# Esercizi Programmazione

October 13, 2017

## Impostazione

Il programma è strutturato in modo da lanciare automaticamente i programmi gia' implementati e testarli con qualche input che ho scelto.
Usando il comando *make* potrai lanciare il programma che vuoi testare. Ad esempio

```
make 3
```

compilerà il file *ex03.java* e cercherà di eseguirlo con vari test da me scelti. Considererò un programma completo se ha passato tutti i test

## Descrizione del problema

Troverai tutti i file nella directory *tasks* con già la classe principale implementata e la descrizione del problema.
Questa avrà un titolo e la consegna. l'ultima riga rappresenterà l'input che dovresti aspettarti con le seguenti convenzioni

- **int** per input intero
- **long** per input intero in doppia lunghezza
- **double** per input reale
- **string** per input simbolico

Se non è presente nessun parametro, il programma non deve leggere alcun parametro

## Soluzioni

Nel caso non riuscissi a risolvere il problema, ho fornito una mia soluzione nella directory *solutions*. Usala come ultima risorsa. Usa allo stesso modo il makefile e osserva le soluzioni da me fornite. Anche se il programma termina con successo, alcuni risultati potrebbero differire dai tuoi. Fai attenzione e in bocca al lupo!

# Esercizi Java

Di seguito elecherò una serie di esercizi riguardante la Programmazione a oggetti (OOP). Questi richiedono una maggiore conoscenza di Java e possono essere in realtà essere risolti in modi diversi, non usando comunque la OOP, perciò cerca di usare il paradigma ad oggetti ove possibile

## Ereditarieta'

```
Object
    AbstractCollection
        AbstractList
            ArrayList
```

Un elemento chiave della OOP è la capacità di creare superclassi estese di volta in volta in base al problema da risolvere. Ciò porta all'efficiente propagazione di una modifica verso tutte le sottoclassi: se in una nuova versione di Java si decide che un oggetto debba avere un parametro aggiuntivo, modificando la classe *Object* si propaga la modifica a tutto ciò che e' un oggetto (in questo caso veramente qualsiasi cosa). Così come i miglioramenti si propagano anche gli errori, quindi parti disegnando bene il tuo sistema.
Cerchiamo di esplorare l'efficienza dell'ereditarieta'.

## Polimorfismo

Attraverso le interfacce, è possibile scrivere una serie di metodi che una classe deve avere. Una classe può implementare piu' interfacce ed è quindi meno vincolante dell'ereditarietà che la limita a una. Vedila piu' semplicemente come un pezzo di un puzzle che posso attaccare alla classe *"Se mi implementi questi metodi, ti posso considerare come quell'interfaccia"*
Inoltre una stessa interfaccia può essere implementata da piu' classi. Tutte queste classi perciò possono essere considerate come quella stessa interfaccia, ma sono effettivamente diverse. Un esempio sarà piu' chiaro:

```
Shape
 -> Circle
 -> Square
 -> Triangle
```

Tutte le forme, Circle, Square e Triangle sono classi diverse che implementano la stessa interfaccia *Shape*. E' possibile comunque avere un array di Shape e popolarlo con tutte e 3 le classi.

```
Shape s = new Triangle();
```

è una sintassi perfettamente valida, poichè Triangle può essere effettivamente visto come Shape.

## Incapsulamento

Sempre attraverso l'ereditarieta', una sottoclasse non reinventa la ruota da 0 ma si ritrova a sua disposizione metodi e parametri che le superclassi hanno scelto di rendere pubblici o protetti. In questo senso tutti i dettagli implementativi delle superclassi vengono nascosti e ne viene quindi garantito il corretto funzionamento.

## Esercizio 1

Implementare una classe *Person* strutturata come segue

```
class Person
    Person(String name, String surname)
    String getName()
    String getSurname()
    void sayHi()
```

dove sayHi() stampi a schermo il nome e il cognome della persona.

In seguito scrivi una classe *Student* che includa anche un campo intero *badge_number* con relativo getter (il metodo che restituisce il campo).
Sempre ereditando da *Person* crea anche una classe *Teacher* con un array di stringhe dei corsi di cui di occupa.
Nel main crea un array di *Person* e popolalo con 2 persone, 3 studenti e un insegnante. Tutti dovranno dire il proprio nome.
Ipotizza che passi del tempo e l'idea di persona non ti soddisfa piu': ti serve anche un campo che ne registri il codice fiscale. Effettua le dovute modifiche in modo che anche le sottoclassi siano preparate a ricevere un codice fiscale in input (crea altri costruttori o modifica l'originale).

- Nota che in tutte le tue istanze è presente il metodo *sayHi()* grazie all'ereditarieta'. Nota anche che dopo aver fatto una modifica a un costruttore, sei costretto a rimaneggiare tutte le successive chiamate nella tua code base. Per piccoli progetti non è un problema, ma quando lavori con decine di migliaia di righe di codice, preferiresti spendere il pomeriggio a lavorare a una nuova feature piuttosto che ripassare tutte le classi del progetto in cerca delle chiamate ai costruttori ;)

## Esercizio 2

Crea due interfacce una chiamata *Quackable* e una *Swimmable*. Crea successivamente una classe papera che implementi l'interfaccia *Swimmable* stampando il nome del metodo per accertarne la chiamata..
Crea due sottoclassi di papera che implementino l'interfaccia *Quackable* ritornando come floating point la velocità di nuoto. Crea una terza classe papera che erediti da papera ma non implementi il metodo *Quack*.
Istanzia un oggetto per classe e fai un Array di *Swimmable* e un array di *Quackable* chiamandone i relativi metodi stampando i risultati a schermo. Gli array devono contenere tutti gli oggetti che possono starci.

## Esercizio 3

Crea una classe astratta *macchina* con i metodi riportati di seguito. Questi medodi dovranno stampare solo il proprio nome per testimoniare la loro chiamata.

```
Car
    drive()
    brake()
    accelerate()
```

Aggiungi i metodi astratti *getSpeed* e *getName* che obblighino le sottoclassi a fornire dei valori. Crea successivamente 2 sottoclassi che implementino questi metodi. Nel main, chiama tutti i metodi di entrambe le macchine per testarne il funzionamento.

## Esercizio 4

Sulla false riga dell'esercizio precedente, crea una classe astratta *Car* che abbia come metodi

```
abstract class Car
    void turnOn();
    void engineNoise();
```

Classi altrettanto astratte dovranno implementare il metodo *engineNoise* di *Car*: una Fiat e una Audi.
Non puoi ancora istanziare ne Fiat ne Audi poiché sono ancora classi astratte. Crea un modello d'auto per queste sottoclassi che implementino il metodo *turnOn*, stampando a schermo una stringa nel formato

```
<nome modello> has started
```

Il main dovrà istanziare un oggetto per ogni modello e testarne il corretto funzionamento.

## Esercizio 5

Un pasticciere deve soddisfare le richieste dei propri clienti.
Se il programma viene lanciato senza parametri, il pasticciere deve elencare ciò che ha in bottega. Se invece il programma viene lanciato con dell'input, dovrà soddisfare la richiesta del cliente, comunicando la lista dei dolci scelti e il totale.
Ogni dolce dovrà avere un id, una stringa con il proprio nome e un prezzo, come nello schema

```
Dolce
    int getId()
    String getName()
    float getPrice()
```

La lista del cliente sarà quindi una lista di interi rappresentanti i dolci.
Crea una classe astratta *Dolce* implementata da una classe *Torta*. Questa dovrà avere una sottoclasse *Torta al cioccolato* che costi un po' di piu'. Poi crea una classe astratta *Pasticcino* implementata da *Pasticcino alla Crema* e *Pasticcino al Pistacchio*.
Usando il paradigma della *Factory Method* crea una classe che restituisca il dolce giusto in base all'id fornito.