Parte I – Java

Problema 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"); }
}
```

e assumete le seguenti dichiarazioni di variabile.

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

Nella tabella seguente, indicate nella colonna di destra l'output prodotto dal comando riportato nella tabella di sinistra. Se il comando produce più di una linea di output, utilizzate il carattere '/' per indicare le diverse linee (ad esempio: `a/b/c` indica tre linee di output, con `a`, `b`, e `c`). Se il comando causa errore, indicate nella colonna il tipo di errore, indicando `errore di compilazione` oppure `errore run-time`.

`var1.m1();`	`A.m1 / B.m2`
`var2.m1();`	`C.m1 / D.m1`
`var3.m1();`	`A.m1 / B.m2`
`var4.m1();`	`C.m1`
`var5.m1();`	`C.m1 / D.m1`
`var6.m1();`	`compiler error`
`var1.m2();`	`B.m2`
`var2.m2();`	`C.m2 / C.m1 / D.m1`
`var3.m2();`	`B.m2`
`var4.m2();`	`C.m2 / C.m1`
`var5.m2();`	`C.m2 / C.m1 / D.m1`
`var6.m2();`	`compiler error`
`var3.m3();`	`B.m3`
`var5.m3();`	`compiler error`
`((B)var1).m3();`	`B.m3`
`((D)var4).m3();`	`runtime error`
`((D)var5).m3();`	`D.m3`
`((B)var2).m3();`	`runtime error`
`((C)var2).m2();`	`C.m2 / C.m1 / D.m1`
`((D)var6).m2();`	`runtime error`

Problema 2

Dato un tipo riferimento `T`, definite il corpo del metodo seguente

```
int iteraSullaLista(List l) {
  // dichiara un iteratore sulla lista e lo utiliza
  // per scorrere la lista l e restituire il numero di
  // elementi in l che hanno tipo T

  Iterator it = l.iterator();
  int count = 0;
  while (it.hasNext())
  if (it.next() instanceof T)  count++;
  return count;

}
```

Sia data la seguente dichiarazione di interfaccia

```
interface I {
   String m1() throws E1;
   String m2() throws E2;
}
```

dove `E1` ed `E2` sono due diverse sottoclassi `Exception`. Completate il corpo del metodo seguente:

```
String proteggiDalleEccezioni(I obj) {
  // restituisce la stringa s1 + s2 dove
  // s1 = obj.m1() se obj.m1() non lancia eccezioni
  // s1 = "m1 raises" altrimenti
  // e analogamente per s2.

  String s1, s2;
  try { s1 = obj.m1(); }
  catch (E1 e) { s1 = "m1 raises" ; }
  try { s2 = obj.m2(); }
  catch (E2 e) { s2 = "m2 raises" ; }
  return s1 + s2;
}
```

Parte II – Programmazione

Assumete data una classe `Stack` che realizza l'implementazione di una pila, con i seguenti metodi:

```
public Stack()
// costruisce uno stack vuoto
public void push(Object value)
// aggiunge un elemento sul top della stack
public Object pop()
// toglie l'elemento sul top e restituisce tale elemento
public boolean isEmpty()
// restituisce true se lo stack e` vuoto, false altrimenti
public int length()
// restituisce il numero di elementi sullo stack.
```

Dovete scrivere una classe `UndoStack` che estende le funzionalità della classe `Stack` fornendo una operazione di "undo". L'idea è che ciascuna operazione di push/pop può essere "undone" con una chiamata ad un metodo `undo()`. Se sullo stack sono state invocate una sequenza di push e/o pop, deve essere possibile eseguire una sequenza corrispondente di `undo()` che annulla l'effetto della operazione più recente di cui non è ancora stato richiesto l'undo. Se non sono state eseguite operazioni non c'è nulla di cui fare l'undo. Inoltre, ogni operazioni può essere "undone" una sola volta: quindi se tutte le operazioni sono state "undone" non c'e` nulla di cui fare undo.

Più precisamente, la nuova classe deve fornire i seguenti metodi in aggiunta a quelli forniti dalla classe `Stack`:

```
public void undo()
// fa' l'undo della push o pop piu` recente di cui non e' gia' stato
// fatto l'undo. Se non c'e' nulla di cui fare l'undo, non ha effetto

public boolean canUndo()
// restituisce true/false se esiste/non esiste una operazione di cui
// fare l'undo
```

Nota bene: non potete fare alcuna assunzione sull'implementazione della classe `Stack`, oltre quelle relative ai metodi, date all'inizio.

```
/*********************************************************************
 SOLUZIONE
 Costruiamo la classe UndoStack utilizzando due stacks. Uno stack
 contents dove teniamo il contenuto dello stack, ed uno stack undo
 dove teniamo traccia  delle operazioni di cui fare l'undo.
 *********************************************************************/

// OPERATIONS

interface Operation { void eval(Stack s); }

class Push implements Operation {
    private Object val;

    public Push(Object val) { this.val = val; }
    public void eval(Stack s) { s.push(val); }
}

class Pop implements Operation {
    public void eval(Stack s) { s.pop(); }
}

// CONTINUA ....
```

```
import java.util.*;

// UNDOSTACK
class UndoStack {
    // java.util definisce una classe stack con
    // l'interfaccia specificata nel testo

    Stack contents = new Stack();
    Stack undo = new Stack();

    public void push(Object value) {
        contents.push(value);
        undo.push(new Pop());
    }

    public Object pop() {
        Object el = null;
        if (!contents.isEmpty()) {
          el = contents.pop();
          undo.push(new Push(el))
        }
        return el;
    }

    public boolean isEmpty() { return contents.isEmpty(); }

    public int length() { return contents.size();   }

    public boolean canUndo() { return !undo.isEmpty(); }

    public void undo() {
        if (canUndo()) ((Operation)undo.pop()).eval(contents);
    }
    // solo per testare l'implementazione
    public void printContents() {
        System.out.println("----------------------\nStack contents");
        for (Enumeration e = contents.elements(); e.hasMoreElements();)
            System.out.print(e.nextElement()+" ");
        System.out.println();
    }
}
// UNA CLASSE PER TESTARE LA SOLUZIONE
class Stacks {
    // potete verificare che l'output e' quello atteso
    public static void main(String[] args) {
        UndoStack s = new UndoStack();
        s.push(new Integer(1)); s.push(new Integer(2));
        s.printContents(); // expect 1,2
        s.undo(); s.printContents(); // expect 1
        s.push(new Integer(3)); s.push(new Integer(4));
        s.printContents(); // expect 1,3,4
        s.pop(); s.printContents(); // expect 1,3
        s.undo();s.printContents(); // expect 1,3,4
    }
}
```

Parte III – Progetto

Dovete realizzare una applicazione per la stampa di documenti. I documenti possono essere di diversi tipi, `ASCII`, `PS`, `PDF`, ciascuno dei quali ha un suo specifico metodo di stampa `String print()`, di cui omettiamo l'implementazione, che converte il documento in una stringa che può essere stampata. Ogni documento, inoltre, ha due metodi `String autore()` e `String data()` che restituiscono l'autore e la data del documento.

L'applicazione permette la stampa dei diversi tipi di documenti in diversi formati che includono un insieme di possibili *headers* e *footers*. In particolare:

- `DraftHeader`: produce la stringa "`Draft - do not circulate`" all'inizio del documento;
- `DateHeader`: produce la stringa "`Date:`" seguita dalla data, all'inizio del documento;
- `AuthorFooter` produce la stringa "`Author:`" seguita dal nome dell'autore al termine del documento;
- `CopyrightFooter` produce la stringa "`© MP 2005, I Appello`" al termine del documento;

L'applicazione permette di comporre gli headers e footers in modo arbitrario su tutti i tipi di documenti. Progettate l'applicazione descritta applicando il pattern *decorator*. In particolare: (i) definite il diagramma del pattern decorator istanziato al caso in questione; (ii) date uno schema dell'implementazione delle classi decorators, fornendo i costruttori e l'implementazione del metodo `String print()` in ciascuno dei decorators; (iii) dimostrate l'utilizzo dell'applicazione, definendo un metodo `void printFull(ASCII d)` che, dato il documento `d`, crea un documento la cui stampa produce il seguente effetto:

```
Draft - do not circulate
Date: ≪ data di d ≫
≪ contenuto di d ≫
Author: ≪ autore di d ≫
© MP 2005, I Appello
```

```
// SOLUZIONE - CLASSI DOCUMENT
abstract class Document {
    public abstract String print();
    public abstract String autore();
    public abstract String data();
}
class PlainDocument extends Document {
    private String text, author; date;
    public PlainDocument(String text, String date, String author)
      { this.text = text; this.author = author; this.date = date; }
    public String autore() { return author; }
    public String data() { return date; }
    public String print() { return text + "\n" ; }
}
class ASCII extends PlainDocument {
    public ASCII(String text, String date, String author)
        { super(text,date,author); }
    public String print() { return "ASCII: " + super.print(); }
}
class PDF extends PlainDocument {
    public PDF(String text, String date, String author)
        { super(text,date,author); }
    public String print() { return "PDF: " + super.print(); }
}
class PS extends PlainDocument {
    public PS(String text, String date, String author)
        { super(text,date,author); }
    public String print() { return "PS: " + super.print(); }
}
```

```
// CLASSI DECORATOR

abstract class Decorator extends Document {
    private Document document;
    public Decorator(Document d) { document = d; }
    public String print() { return document.print(); }
    public String data() { return document.data(); }
    public String autore() { return document.autore(); }
}

class DraftHeader extends Decorator {
    public DraftHeader(Document d) { super(d); }
    public String print() {
        return "Draft - do not circulate\n" + super.print();
    }
}

class DateHeader extends Decorator {
    public DateHeader(Document d) { super(d); }
    public String print() {
        return "Date: " + data() + "\n" + super.print();
    }
}

class AuthorFooter extends Decorator {
    public AuthorFooter(Document d) { super(d); }
    public String print() {
        return super.print() + "Author: " + autore() + "\n";
    }
}

class CopyrightFooter extends Decorator {
    public CopyrightFooter(Document d) { super(d); }
    public String print() {
        return super.print() + "@ MP 2005, I Appello";
    }
}

class prob4 {
    public static void main(String[] args) {
        ASCII doc = new ASCII("Test parte III","26/1/2005/","Mr. X");
        printFull(doc);
    }
    public static void printFull(ASCII d) {
        Decorator dd = new DraftHeader(
                        new DateHeader(
                          new CopyrightFooter(
                              new AuthorFooter(d))));

        System.out.println(dd.print());
    }
}
```