Metodologie di Programmazione

Principi dell’OOP - Ereditarietà ed Interfacce

Le classi in Java

La programmazione orientata agli oggetti consiste nel definire oggetti che interagiscono tra loro, ognuno deve rappresentare un singolo concetto. Si definiscono le **classi**, tipi di dati astratti rappresentanti un oggetto, con determinati attributi (variabili della classe) e operazioni (metodi della classe).

I metodi possono essere:

- ***Accessor*** - ossia accedere agli attributi della classe senza modificarli (ad esempio un metodo che ritorna l’area di un rettangolo).
- ***Mutator*** - modificano lo stato (variabili) della classe.

Una **classe immutabile** non ha metodi Mutator.

Le varie classi sono organizzate e definite da una relazione di ereditarietà. È possibile definire una **sotto-classe** di una classe già esistente, che erediterà i metodi e gli attributi della sua classe padre, detta **super-classe**, definendone anche di nuovi. Una classe padre modella un concetto generico ed una sua sotto-classe un concetto più specifico. Per definire che una classe è figlia di un’altra classe si utilizza la nomenclatura `extends` . Vediamo un esempio tramite codice :

```
public class Person{ //Classe padre
  private String name;
  private String surname;
  public Person(String n,String s){
    name=n;
    surname=s;
  }
  public String getName(){
    return name;
  }
}
public class Student extends Person { //sotto-classe di Person
  private int studentId;
  public Student(String n,String s,int m){
    super(n,s);
    studentId=m;
  }
```

```
  public int getId(){
    return studentId;
  }
}
```

All'interno di una sotto-classe, è possibile accedere ai metodi della sua super classe tramite la nomenclatura `super`, ad esempio, se dentro la classe `Person` fosse definito un metodo chiamato `getInfo()`, sarebbe possibile richiamarlo dentro un metodo della sua sotto-classe `Student` tramite la nomenclatura `super.getInfo()`.

Tutte le classi in Java **sono figlie di una classe comune Object**, essa ha definiti i metodi :

- `toString()` - ritorna una stringa contenente le informazioni dell'oggetto.
- `equals(Object e)` - ritorna True se l'oggetto è identico all'oggetto passato come parametro.
- `clone()` - ritorna una copia duplicata dell'oggetto.

Si ricordi che solamente due numeri possono essere confrontati con il costrutto `==`, qualsiasi altro oggetto per essere confrontato con un altro oggetto necessita del metodo `equals()`.

Modificatori di accesso

Java, fornisce alle classi, metodi e variabile, quattro livelli per il **controllo dell'accesso**, ossia è possibile definire per ogni classe, metodo o attributo , un **modificatore di accesso**, che ne determinerà la **disponibilità di utilizzo** all'interno del codice.

- `public` - una classe/attributo/metodo public è accessibile da qualsiasi altra classe all'interno del codice.
- `private` - ad un metodo/attributo definito come private si può accedere esclusivamente nella classe nella quale è definita.
- `protected` - ad un metodo/classe /attributo protected si può accedere esclusivamente nella stessa classe in cui è definito, nelle sotto-classi della classe in cui è definito e nelle classi appartenenti allo stesso $Package$.
- `default/package` - Se non si definisce nessun modificatore per un metodo/classe /attributo, esso sarà accessibile esclusivamente nella classe in cui è definito e nelle classi appartenenti all stesso $Package$.

È buona norma che gli attributi di esemplare delle classi siano privati, e che i metodi siano tutti o privati o pubblici. L'accesso alle classi dovrebbe essere pubblico o di pacchetto.

Esistono inoltre, due ulteriori modificatori :

- `final` - Un attributo definito come final è una **costante** e non può essere modificato. Un metodo final non può essere sovrascritto con l'*override* ed una classe final non può avere sotto-classi.
- `static` - Un attributo statico è condiviso da tutte le istanze della classe, sarà quindi una **variabile globale**, un metodo statico invece sarà comune per tutte le istanze della classe. Sia un attributo che un metodo statico sono accessibili all'interno di un altra classe senza dover per forza istanziare un oggetto. Ad esempio, la classe `Math` ha una variabile statica e pubblica `pi`, accessibile ovunque con al dicitura `Math.pi`, senza istanziare nessun oggetto di tipo `Math`.

Polimorfismo

Il polimorfismo è un paradigma che permette alle classi in Java di assumere diverse proprietà in base ai tipi di dati da manipolare, in termini più tecnici, un oggetto può assumere diverse forme, essere trattato come un istanza della sua classe, o come un istanza di una delle sue super-classi. Il polimorfismo è quindi il principio secondo cui il **tipo effettivo** di un oggetto **determina** il **metodo da chiamare**.

Il polimorfismo viene applicato tramite due importanti costrutti di Java :

Overloading

Con *Overloading* si intende la possibilità di definire **più metodi con lo stesso nome** che operano i dati in modo diverso all'interno di una classe, che però divergono nella quantità e nei tipi dei parametri in ingresso. Un esempio di *Overloading* :

```
public class Calculator{
  public int somma(int a, int b){
    return a+b;
  }
  public int somma(int a, int b,int c){
    return a+b+c;
  }
}
```

Overriding

Si intende la possibilità di definire all'interno di una classe, un metodo con lo **stesso nome e semantica di un metodo appartenente alla sua super-classe**, ma avendo un corpo della funzione diverso, rendendo il codice più specifico alla classe figlia. Un esempio di *Overriding* :

```
public class Person{
  private String name;
  private String surname;
  public void printInfo(){
    System.out.println(name);
    System.out.println(surname);
  }
public class Student extends Person{
  private int studentId;
  @override //Non è obbligatorio, ma è buona norma segnalare così un override
  public void printInfo(){
    super.printInfo(); //Richiama il metodo della sua super-classe
    System.out.println(studentId);
  }
}
```

Classe astratta

Una classe astratta in java, non è altro che una classe talmente generica, che è utilizzata per definire caratteristiche comuni fra le proprie classi figlie, ma che in se, non ha bisogno di essere istanziata, e non è quindi possibile farlo. Ad esempio, se nel nostro programma dobbiamo definire il comportamento di due classi `Student` e `Professor`, possiamo definire una loro classe padre astratta `Person` , che definirà dei **metodi astratti** che non avranno corpo, i quali le classi figlie dovranno *overridare* e definire. Una classe ed i suoi metodi astratti vanno dichiarati con la nomenclatura `abstract`. Vediamo un esempio di codice :

```
public abstract class Animal{
  public abstract void emitVerse(); //metodo astratto senza corpo
}
public class Dog extends Animal{
  @override
  public void emitVerse(){ //metodo astratto dichiarato nella classe padre
    System.out.println("Bau!");
  }
}
public class Cat extends Animal{
  @override
  public void emitVerse(){ //metodo astratto dichiarato nella classe padre
    System.out.println("Miao!");
```

```
  }
}
```

Le interfacce in Java

In Java, quando si vuole descrivere il **comportamento comune** di un insieme di classi è possibile definire un **interfaccia**. Un'interfaccia è come un **etichetta** che si assegna ad una certa classe, se essa fa parte di quell'etichetta, allora avrà a disposizione un insieme di metodi definiti da essa.

Un interfaccia, quindi, **dichiara un insieme di metodi** e le loro firme. Si definisce in maniera simile ad una classe con la nomenclatura `interface`, solo che non può avere attributi, e tutti i suoi metodi saranno pubblici ed astratti, ossia senza corpo, e le classi appartenenti a tale interfaccia dovranno **obbligatoriamente** eseguire l'*override* dei suoi metodi.

```
public interface Measurable{
  double getMeasure();
}
```

Definita l'interfaccia `Measurable`, tutte le classi che la *implementeranno* dovranno sovrascrivere il metodo `getMeasure()` definendone il corpo.

Per indicare che una classe appartiene ad un interfaccia, si utilizza la nomenclatura `implements`, una classe può estendere una sola classe, ma può *implementare* più interfacce.

```
public class Usb implements Measurable{
  private double capacity;
  public Usb(double c) { capacity = c; }
  public double getMeasure(){ //Deve obbligatoriamente definire il metodo
    return capacity;
  }
}
```

Non è possibile istanziare un interfaccia, è però possibile definire un oggetto di un tipo di un interfaccia, **istanziandolo** con una delle classe che la **implementa**.

```
Measurable x = new Usb(4096);
```

Eccezioni

Un eccezione non è altro che l'interruzione del flusso di esecuzione del programma, e può essere segno di un errore durante la computazione, quando viene "lanciata" un eccezione, il metodo nella quale è stata lanciata termina la sua esecuzione.

Un eccezione può essere :

- **Controllata** - Un eccezione dovuta ad una condizione esterna che il programmatore non può controllare, quindi gestisce *prevedendola*, segnalando con apposite tecniche che potrebbe esser generata un eccezione, la maggior parte di esse son dovute alla gestione dei dati in ingresso ed uscita, essendo terreno fertile di guasti esterni che non sono sotto il controllo di chi scrive il codice. Un esempio di eccezioni non controllate sono tutte le `IOException`. Un possibile esempio è, quando alla richiesta di inserire un percorso in console di un file, l'utente inserisce un percorso di un file inesistente, generando una `FileNotFoundException`.
- **Non Controllata** - Le eccezioni che il programmatore non aveva previsto, dovute quindi ad un proprio errore, ossia tutte quelle eccezioni che avvengono *Run-time*, che sarebbero potute essere evitate. Tutte le eccezioni `RuntimeException` ed `Error` sono non controllate. Un possibile esempio può essere quando si scorre un Array, ma erroneamente si controlla un indice i che risulta essere maggiore della lunghezza di tale Array.

Esistono due possibilità di gestire un eccezione.

Lanciare un eccezione

La prima consiste nel segnalare che il metodo potrebbe generare l'eccezione con l'enunciato `throws` seguito dal tipo di eccezione, esse farà si che, se l'eccezione si verificherà, il metodo verrà interrotto.

```
public void read() throws FileNotFoundException {
 //Codice che potrebbe generare l'eccezione
}
```

Catturare un eccezione

Il secondo metodo consiste nello specificare un certo blocco di codice che potrebbe potenzialmente generare un eccezione, e dichiarare un altra porzione di codice che verrà eseguita

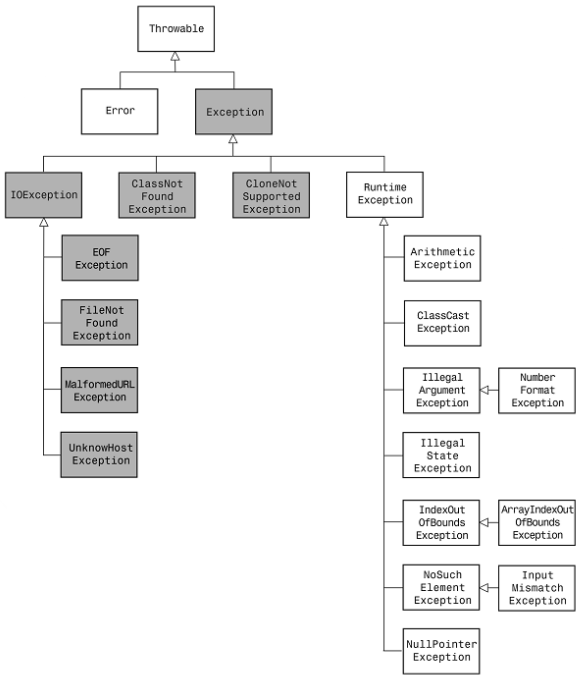
se e solo se tale eccezione si verifichi. Ciò si fa tramite l'enunciato `try/catch`, dentro il blocco `try` si inserisce il codice "sospetto" di generare eccezioni, e dentro il catch, definendo il tipo di eccezione che si vuole catturare, il conseguente codice di "risposta".

```
public void read(){
 try{
      //Codice che potrebbe generare l'eccezione
  }catch(IOException e){
      //l'oggetto "e" avrà informazioni riguardo l'errore
  }
}
```

Se un eccezione viene lanciata, le rimanenti istruzioni del blocco `try` non verranno eseguite.

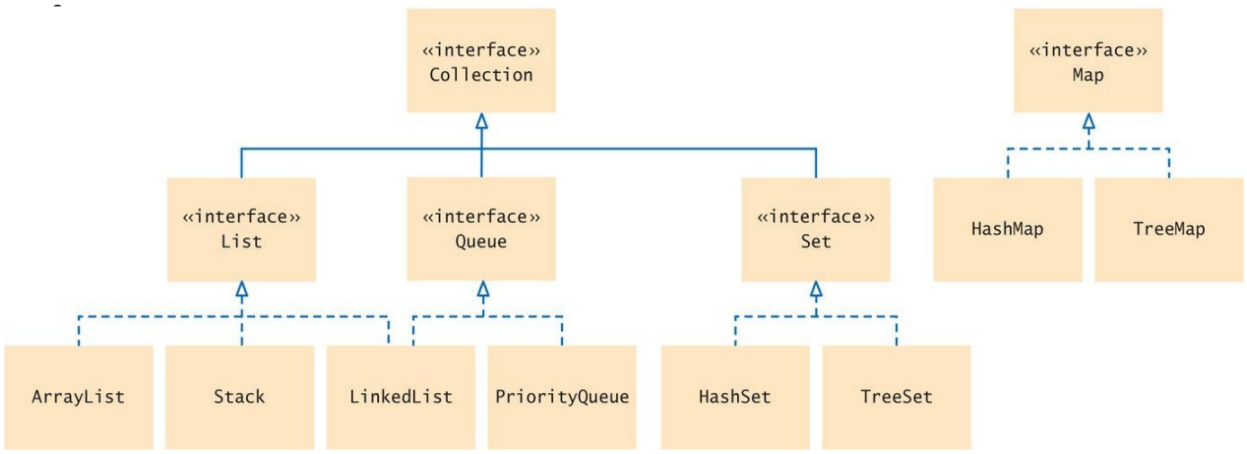
È possibile aggiungere un aggiuntiva clausola `finally` che verrà eseguita al termine del blocco `try` a prescindere dalla cattura o no dell'eccezione.

Gerarchia delle eccezioni



Le collezioni in Java

Una collezione è un gruppo di elementi che possono essere selezionati ed “utilizzati”, in Java sono gestiti dal framework delle collezioni, e sono organizzate in un albero gerarchico di classi ed interfacce.



Ovviamente tutte le classi/interfacce figlie dell’interfaccia padre “Collection” condividono gli stessi metodi.

L’interfaccia Collection

Metodi

L’interfaccia Collection fa si che tutte le classi/interfacce che la implementano ereditino i seguenti metodi :

`size()` - Ritorna il numero di elementi della collezione.

`add(elemento)` - Aggiunge l’elemento passato come parametro alla collezione.

`toString()` - Ritorna una stringa con scritti gli elementi della collezione.

`remove(elemento)` - Elimina dalla collezione l’elemento passato come parametro, se presente esso verrà eliminato ed il metodo ritornerà *true*, altrimenti false.

`contains(elemento)` - Ritorna *true* se l’elemento passato come parametro è presente all’interno della collezione, altrimenti false.

`for ( Object e : collezione ){}` - è possibile iterare gli elementi di una collezione con la seguente sintassi, all’interno del ciclo si accederà sequenzialmente a tutti gli oggetti di tipo

`Object` appartenenti alla Collection `collezione` tramite il nome della variabile `e`.

List

L'interfaccia **lista** descrive una collezione di oggetti *ordinati*, che tiene quindi conto dell'ordine, è implementata da $ArrayList$, $LinkedList$ e $Stack$. Data la sua implementazione fisica, è opportuno utilizzare una $LinkedList$ quando siamo interessati all'efficienza riguardo inserire/rimuovere elementi e non dobbiamo accedere *randomicamente* ad essi, dato che l'accesso sequenziale è molto più efficiente di quello randomico.

Alcuni metodi aggiuntivi di $LinkedList$:

`addLast(e)` - Aggiunge L'elemento passato come parametro all'inizio della lista.

`addFirst(e)` -Aggiunge L'elemento passato come parametro alla fine della lista.

`getLast()` -Ritorna l'ultimo elemento della lista.

`getFirst()` -Ritorna il primo elemento della lista.

`removeLast(e)` -Rimuove L'elemento passato come parametro alla fine della lista.

`removeFirst(e)` - Rimuove L'elemento passato come parametro all'inizio della lista.

Set

L'interfaccia **set** descrive un insieme disordinato di elementi **unici**, ossia come un insieme in termini matematici. Raggruppa gli elementi in modo che trovarli, aggiungerli e rimuoverli sia efficiente, è implementata da $HashSet$ e $TreeSet$, a livello di implementazione fisica, il primo utilizza una tabella hash, il secondo un albero binario di ricerca.

Stack

La classe **stack** non è altro che una pila (come la pila di sistema), tiene conto dell'ordine, ma l'inserimento avviene in testa e la rimozione pure, seguendo una politica FILO.

Alcuni metodi aggiuntivi di $Stack$:

`push(e)` - Aggiunge in testa l'elemento passato come parametro.

`pop()` - Rimuove l'elemento in testa allo Stack.

`peek()` - Ritorna l'elemento in testa allo Stack.

Queue

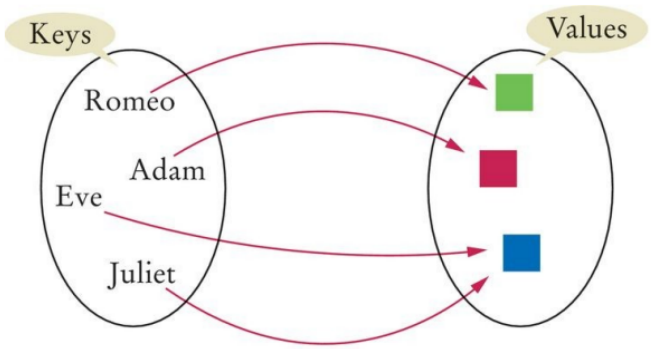
L'interfaccia **queue** descrive il funzionamento di una coda, ossia, segue una politica FIFO, l'inserimento avviene in testa e la rimozione in coda. È implementata da $PriorityQueue$ e $LinkedList$. Un metodo aggiuntivo delle Queue è

`peek()` - Ritorna l'elemento in testa allo Stack.

La **PriorityQueue** è una coda che ha però come metro di selezione di rimozione di un elemento, non il momento in cui è stato inserito l'elemento ma un valore aggiuntivo detto "priorità". L'elemento con la priorità più elevata ha diritto ad essere rimosso prima.

L'interfaccia Map

L'interfaccia **Map** è distaccata da Collection perchè descrive non una collezione di elementi, bensì una collezione di *coppie* chiave-valore, quello che di solito è noto col nome di *dizionario*, ogni chiave nella Map ha un valore associato, le chiavi sono **tutte** uniche tra loro. Infatti, le chiavi senza valori, sono raggruppabili in un Set. È implementata da $HashMap$ e $TreeMap$. A livello di implementazione fisica, il primo utilizza una tabella *hash*, il secondo un albero binario di ricerca.



Metodi

`put(key,value)` - Aggiunge al dizionario la coppia chiave-valore passata come parametro, il primo parametro è la chiave il secondo è il valore. Si ricordi che passando come parametro una chiave già presente, il dizionario non farà altro che aggiornare il suo valore.

`get(key)` - Ritorna il valore associato alla chiave passata come parametro.

`remove(key)` - Elimina la coppia chiave-valore data dalla chiave passata come parametro.

`keySet()` - Ritorna un oggetto Set contenente tutte le chiavi appartenenti al dizionario.

L’utilizzo dei Generics

La **programmazione generica** consiste nel creare costrutti di programmazione che funzionano con differenti *tipi* di dati, le *classi generiche* utilizzano uno o più parametri di tipo generico.

Esempio di un metodo generico :

`public void add(E element)` Si definisce un parametro generico con la nomenclatura `E` piuttosto che dichiararne il tipo, quel parametro `element` potrà essere un intero o una stringa, non importa, dato che un metodo generico è in grado di lavorare con il dato a prescindere dal suo tipo. Quando viene istanziato un oggetto di una classe generica, si definisce un **parametro-tipo**.

Le collezioni in Java sono generiche, ad esempio, posso creare un $ArrayList$ di stringhe, di interi, o di una classe nuova definita dal programmatore, l’importante è che, quando istanzio l’oggetto di una classe generica, ne definisco il parametro-tipo tramite la sintassi corretta, ossia :

```
ArrayList<String> listaStringhe = new ArrayList<String>();
```

Come si può notare, all’interno dei caratteri `<>` definisco il tipo. È importante sapere che **non si possono utilizzare tipi primitivi come parametri**, bensì, bisogna utilizzare le classi involucro!

```
ArrayList<float> listaFloat = new ArrayList<float>();
```
SBAGLIATO

```
ArrayList<Float> listaFloat = new ArrayList<Float>();
```
CORRETTO

L’utilizzo dei parametri-tipo rende più sicuro e facile da leggere il codice.

Se istanzio un $ArrayList$ di Stringhe, mi sarà impossibile aggiungere a quella collezione un intero, ed il compilatore mi segnalerà l’errore prima di poter eseguire il codice.

Quando si definisce un metodo/classe generica, e si utilizza appunto un parametro generico, è possibile **costringere** il compilatore a far si che, il parametro che si vorrà utilizzare deve obbligatoriamente far parte di una determinata interfaccia o classe.

Ossia, creare un metodo generico che accetti esclusivamente tipi figli di una certa classe o che implementino una certa interfaccia. Ciò si può fare con la nomenclatura :

```
public void <E extends Measurable > get(E object)
```

L’oggetto o una delle sue superclassi devono per forza estendere o implementare `Mesurable`.

Al livello fisico, quando il codice viene compilato, la **Java Virtual Machine** elimina i parametri-tipo e vengono letti come $Object$.

Gli Eventi in Java

Con **eventi dell'interfaccia utente** si intendono tutti quei "segnali" che vengono dall'utente alla quale il programma deve rispondere una volta che si verificano, come la pressione di un pulsante o il movimento del mouse.

Un programma può indicare un insieme di determinati eventi che gli interessano e definire una risposta per essi. Quando si parla di eventi, esistono :

- **Ricevitore di eventi** - È la risposta ad un evento, è classe definita dal programmatore e descrive i metodi da eseguire quando si verifica un particolare evento, ogni programma indica quali eventi ricevere mediante l'istallazione di oggetti ricevitori di eventi.
- **Sorgente di eventi** - La sorgente genera gli eventi stessi, e quando accade, invoca tutti i ricevitori di tale evento.

In Java esiste l'interfaccia `ActionListener`, e definisce il metodo:

`void actionPerformed(ActionEvent event)` dove il parametro `event` contiene dettagli dell'evento, come l'istante temporale in cui è avvenuto. Vediamo un esempio di codice in cui si definisce una classe ricevitore.

```
public class ClickListener implements ActionListener{
  public void actionPerformed(ActionEvent event)
  {
    System.out.println("Il pulsante è stato premuto");
  }
}
```

Adesso, quando ad un pulsante si assegnerà il ricevitore `ClickListener`, se esso verrà premuto eseguirà il metodo, scrivendo a schermo : *"Il pulsante è stato premuto"*.

Spesso, una classe che funge da ricevitore, per comodità, viene realizzata come **classe interna**, ossia come classe all'interno di una altra classe, utilizzabile esclusivamente in quel contesto, senza creare confusione nella restante parte del progetto in cui non serve.

Principi Solid

S - Single responsibility principle

- Ogni classe dovrebbe avere una ed una sola responsabilità, interamente **incapsulata** al suo interno.
- ***Esempio***: se esiste una classe Database in grado di creare una connessione, leggere e modificare i dati del database, ogni compito deve essere affidato ad un'istanza separata (`ConnectionHandler`, `DataReader`, `DataWriter`)

O - Open/closed principle

- Un'entità software dovrebbe essere aperta alle **estensioni**, ma chiusa alle modifiche.
- ***Esempio***: una classe non dovrebbe possedere attributi **troppo specifici**, poiché impedirebbe la possibilità di estenderne il funzionamento.

L - Liskov substitution principle

- Gli oggetti dovrebbero poter essere sostituiti con dei loro **sottotipi**, senza alterare il comportamento del programma che li utilizza.
- ***Esempio***: se una classe implementa un'istanza della classe `Animale`, sostituirla con la classe Cane non deve comportare modifiche al programma.

I - Interface segregation principle

- Sarebbero preferibili **più interfacce specifiche**, che una singola generica.
- ***Esempio***: un'interfaccia `Measurable` risulta troppo generica, poiché ogni oggetto è potenzialmente misurabile in qualche modo.

D - Dependency inversion principle

- Una classe dovrebbe dipendere dalle **astrazioni**, non da classi concrete.
- ***Esempio***: se esiste una classe Animale e varie sue sottoclassi (`Cane`, `Gatto`, ...), il codice implementante un'istanza di tali classi deve essere scritto basandosi sulla classe `Animale` e non sulle sue sottoclassi.