Programmazione ad Oggetti – Modulo A (Esame del 11/9/2015)

Esercizio 1

Considerate la seguente gerarchia di classi:

```
class A {
public void print(String s) { System.out.println(s); }
public void m1() { print("A.m1"); m2(); }
public void m2() { print("A.m2"); }
}
class B extends A {
public void m2() { print("B.m2"); }
public void m3() { print("B.m3"); }
}
class C extends A {
public void m1() { print("C.m1"); }
public void m2() { print("C.m2"); m1(); }
}
class D extends C {
public void m1() { super.m1(); print("D.m1"); }
public void m3() { print("D.m3"); }
}
```

siano inoltre date le seguenti definizioni:

```
A var1 = new B();  A var2 = new D();
B var3 = new B();  C var4 = new C();
C var5 = new D();  Object var6 = new C();
```

Quale è il risultato della compilazione e della (eventuale, nel caso la compilazione non dia errori) esecuzione dei seguenti frammenti?

1. `var1.m1();`
2. `var2.m1();`
3. `var3.m1();`
4. `var4.m1();`
5. `var5.m1();`
6. `var6.m1();`
7. `((B)var1).m3();`
8. `((B)var2).m3();`
9. `(D)var5).m3();`
10. `((D)var6).m3();`

Soluzione Esercizio 1

1. `var1.m1();` `A.m1 / B.m2`
2. `var2.m1();` `C.m1 / D.m1`
3. `var3.m1();` `A.m1 / B.m2`
4. `var4.m1();` `C.m1`
5. `var5.m1();` `C.m1 / D.m1`
6. `var6.m1();` compiler error
7. `((B)var1).m3();` `B.m3`
8. `((B)var2).m3();` runtime error
9. `(D)var5).m3();` `D.m3`
10. `((D)var6).m3();` runtime error

Esercizio 2

Vogliamo costruire un sistema di classi per gestire le operazioni di un file system. La seguente classe astratta descrive la struttura comune degli elementi del file system.

```
abstract class FSItem {
  private String nome;
  // diritti
public class Permessi {
    public boolean rr;
    public boolean ww;
}
protected Permessi permessi = new Permessi();
  protected FSItem(String s) {nome = s; permessi.rr = true; permessi.ww = true; }
  public String name() { return nome; }
  public void chmod (boolean r, boolean w) { permessi.rr = r; permessi.ww = w; }
}
```

Un `FSItem` ha un nome, due booleani che definiscono i diritti di lettura e scrittura, ed il metodo `chmod()` per modificarli. Le diverse componenti del filesystem sono descritte dalle classi specificate qui di seguito.

```
class Directory extends FSItem {
  // OVERVIEW: una directory e un FSItem che puo contenere
  // un insieme di sotto-directories, files o links

  public string ls() throws IllegalOperation
  // Se this permette la lettura, restituisce una stringa che
  // contiene i nomi di tutti gli FSItems contenuti nella
  // directory; altrimenti lancia leccezione

  public void add(FSItem i)
  // Aggiunge i a this se this permette la scrittura;
  // Altrimenti lancia leccezione
}

class File extends FSItem {
  // OVERVIEW: un File e‘ un FSItem con un contenuto di tipo string

  public void set(string s) throws IllegalOperation
  // Se l’operazione di scrittura e permessa setta ad s il
  // contenuto del file; altrimenti lancia leccezione

  public string get()
  // Se l’operazione di lettura e‘ permessa restituisce il
  // contenuto del file; altrimenti lancia leccezione
}

abstract class Link extends FSItem {
  // OVERVIEW: un Link e‘ un FSItem con associato un altro FSItem,
  // detto il "target". Implementa tutte le operazioni del target,
  // e ciascuna operazione viene implementata mediante
  // linvocazione della stessa operazione sul target.
}

class FileLink extends Link {
  // OVERVIEW: un FileLink rappresenta il link ad un File
  // e implementa tutte le operazioni del File "target".
}

class DirectoryLink extends Link {
  // OVERVIEW: un DirectoryLink rappresenta il link ad una directory
  // e implementa tutte le operazioni della Directory "target".
}
```

Fornite l'implementazione completa delle classi `Directory`, `File`, `Link`, `FileLink` e `DirectoryLink`.

Soluzione Esercizio 2

```
class Directory extends FSItem {
  private ArrayList<FSItem> items = new ArrayList<FSItem>();

  public Directory(String name) { super(name); }

  public String ls() throws IllegalOperation {
    if (!permessi.rr) throw new IllegalOperation();
    String result = "";
    for (FSItem i : items) {
      result =  result + i.name() + "\n";
    }
    return result;
  }

  public void add(FSItem i)throws IllegalOperation {
    if (!permessi.ww) throw new IllegalOperation();
    items.add(i);
  }
}


class File extends FSItem {
  private String content;

  public File(String name) { super(name); }

  public void set(String s) throws IllegalOperation {
    if (!permessi.ww) throw new IllegalOperation();
    content=s;
  }

  public String get() throws IllegalOperation {
    if (!permessi.rr) throw new IllegalOperation();
    return content;
  }
}


abstract class Link extends FSItem {
  protected FSItem target;

  public Link(FSItem t, String nome) {
    super(nome);
    target=t;
    permessi=t.permessi;
  }
}


class FileLink extends Link {
  protected File target;

  public FileLink(File f, String nome) {
    super(f,nome);
    target=f;
  }

  public void set(String s) throws IllegalOperation {
```

```
    target.set(s);
  }

  public String get() throws IllegalOperation {
    return target.get();
  }
}

class DirectoryLink extends Link {
  protected Directory target;

  public DirectoryLink(Directory f, String nome) {
    super(f,nome);
    target=f;
  }

  public String ls() throws IllegalOperation {
    return target.ls();
  }

  public void add(FSItem i) throws IllegalOperation {
    target.add(i);
  }
}
```

Esercizio 3

Considerate linterfaccia la seguente interfaccia che descrive una generica lista di `T`.

```
public interface List<T>
{
  boolean empty();
  List<T> zip(List<T> l);
}
```

Definite il codice di due classi `MTList<T>` (la lista sempre vuota) e `NEList<T>` (una lista sempre non vuota) che implementano linterfaccia `List<T>` e realizzando i metodi `empty()` e `zip()`. In particolare, `empty()` restituisce `true` se la lista è vuota, `false` altrimenti, mentre `zip()` è definito come segue:

- se `l1:MTList<T>`, oppure `l2:MTList<T>`, allora `l1.zip(l2)` restituisce la lista la vuota;
- altrimenti, `l1.zip(l2)` costruisce la lista il cui i primi due elementi sono rispettivamente il primo emento di `l1` e il primo elemento di `l2`, e il resto è ottenuto dallo zip del resto di `l1` con il resto di `l2`.

Chiariamo il comportamento del metodo zip() con due esempi:

- se `l1 = [1,5,3]` e `l2 = [4,2]`, allora `l1.zip(l2) = [1, 4, 5, 2]`.
- se `l1 = []` allora `l1.zip(l2) = []` per qualunque `l2`.

Soluzione Esercizio 3

```
class MTList<T> implements List<T> {
  public boolean empty() { return true; }
  List<T> zip(List<T> l) { return new MTList<T>(); }
}

class NEList<T> implements List<T> {
  protected T head;
  protected List<T> tail;

  public NEList(T head, List<T> tail) {
    this.head=head;
    this.tail=tail;
```

```
  }

  public boolean empty() { return false; }

  List<T> zip(List<T> l) {
    List<T> result;
    if (l.empty()) {
      result=new MTList<T>();
    } else {
      NEList<T> ll=(NEList<T>)l;
      result = new NEList<T>(head, new NEList(ll.head, tail.zip(ll.tail)));
    }
    return result;
  }
}
```

Esercizio 4

Si scriva una gerarchia di classi che rappresenti una composizione arbitraria di resistenze in un circuito elettrico in termini di resistenze atomiche e composizioni in serie ed in parallelo di resistenze. La gerarchia dovrà offire un metodo polimorfico `double calcola()` che restituisce il valore della resistenza effettiva del circuito. Si ricordi che la resistenza effettiva R_e di due composizioni di resistenze messe in serie tra di loro e aventi resistenze effettive R_1 ed R_2, è $R_e = R_1 + R_2$. Per due composizioni di resistenze messe in paralleo vale $\frac{1}{R_e} = \frac{1}{R_1} + \frac{1}{R_2}$.

Soluzione Esercizio 4

```
interface Resistenza {
  double calcola();
}

class ResistenzaSingola implements Resistenza {
  private double valore;
  public ResistenzaSingola( double v ) { valore=v; }
  public double calcola() { return valore; }
}

class ResistenzeInSerie implements Resistenza {
  private Resistenza prima, dopo;
  public ResistenzeInSerie( Resistenza p, Resistenza d ) { prima=p; dopo=d; }
  public double calcola() { return prima.calcola() + dopo.calcola(); }
}

class ResistenzeInParallelo implements Resistenza {
  private Resistenza destra, sinistra;
  public ResistenzeInSerie( Resistenza dx, Resistenza sx ) { destra=dx; sinistra=sx; }
  public double calcola() { return 1.0/(1.0/destra.calcola() + 1.0/sinistra.calcola()); }
}
```

Esercizio 5

State scrivendo un programma gestionale e dovete gestire l'anagrafe delle persone legali con cui la ditta ha rapporti. Una *persona legale* è idetificata da una *partita iva* (una stringa) e nei confronti della ditta possono essere *clienti* o *fornitori*.
Per i clienti possiamo ottenere lo storico delle vendite (nella forma di un `Iterator<Vendite>`), mentre dai fornitori possiamo ordinare nuovi prodotti (`void ordina(Prodotto p, int quantita)`).
La stessa persona legale può essere sia cliente che fornitore e può essere registrato nel sistema con un ruolo, mentre il secondo ruolo può essere aggiunto sucessivamente. Cioè una persona legale può essere registrata nel sistema come cliente e sucessivamente diventare anche fornitore, o essere registrato come fornitore e sucessivamente diventare cliente. Una volta che una persona legale ha assunto un ruolo lo mantiene per sempre.
Scrivete il codice Java per implementare le entià ed il loro comportamento (lasciando in bianco l'implementazione dei metodi). Motivate le scelte nella strutturazione delle classi ed interfacce utilizzate, indicandone vantaggi e limiti.