Ingegneria del Software

Esercitazione 2

Gara Canora

Definire la classe Singer e Competition

Specifiche Singer:

- Un cantante ha un nome, cognome, luogo e data di nascita e un titolo di canzone
- Deve essere possibile comparare alfabeticamente i cantanti (per cognome e in caso di omonimia si usa il nome)
- Due cantanti sono uguali se tutti i loro dati coincidono

Gara Canora

Definire le classi Singer e Competition

***Specifiche Competition**:*

- Una competizione è composta da una classifica di 10 cantanti
- Una competizione ha inoltre un nome, un luogo, un presentatore, un data di inizio e fine (tutti elementi stringa)
- Metodo *addSinger(Singer singer)*: aggiunge un cantante alla classifica (in coda se non vuota)
- Metodo *addSinger(Singer singer, int atPosition):* aggiunge un cantante alla classifica alla posizione specificata, gli altri cantanti scalano (e di conseguenza uno può uscire dalla classifica)
- Metodo *deleteSinger(Singer singer):* elimina il cantante dalla competizione se esiste
- Metodo *deleteSinger(String surname, String name)*: elimina un cantante dalla competizione con nome e cognome specificato
- Metodo *generateRank()*: ritorna la rappresentazione testuale della classifica
- Metodo *generateOrderedList():* ritorna una lista testuale alfabetica dei cantanti in classifica

Complex (1)

Definire una classe per la gestione di numeri complessi

Specifiche:

- Un numero complesso è identificato da due numeri *real* e *imaginary* di tipo double
- Dato un numero complesso $z = x + yi$, definire le operazioni di *modulo* e *fase*

$$r = |z| = \sqrt{x^2 + y^2}.$$

$$\varphi = \arg(z) = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan(\frac{y}{x}) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{indeterminate} & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

Complex (2)

Definire una classe per la gestione di numeri complessi

Specifiche:

- Dati due numeri complessi definire le operazioni di somma, sottrazione, moltiplicazione e uguaglianza.

$$(a + bi) + (c + di) = (a + c) + (b + d)i.$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i.$$

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i.$$

$$z_1 = z_2 \;\leftrightarrow\; (\mathrm{Re}(z_1) = \mathrm{Re}(z_2) \wedge \mathrm{Im}(z_1) = \mathrm{Im}(z_2)).$$

Strings

Cosa stampa questo programma?

```
public class StringDemo {
 public static void main(String[] args){
  String s1 = “Guess who”;
  String s2 = s1;
  s1 = s1 + “ is back”;
  System.out.println(s1);
  System.out.println(s2);
 }
}
```

Strings are Immutable

Risposta:

> Guess who is back?

> Guess who

- `s2` punta a `s1`
- `s1` concatenato con un'altra stringa genera un nuovo oggetto
- `s2` continua a puntare all'oggetto precedente

Persons and Students

Definire le classi Person, Student e Grade

Specifiche:

- Una persona ha un nome, cognome e una data *(java.util.Date)*
- Uno studente è una persona con un id e una lista di voti
- Un voto contiene punteggio e crediti
- Lo studente espone due funzionalità:
  - Calcolo media pesata
  - Controllo se è possibile che si laurei (crediti totali >= 180)

Collections and Generics

```
List<Grade> grades = new ArrayList<Grade>()
```

- Questo è un esempio dell'uso dei *generics*, classi il cui tipo è parametrico rispetto ad altri tipi.
- List è una interfaccia, per creare un oggetto ci serve una classe concreta. Nell'esempio ArrayList

Comparable Interface

Implementare la classe SortAlgorithms

***Specifiche**:*

- Un solo metodo statico *sort* che ordina un array di oggetti che implementano l'interfaccia *java.lang.Comparable*
- Person implementa *Comparable* controllando l'ordine del cognome e poi, in caso di omonimia, il nome. Student aggiunge a questo comportamento in caso di omonimia sia sul nome che sul cognome il controllo sull'id.

Access Modifier

Completare il codice con gli opportuni modificatori di visibilità in modo tale che l'accesso alle variabili e ai metodi sia il più ristretto possibile ma che non crei errori di compilazione

Memento: Access Modifier

`public`	visibile da qualsiasi parte del programma
`private`	visibile solo dall'interno della classe stessa
`protected`	visibile solo dalle classi dello stesso package e dalle sottoclassi
`default`	visibile dallo stesso package e dalle sottoclassi se sono nello stesso pacchetto.

```
package a;
... class First {
  ... int x;
  ... int y;
  ... void h() { y = -1; }
}
... class Second extends First {
  ... void f(int x) { this.x = x; h(); }
}

package b;
imports a.*;
class Third {
  public static void main(String[] s) {
    Second z = new Second();
    z.f(3);
}

class Fourth extends First {   void g(void) { h(); }  }
```

```
package a;
public class First {
  int x;  // default
  private int y;
  protected void h() { y = -1; }
}
public class Second extends First {
  public void f(int x) { this.x = x; h(); }
}

package b;
imports a.*;
class Third {
  public static void main(String[] s) {
    Second z = new Second();
    z.f(3);
}
class Fourth extends First {   void g(void) { h(); }  }
```

Access Modifier (2)

Questo codice è corretto?
Che output produce una chiamata al metodo m1 di C2?

```
package A;
public class C1 {
    public void m1() { }
    protected void m2() { }
    private void m3() { }
}

package B;
import A.*;
public class C2 extends C1 {
    public void m1() { System.out.print("Hello"); m2(); m3(); }
    protected void m2() { System.out.print(“ World”); }
    private void m3() { System.out.print("!"); }
}
```

Access Modifier (2)

Risposta:

È corretta.

Una chiamata a m1 di C2 stampa "Hello World!"

C2 non vede la definizione di *m3* data da *C1*, perché questa è *private*. Pertanto la definizione di *m3* in *C2* è, banalmente, la definizione di un nuovo metodo. Inoltre *C2* ridefinisce i metodi *m1* e *m2* ereditati da *C1*.

Access Modifier (2)

E questa definizione di C2 è corretta? Perché?

```
public class C2 extends C1 {
    public void m1() {
      System.out.print("Hello");
      m2();
      m3();
   }
    protected void m2() {
      System.out.print(“ World");
    }
}
```

Access Modifier (2)

Risposta:

È scorretta.

Il metodo m3 è definito private nella classe C1 e pertanto non è visibile in C2. Si ottiene quindi un errore a compile-time.

Access Modifier (2)

E questa definizione di C2 è corretta? Perché?

```
public class C2 extends C1 {
    public void m1() {
      System.out.print("Hello");
      m2();
      m3();
   }
    private void m3() {
      System.out.print(“!”);
    }
}
```

Access Modifier (2)

Risposta:

È corretta.

Il metodo m3 è definito nella classe C2 e il metodo m2 è definito protected in C1 quindi visibile in C2. Una chiamata ad m1 stampa "Hello!"

Static vs Dynamic Types

Sia dato il seguente frammento di codice.

Indicare gli errori a compile-time.

Eliminare le istruzioni che generano errore a compile-time, e dire se il codice genera errori a runtime.

Eliminare anche le istruzioni che generano errore a runtime, e dire cosa produce in output il programma.

Static vs Dynamic Types

```
package C;

public class C3 {
  public static void main(String[] s) {
    C1 c1;    C2 c2;  Object o;
    c1 = new C1();  /*1*/
    c1.m1();        /*2*/
    c2 = new C2();  /*3 */
    c2.m2();        /*4 */
    c1 = c2;        /*5 */
    c1.m1();        /*6 */
    c2 = new C1();  /*7 */
    o = new C1();   /*8 */
    c2 = (C2) o;    /*9 */
    o = new C2();   /*10 */
   c1 = (C1) o;    /*11 */
    c1.m1();        /*12 */
  }
}
```

```
package A;
public class C1 {
    public void m1() { }
    protected void m2() { }
    private void m3() { }
}

package B;
import A.*;
public class C2 extends C1 {
    public void m1() { }
    protected void m2() { }
    private void m3() { }
}
```