il dynamic binding in quanto, mentre la firma del metodo può essere stabilità in fase di compilazione, l'implementazione è decisa in base al tipo determinato a runtime.

- **Che relazione c'è tra costruttore (e distruttore) ed ereditarietà?**

  Data una gerarchia di classi, quando si istanzia una classe ne viene invocato il costruttore, ma prima di invocare quel costruttore viene richiamato quello della sua classe madre. A sua volta, prima di invocare il costruttore della classe madre, viene invocato quella della classe precedente

nella gerarchia. La catena di invocazioni continua fino a che non si arriva al costruttore della classe `Object`.

Il distruttore viene invece invocato automaticamente prima che l'oggetto venga distrutto e, se il costruttore serviva per allocare le risorse necessarie, il distruttore si occupa di deallocare le stesse ed eventuali altre risorse allocate in un secondo momento. In Java il distruttore viene implementato dal metodo `finalize` che viene invocato dal *Garbage Collector*. Una fondamentale differenza col costruttore è che non esiste alcuna catena di invocazione dei distruttori, cioè il distruttore di una classe, se non esplicitato nel codice, non invoca il distruttore della super classe. Di conseguenza, se non viene implementato il metodo `finalize` non viene invocato alcun distruttore.

- **Cos'è la variabile super?**

  La variabile `super` è equivalente al `this`, ma fa riferimento all'istanza della super classe. Non è possibile usare `super` per accedere all'istanza di super classe della super classe.

- **Cosa si intender per "upcast" e quando è lecito?**

  Con *upcast* si intende l'operazione di casting che trasforma l'istanza di una classe, nell'istanza di una sua particolare super classe. Questa operazione è possibile solo quando il tipo dell'istanza soddisfa una relazione `is-a` con la classe verso cui si sta facendo il casting. Quando possibile, l'*upcasting* può essere implicito.

- **Cosa si intender per "downcast" e quando è lecito?**

  Il *downcast* è l'operazione di casting da una classe ad un tipo più specializzato, cioè ad un tipo più in basso nella gerarchia. Questo tipo di casting, è possibile solo se il tipo di destinazione è in una relazione `is-a` con il tipo originale dell'istanza.

- **A cosa serve l'operatore instanceof?**

  L'operatore `instanceof` serve per testare il tipo a runtime di un'istanza. Il test risulta positivo se il tipo dell'istanza è in relazione `is-a` con il tipo specificato nell'instanceof.

- **A cosa serve l'annotazione @Override?**

  L'annotazione `@Override` viene premessa alla definizione di un metodo quando si sta implementando un metodo astratto ereditato da un super classe o da un'interfaccia, o semplicemente quando si fa l'*overriding* di un metodo.

7 Equals

- **Come si scrive una equals corretta?**

  Una corretta implementazione del metodo `equals` deve riceve come parametro un riferimento di tipo `Object` e restituire `true` se e solo se l'oggetto sul quale è stato invocato il metodo `equals` è semanticamente uguale al parametro.

  Inoltre, una corretta implementazione deve soddisfare le seguenti:

  (i) *Proprietà riflessiva*: per ogni riferimento non nullo `x`, `x.equals(x)` ritorna `true`;

  (ii) *Simmetria*: per ogni riferimento non nullo `x` e `y`, `x.equals(y)` ritorna `true` se e solo se lo fa anche `y.equals(x)`;

  (iii) *Proprietà transitiva*: per ogni riferimento non nullo `x, y` e `z`, se `x.equals(y)` e `y.equals(z)` ritornano `true`, allora anche `x.equals(z)` lo fa;

  (iv) *Consistenza*: per ogni riferimento non nullo `x` e `y`, invocazioni diverse di `x.equals(y)` ritornano lo stesso valore, se nessuna delle informazioni usare dalla `equals` sono state modificate;

  (v) Per ogni riferimento non nullo `x`, `x.equals(null)` ritorna `false`;

- **Qual è il ruolo della hashCode?**

  La funzione `hashCode`, permette di rendere più efficiente a runtime i controlli di uguaglianza. In particolare, la `hashCode` mappa su un valore intero lo stato dell'istanza. Il comportamento di `hashCode` è legato al metodo `equals` e infatti valgono le seguenti:

  (i) Due oggetti semanticamente uguali devono avere lo stesso hash;

  (ii) Due oggetti con hash code diverso devono essere semanticamente diversi;

  La `hashCode` viene utilizzata soprattutto per velocizzare la ricerca di elementi all'interno di una struttura dati. Nelle strutture dati che utilizzano gli hash viene infatti sfruttata l'associazione tra hash e oggetto. Nello specifico, quando si inserisce un nuovo elemento questo viene posizionato all'indice corrispondente all'hash code e elementi diversi con lo stesso hash sono inseriti in una lista detta *bucket*. Per la ricerca di un oggetto, a questo punto, è sufficiente calcolarne l'hash e cercarlo nella lista corrispondente.

8 Comparazione ai fini dell'ordinamento

- **Come si scrive una compareTo corretta?**

  Una corretta implementazione del metodo `compareTo` riceve come parametro un riferimento di tipo `Object` e restituisce un intero positivo se l'istanza è maggiore del parametro, 0 se è uguale, negativo altrimenti.

  Inoltre, una corretta implementazione deve soddisfare le seguenti:

  (i) Per ogni `x` e `y` deve valere: `-sgn(x.compareTo(y)) == sgn(y.compareTo(x))`;

  (ii) *Proprietà transitiva*: se `x.compareTo(y) > 0` e `y.compareTo(z) > 0`, allora deve valere anche `x.compareTo(z) > 0`;

  (iii) Se `x.compareTo(y) == 0`, vale anche `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`;

  Infine, è consigliabile che `(x.compareTo(y) == 0) == x.equals(y)`. È bene notare che se le `equals` restituisce `true`, la `compareTo` deve restituire 0, ma non è sempre vero (quanto meno non è richiesto) che ciò sia vero a parti invertite.

- **Che differenza c'è tra Comparable e Comparator?**

  `Comparable` e `Comparator` sono due interfacce che consentono di definire l'ordinamento degli oggetti di una classe. La differenza sta nel fatto che l'interfaccia `Comparable` viene implementata dalla classe per la quale si vuole definire l'ordinamento, e tipicamente la si usa per codificare l'ordinamento naturale di quella classe (e.g. ordine alfabetico per le stringhe, cronologico per le date). L'interfaccia `Comparator` viene invece implementata da una classe terza che si occupa soltanto di definire un ordinamento diverso, tipicamente, da quello naturale (e.g. ordine per titolo di studio delle persone, invece che per nome).

  Le interfacce `Comparable` e `Comparator` forniscono rispettivamente i metodi `compareTo(Object o)` e `compare(Object o1, Object o2)`. Il metodo `compare` è soggetto agli stessi vincoli di `compareTo`.

9 Collections

- **Perché quasi sempre conviene usare collections invece arrays?**

  Sia gli array che le collection sono raggruppamenti di elementi. La prima differenza sta nella dimensione che è fissa negli array e variabile nelle collection. Inoltre, le collection mettono a disposizione classi che implementano alcuni algoritmi (e.g. ricerca, ordinamento) che altrimenti andrebbero implementati ex novo se si usassero gli array.

- **Che differenza c'è tra liste e set?**

  Le liste possono contenere più copie dello stesso oggetto e ogni elemento è accessibile tramite indice, mentre nei set non ammessi duplicati e non è possibile utilizzare indici.

- **Quando una collezione è ordinabile?** Una collezione è ordinabile quando i suoi elementi lo sono, ovvero quando implementano l'interfaccia `Comparable`.

- **Come posso mescolare/ordinare una collection?**

  Una collection può essere mescolata o ordinata rispettivamente con i metodi `shuffle` e `sort` della classe `Collections`. Se si vuole ordinare una collection utilizzando un ordinamento non naturale, ovvero se si vuole sfruttare un Comparator, nel metodo `sort` è possibile passare come secondo parametro un'istanza del comparatore.

  Tuttavia, le collections che sono intrinsecamente ordinate non è possibile invocare né il metodo `shuffle`, né `sort` poiché l'ordine degli elementi è stabilito dall'implementazione di quella particolare collection.

- **Perché la definizione di equals e hashCode è importante per le classi i cui oggetti sono inseriti in una Collection?**

  Nel caso di collection che non ammettono elementi duplicati i metodi `equals` e `hashCode` sono utilizzati per verificare l'unicità degli elementi, pertanto nel caso di cattive implementazioni dei suddetti l'unicità non può essere garantita.

  Inoltre, il metodo `hashCode` viene usato nelle strutture che utilizzano gli hash come chiavi (e.g HashMap) mentre, il metodo `equals` potrebbe essere richiamato dai metodi `compareTo` e `compare` per cui nel caso di cattive implementazioni potrebbero verificarsi comportamenti scorretti.

- **Come posso scorrere gli elementi di una collezione?** È possibile scorrere tutti gli elementi di una collezione utilizzando gli iteratori o con ciclo *for-each*. Inoltre, se la collection permette di accedere agli elementi tramite indice è possibile sfruttare un generico ciclo *for* iterando su tutti i possibili indici.

- **Come si usa un iteratore?** Tramite il metodo `iterator()` si ottiene un iteratore per la collection e tramite `hasNext()` e `next()` è possibile, se ce n'è uno, passare al prossimo elemento della collection. È anche possibile rimuovere un elemento dalla collection utilizzando il metodo `remove()` sull'iteratore.

- **Che differenza tra usare un iteratore e un ciclo for generalizzato?**

  Il ciclo for utilizza comunque un iteratore, tuttavia non è possibile rimuovere gli elementi dall collezione, in quanto non c'è alcun riferimento all'iteratore sul quale possa essere invocato il metodo `remove()`.

10 Gestione degli errori - terminazione del programma

- **Che cos'è e a cosa serve la assert?**

  La `assert()` è una funzione del C++ che testa una condizione e nel caso in cui la condizione non sia verificata termina il programma. Una peculiarità delle `assert()` è che se non abilitate tramite alcuni flag del compilatore non vengono eseguite.

- **Come si usa il costrutto try-catch?**

  Il costrutto *try-catch* permette di catturare le eccezioni e gestirle. In particolare, il codice che potrebbe generare un'eccezione viene racchiuso all'interno della clausola `try`. Ci sono poi una o più clausole di `catch` che catturano una più eccezioni. Nel momento in cui un'eccezione viene

lanciata, le clausole vengono controllate partendo dalla prima e viene eseguito il codice della prima clausola che cattura il tipo di eccezione lanciato.

È possibile gestire più di un'eccezione in un singolo catch specificandone i tipo separandoli con un pipe (|), oppure si può anche ignorare delle eccezioni specificando `ignored` come nome del parametro del `catch`.

- **A cosa serve la clausola finally?**

  La clausola finally contiene codice che deve essere eseguito indipendentemente dal fatto che siano state lanciate eccezioni o meno. Cioè viene eseguito al termine del codice dentro il `try` o dentro la clausole `catch` che è stata attivata.

- **Come faccio a far terminare un programma?**

  È possibile terminare un programma invocando il metodo `System.exit(int x)` e fornendo come parametro il codice di terminazione.