Programmazione a Oggetti

Metodologie di Programmazione

24 Gennaio 2012

Nome: _____ Matricola: _____

Metodologia di Programmazione [ ]
Programmazione a Oggetti [ ]

Istruzioni

- Scrivete il vostro nome sul primo foglio.
- Indicare se la vostra prova è riferita all'esame MP o PO
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- **Gli esercizi 9 e 10 sono rivolti a coloro che non hanno svolto le esercitazioni, ovvero le hanno svolte ottenendo una valutazione insufficiente / insoddisfacente. Per chiunque sceglga di svolgerli, il punteggio su questi esercizi cancella il punteggio ottenuto nelle esercitazioni.**

LASCIATE IN BIANCO:

1	2	3	4	5	6	7	8	9	10	TOT

1 Esercizio

Consideriamo la definizione della classe `Point` qui di seguito.

```
class Point
{
    private double x;
    private double y;
    public Point(double x, double y) { this.x = x; this.y = y; }

    public boolean equals (Point p)  { return p.x == x && p.y == y; }









}
```

La definizione dovrebbe assicurare che, dato un riferimento `p:Point`, l'invocazione `p.equals(q)` restituisca `true` se e solo se `q` è un `Point` con coordinate uguali a quelle di `p`. Modificate e/o completate la definizione della classe in modo da rispettare questa specifica. [3pt]

Supponete di essere il compilatore Java, e di trovarvi a compilare le seguenti classi. Per ciascun comando del metodo `main` indicate se genera un errore di compilazione e, in caso non lo faccia, descrivete cosa stampa in esecuzione. Per la descrzione del comportamento in esecuzione, assumete che tutti gli statement che generano errori di compilazione vengano rimossi. [3pt]

```
class Foo
{
    protected void print(String s) { System.out.println(s); }
    public void test( int x )      { print("Foo.test(int)"); }
}
class Bar extends Foo
{
    public void test( int x )    { print("Bar.test(int)"); }
    public void test( double d ) { print("Bar.test(dbl)"); }
}
class Test
{
    public static void main(String[] args)
    {
        Foo a = new Foo( );
        Foo b = new Bar( );
        Bar c = new Bar( );
        a.test( 1 );
        a.test( 2.0 );
        b.test( 3 );
        b.test( 4.0 );
        c.test( 5 );
        c.test( 6.0 );
     }
}
```

2 Esercizio

Considerate la seguente definizione dell'interfaccia `List<E>`.

```
public interface List<E>
{
    // restituisce il numero di elementi nella lista
    int size();
    // restituisce una nuova lista ottenuta aggiungendo e a this
    List<E> add(E e);
}
```

Completate il codice delle due classi `EmptyList<E>` (la lista sempre vuota) e `NonEmptyList<E>` (una lista sempre non vuota) che implementano l'interfaccia `List<E>`.

```
public class EmptyList<E> implements List<E>
{
    private static final EmptyList<?> singleton = new EmptyList();

    public static <E> List<E> getSingleton()  { return (List<E>) singleton;   }









}
```

[2pt]

```
public class NonEmptyList<E> implements List<E>
{
    private E first;
    private List<E> rest;

    public NonEmptyList(E f, List<E> r)
    {
       first = f; rest = r;
    }









}
```

[2pt]

3 Esercizio

Considerate le seguenti classi che descrivono alcuni prodotti in vendita in un supermercato.

```
class Clothing
{
  private int size;
  private double price;
  ...
  public double price(){ return price; }
}
class Toy
{
  private int age;
  private double price;
  ...
  public double price(){ return price; }
}
```

Definite un nuovo tipo `ItemForSale` che descriva in modo uniforme i diversi prodotti in vendita. Utilizzate il nuovo tipo per completate il codice della classe `Store` descritto qui di seguito. [4pt]

```
// classe Store  inizialmente senza prodotti
 public class Store
 {
      private ................ catalog  =  ...................;


      // Aggiunge un prodotto al catalogo
      public void add(ItemForSale i)
      {

      }

      // Restituisce tutti i prodotti in catalogo
      public Iterator<ItemForSale> items()
      {



      }

      // Restituisce il prezzo totale degli items in catalogo
      public double total()
      {



      }
 }
```

4 Esercizio

Considerate la classe seguente:

```
class Position
{
     private double x;
     private double y;
     public Position(double x, double y) {this.x=x; this.y=y;}
     public double getX() {return x;}
     public double getY() {return y;}
}
```

Che cosa sbagliato nella codice seguente? [1pt]

```
class Position3D extend Position
{
    private double z;
    public Position3D(double x, double y, double z)
    {
       this.x = x; this.y = y; this.z = z;
    }
    public double getZ() { return z;}
}
```

Correggete il codice della classe `Position` lasciando intatto il codice di `Position3D`. [1pt]

Correggete il codice della classe `Position3D` lasciando intatto il codice di `Position` [2pt]

5 Esercizio

Considerate la seguente rappresentazione degli insiemi.

```
class Set<E>
{
     private TreeSet<E> contents;
     public Set() { this.contents = new  TreeSet<E>(); }
     public Iterator<E> getContents(){ return contents.iterator(); }
     public void add(E val)
     { if (!contents.contains(val)) contents.add(val); }
     public void remove(E val){ contents.remove(val); }
}
```

Completate la definizione della classe `MaxSet<E>` seguendo le indicazioni date nei commenti. [3pt]

```
class MaxSet<E extends Comparable<E>> extends Set<E>
{
    private E max;  // massimo dell'insieme
    // costruisce un MaxSet inizializzando opportunamente max
    public MaxSet()
    {


    }
    // aggiunge il nuovo valore a this, aggiornando il campo
    // max nel caso il nuovo valore sia il nuovo massimo
    public void add(E val)
    {




    }
    // rimuove una occorrenza di val da this, aggiornando il
    // max in tutti i casi in cui sia necessario
    public void remove(E val)
    {









    }
}
```

Ricordiamo qui di seguito la specifica dell'interfaccia `Comparable<E>`.

```
interface Comparable<E>
{
  public int compareTo(E e)
  // confronta this con e restituendo un intero negativo, zero, o un intero
  // positivo se, rispettivamente, this e` minore, uguale o maggiode di e
}
```

6 Esercizio

Considerate la seguente definizione.

```
public class Tree<E>
{
    E value;
    Tree<E> left = null;
    Tree<E> right= null;

    public Tree(E e)
    {
       value=e;
    }
    public void setLeft(Tree<E> t)
    {
        left=t;
    }
    public void setRight(Tree<E> t)
    {
        right=t;
    }

    public void visit()
    {
       System.out.println(value.toString());
       if (left!=null)
          left.visit();
       if (rigth!=null)
          rigth.visit();
    }
}
```

Riformulate la classe `Tree` applicando il design pattern *Null Object*. [3pt]

7 Esercizio

Dato il seguente codice:

```
public class Point {
   private double x;
   private double y;

   public Point(double x, double y){
     this.x=x;
     this.y=y;
   }

  public Point(double d, double a){
     x = d * Math.cos(a);
     y = d * Math.sin(a);
  }
}
```

Correggete l'errore (quale è?) utilizzando il Design Patter Factory Method. [3pt]

8 Esercizio

La seguente classe astratta descrive la struttura degli elementi di sistema a finestre.

```
abstract class Window
{
   private int width, height;
   protected Window(int w, int h) {  width =  w; height = h;   }
   protected abstract boolean resizable();
   public void resize(double dx, double dy)
   {
     if (resizable()) {
         width  = (int) (width * dx);  height = (int) (height * dy);
     }
   }
}
```

Una `Window` ha una dimensione (altezza e larghezza), un metodo che informa se è possibile o meno ridimensionarla, ed un metodo che permette di ridimensionarla di un valore percentuale rispetto alla dimensione corrente. Progettiamo un sistema a finestre che includa solo due tipi di componenti: `Panels` e `Labels` descritti come segue. Completate le definizioni delle due classi, aggiungendo tutte le definizioni di campi e/o metodi che ritenete necessari oltre a quelli richiesti.

```
// Una  Window ridimensionabile che funge da contenitore di Panels o Labels
class Panel extends Window
{

  // Ridimensiona il pannello e tutte le componenti in esso contenute
  public void resize()
  {


  }

  // Aggiunge w a this
  public void add(Window w)
  {

  }









}
```

[2pt]

[continua alla pagina successiva]

```
// Una Window, non resizable, con un elemento che descrive l'etichetta
  class Label extends Window
  {

    // Modifica l'etichetta di this, assegnandole il valore s
    public void setLabel(String s) { ..................................... }

    // Restituisce il valore corrente dell'etichetta
    public String get()  { ..................................... }

  }
```

[2pt]

9 Esercizio

Assumete data la seguente definizione:

```
interface Cond { public boolean cond(); }
```

Definite il seguente metodo di utilità per collections. [punteggio: vedi nota a pag 1]

```
/**
 * Dato l'iteratore it, restituisce la collection costituita dagli
 * elementi ottenuti da it che soddisfano il predicato cond()
 */

public static <E extends Cond> Collection<E> select(Iterator<E> it)
{









}
```

10 Esercizio

Considerate la classe seguente:

```
class Repository<T>
{
     private T contents;
     public Repository(T contents)  { this.contents = contents; }
     public T getContents()         { return contents;           }
     public void setContents(T val) { contents = val;            }
}
```

Completate la definizione della classe `RCSRepository<T>` seguendo le indicazioni date nei commenti. [punteggio: vedi nota a pag 1]

```
// Un Revision Control System Repository, che tiene traccia
// delle versioni e permette di ripristinare una qualunque
// versione antecedente a quella presente. Ogni versione e'
// identificata univocamente da un numero.
class RCSRepository<T> extends Repository<T>
{


    // Costruttore: costruisce la versione 0 del repository
    public RCSRepository(T contents)
    {



    }

    // Sia n il numero della versione corrente. Salva la
    // versione corrente e costruisce la versione n+1,
    // che ha val come nuovo contenuto
    public void setContents(T val)
    {



    }

    // ripristina il valore di contents corrispondente alla
    // versione identificata dal ``version number'' vno, se
    // una tale versione esiste, altrimenti eccezione
    // Tutte le versioni, da quella corrente a quella
    // identificata da vno+1 vengono perse.
    public void revert(int vno) throws NoSuchVersionException
    {




    }
}
```