Laboratorio di Programmazione

ESAME del 25 Novembre 2016

Contents

Avvertenze

- Nello svolgimento dell'elaborato è possibile usare qualunque classe delle librerie standard di Java.
- Non è invece ammesso l'uso delle classi del package `prog` allegato al libro di testo del Prof. Pighizzini e impiegato nella prima parte del corso.
- Si consiglia CALDAMENTE l'utilizzo dello script "checker.sh" per compilare ed effettuare una prima valutazione (NON esaustiva!) del proprio elaborato. Si consiglia anche di leggere il sorgente dei `Test_*.java` per capire cosa devono offrire le classi da sviluppare.
- Ricordarsi, quando si programma: *Repetita NON iuvant* o DRY (*Don't Repeat Yourself*).
- Per la procedura di **consegna** si veda in fondo al documento.

1 ESERCIZIO FILTRO

===>>> INIZIARE PRIMA CON QUESTO, se non si è in grado di portare a termine questo esercizio... NON PROSEGUIRE!!! (la correzione del resto dell'elaborato è subordinata alla correttezza di questo primo esercizio)

La successione di Fibonacci $f_1, f_2, f_3, ..., f_n, ...$ è una successione di numeri naturali così definita.

- $f_1 = 1$.
- $f_2 = 1$.
- Per ogni $n >= 3$, $f_n = f_{n-1} + f_{n-2}$

Ecco i primi termini della successione: $1, 1, 2, 3, 5, 8, 13, 21, ...$
Scrivete un programma che visualizzi una rappresentazione "grafica" della magnitudine dei primi n termini della successione di Fibonacci. Il programma è costituito da una sola classe di nome *Fibonacci*, contenente il metodo *main*, assieme eventualmente a dei metodi ausiliari statici (a discrezione). Il programma legge il numero intero n dalla lista degli argomenti del programma. Se la condizione $n >= 1$ non è soddisfatta, il programma termina. Poi, il programma visualizza n righe, ciascuna costituita esclusivamente da asterischi consecutivi. La riga i-esima dovrà contenere esattamente f_i asterischi, per ogni $i = 1, ..., n$. Ciò fatto, il programma termina. Ad esempio il comando `java Fibonacci 4` deve generare il seguente output:

```
*
*
**
***
```

E null'altro!

2 Tema d'esame

Lo scopo dell'esercizio è realizzare un modello semplificato per la gestione di strade a pedaggio e non. Quindi avremo una gerarchia di strade (Strada, Autostrada) e una gerarchia di veicoli (Veicolo, Auto, Camion) e le classi dovranno esporre metodi per il calcolo del pedaggio, della velocità media, etc.
Le **classi** da realizzare sono le seguenti (dettagli nelle sezioni successive):

1. `Strada`: strada generica, non prevede pedaggio
2. `Autostrada`: sottoclasse di Strada, introduce il concetto di "veicolo ammesso al transito" (deve avere una certa potenza) e di pedaggio
3. `Biglietto`: biglietto di ingresso, tiene traccia dell'orario di ingresso
4. `Veicolo`: veicolo generico, classe astratta
5. `Auto`: sottoclasse di Veicolo
6. `Camion`: sottoclasse di Veicolo
7. `Main`: programma principale

2.1 Specifica delle classi

Le classi (**pubbliche**!) dovranno esporre almeno i metodi e costruttori **pubblici** specificati, più eventuali altri metodi e costruttori se ritenuti opportuni. Gli attributi (campi) delle classi devono essere **privati**. per leggere e modificarne i valori, creare opportunamente, e solo dove necessario, i metodi di accesso (`set` e `get`). Se si usano classi che utilizzano tipi generici, si suggerisce di utilizzarne le versioni opportunamente istanziate (es. `ArrayList<String>` invece di `ArrayList`). Ogni classe deve avere il metodo `toString` che rappresenti lo stato delle istanze.

2.1.1 public class Strada

Deve definire gli attributi: `int limite (Km/h); float lunghezza (Km).`
Deve definire almeno un costruttore che permetta di impostare gli attributi.
E i seguenti metodi (oltre ai get per gli attributi, NON implementare i set, basta il costruttore):

- `public Strada(float lunghezza, int limite)` *per avere l'ordine dei parametri*
- `public float orePercorrenzaVelocitaCodice()` restituisce quante ore ci vogliono a percorrere tutta la lunghezza della strada andando alla velocità massima possibile (limite di velocità). Nota: il float restituito rappresenta ore e frazioni di ore, ad esempio 1.20 rappresenta un'ora e 12 minuti.
- `public float velocitaMediaDatoTempoPercorrenzaInSec(float percorrenza)` restituisce la velocità media in Km/h tenuta, dato un tempo di percorrenza effettivo espresso in secondi.
- `public boolean superatoLimiteDatoTempoPercorrenzaInSec(float percorrenza)` restituisce *true* se, dato il tempo di percorrenza (in secondi), si può supporre che il limite di velocità sia stato superato.

2.1.2 public class Autostrada

Sottoclasse di Strada, aggiunge la gestione del pedaggio, la gestione dell'accesso (accedono solo i veicoli sopra una certa potenza) e la gestione degli ingressi (tiene traccia di chi è dentro, calcola se sono stati superati i limiti, etc.).
Deve definire gli attributi: `float tariffaBase (euro/Km)`, `float potenzaMinimaPerAccedere (kW)` e un container (a scelta) per tenere traccia dei veicoli entrati.
Deve definire almeno un costruttore che permetta di impostare gli attributi.
E i seguenti metodi (oltre ai get per gli attributi, NON implementare i set, basta il costruttore):

- `public Autostrada(int lunghezza, int limite, float tariffaBase, float potenzaMinimaPerAccedere)` *per avere l'ordine dei parametri*
- `public float pedaggio(Veicolo v)` accetta solo veicoli con potenza superiore a potenzaMinima (se il veicolo non ha sufficiente potenza lancia eccezione) e calcola il pedaggio (in euro) in funzione di:
  - tariffa base
  - lunghezza della strada
  - numero di assi del veicolo: fino a 3 assi si usa la tariffa base, oltre i 3 assi si usa 1.5*(tariffa base)
- `public Biglietto ingresso(Veicolo v)` se il veicolo è ammesso (controllo su potenza), lo lascia entrare ed emette un biglietto (vedi classe relativa) per il veicolo stesso; lancia un'eccezione se il veicolo non è ammesso. Nota: java.lang.System.currentTimeMillis() permette di avere l'ora esatta espressa in millisecondi.
- `public float uscita(Veicolo v)` se il veicolo non è presente tra i veicoli entrati, lancia un'eccezione; se il tempo di percorrenza (calcolato in base all'ora attuale e all'ora di ingresso) è inferiore a quello "legale" (rispettando il limite di velocità), lancia un'eccezione, altrimenti fa uscire il veicolo (elimina dal container), calcola il pedaggio e lo restituisce.
- `public int quantiVeicoli()` restituisce il numero di veicoli attualmente in viaggio.
- `public float potenzaMedia()` calcola e restituisce la potenza media dei veicoli attualmente in viaggio.

2.1.3 public abstract class Veicolo

Deve definire gli attributi: `String targa; int assi (numero di); float potenza (in kW)`.
Inoltre deve avere un reference a Biglietto (vedi classe relativa) con i metodi set e get relativi.
`public Veicolo(String targa, int assi, float potenza)` *per ordine parametri*

2.1.4 public class Auto

Sottoclasse di Veicolo, non ha attributi aggiuntivi, implementa un costruttore che fissa il numero di assi a 2.
`public Auto(String targa, float potenza)` *per ordine parametri*

2.1.5 public class Camion

Sottoclasse di Veicolo, non ha attributi aggiuntivi, implementa un costruttore che fissa il numero di assi a 5 e un costruttore con numero di assi parametrico.
`public Camion(String targa, float potenza)`
`public Camion(String targa, float potenza, int assi)` *per ordine parametri*

2.1.6 public class Biglietto

Deve definire un attributo `long timestamp` (per registrare l'orario di ingresso) e avere un reference a Veicolo, coi relativi set e get.
Suggerimento: per il timestamp usare `java.lang.System.currentTimeMillis()`, che restituisce l'ora corrente espressa in millisecondi.

2.1.7 public class Main

Creare un main che realizzi i seguenti:

1. istanzi una Autostrada (attributi: 50 Km lunghezza, 100 Km/h limite vel., 0.20 euro/Km pedaggio, 50 kW potenza minima per accedere)
2. istanzi una serie di veicoli prendendo i dati da un file di testo (vedi sotto)
3. li immetta uno a uno nell'autostrada
4. stampi la situazione dell'autostrada dopo ogni ingresso
5. calcoli la potenza media dei veicoli che sono effettivamente entrati
6. li faccia uscire uno a uno dall'autostrada, stampando la situazione dopo ogni uscita

Formato del file di testo: TipoVeicolo;Targa;Potenza
Utilizzare il seguente contenuto:

```
Camion;MI2121212;600
Auto;PV77777;60
Camion;CO98989;900
Auto;MI656565;200
Auto;MI43432;40
Camion;MI2323232;600
```

(potete usare il file `autostrada.txt` allegato al tema)

---

3 Raccomandazioni

Affinché l'elaborato sia valutato, è richiesto che sia le classi sviluppate che i test risultino *compilabili*. A tal fine i metodi/costruttori che non saranno sviluppati dovranno comuque avere una implementazione fittizia come la seguente:

```
public void nomeDelMetodo () {
   throw new UnsupportedOperationException();
}
```

Si suggerisce quindi di dotare da subito le classi di tutti i metodi richiesti, implementandoli in modo fittizio, e poi di sostituire man mano le implementazioni fittizie con implementazioni che rispettino le specifiche.

---

4 Consegna

Si ricorda che le classi devono essere tutte *public* e che vanno consegnati tutti (e soli) i file *.java* prodotti. NON vanno consegnati i *.class*. NON vanno consegnati i file relativi al meccanismo di autovalutazione (*Test_*.java, AbstractTest.java, *.sh*). Per la consegna, eseguite l'upload dei SINGOLI file sorgente (NON un file archivio!) dalla pagina web: http://upload.di.unimi.it nella sessione del vostro docente.

*** ATTENZIONE!!! ***

NON VERRANNO VALUTATI GLI ELABORATI CON ERRORI DI COMPILAZIONE O LE CONSEGNE CHE NON RISPETTANO LE SPECIFICHE (ad esempio consegnare un archivio zippato è sbagliato, come anche consegnare ad un docente diverso dal proprio assegnato).
UN SINGOLO ERRORE DI COMPILAZIONE O DI PROCEDURA INVALIDA **TUTTO** L'ELABORATO.

Per ritirarsi fare l'upload di un file vuoto di nome `ritirato.txt`. Se avete caricato dei file nella sessione del docente sbagliato, caricate lì un file vuoto di nome `erratoConsegna.txt` e caricate poi i file nella sessione giusta.
