Metodologie di Programmazione 2003 – 2004

PRIMO APPELLO: 23 Gennaio 2004

Nome: _____ Matricola: _____

Istruzioni

- Scrivete il vostro nome su tutti i fogli.
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- Giustificate le risposte: le risposte senza giustificazione non saranno considerate.
- Tempo a disposizione 2 ore e 30.
- No libri, appunti o altro.

LASCIATE IN BIANCO:

A1	
A2	
A3	
B1	
B2	
B3	
Totale	

Nome: _____ Matricola: _____

Esercizio A1 Considerate la seguente gerarchia di classi:

```
interface M { M m(); }
interface N { void n(); }

class A implements M {
   public M m() { return this; }
}
class B extends A {
   public void k() { }
}
class C extends A implements N {
   public void n() {}
   public void p() {}
}
```

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

1. `N x = new C(); M y = x.m();`

2. `M x = new A(); B y = (B)x.m();`

3. `A x = new C(); ((C)x).p();`

Nome: _____ Matricola: _____

Esercizio A2 Considerate le seguenti classi.

```
class A {
  void test(double x)
  { this.foo(x); }
  void foo(double x)
  { System.out.print("A"); }
}
class B extends A {
  void foo(double x)
  { System.out.print("B-double"); }
  void foo(int x)
  { System.out.print("B-int"); }
}

A a = new A(); B b = new B();
```

1. Cosa stampa `b.test(1)` ? **Motivate la risposta**

2. Cosa stampa `b.test(1.0)` ? **Motivate la risposta**

3. Cosa stampa `b.foo(1)` ? **Motivate la risposta**

Nome: _____ Matricola: _____

Esercizio A3 Dato un tipo `T`, definite il corpo del seguente metodo:

```
int hopeYouKnowHowToIterateOnAList(List l) {
  // scorre la lista l e restituisce il numero di
  // elementi in l che hanno tipo SomeType









}
```

Nome: _____ Matricola: _____

Parte B In questo set di esercizi dovete costruire un sistema di classi per la gestione di un albergo.

L'ALBERGO La classe `Albergo` gestisce un insieme di camere, suddivise in due liste: la lista delle camere libere, e la lista delle camere occupate. Inizialmente tutte le camere sono libere.

La classe `Albergo` ha un costruttore `Albergo(int n)` che costruisce una lista di `n` camere (tutte libere) in cui le camere con numero dispari hanno tipo `Camera` mentre le camere di numero pari hanno tipo `CameraConTV` (vedi seguito). La classe fornisce inoltre due metodi:

- `Camera checkin (Cliente c) throws FullyBookedException`: restituisce la prima camera libera, se ne esiste una, la rimuove dalla lista e la inserisce nella lista delle camere occupate. Se non esiste alcuna camera libera lancia una `FullyBookedException`
- `double checkout (Cliente c, int giorni)`: rimuove la camera occupata dal cliente `c` dalla lista delle camere occupate e la inserisce nella lista delle camere libere. Restitusce il conto che il cliente deve pagare: il conto è calcolato moltiplicando il costo della camera per `giorni`.

Le CAMERE Le camere sono orgnizzate in una gerarchia che modella due tipologie di camere. Ogni `Camera` ha un numero (di tipo `int`), un cliente (il cliente *checkedIn* nella camera) ed un prezzo (di tipo `double`). La classe `Camera` ha due costruttori. Il primo prende un intero che utilizza come numero della camera, mentre inizializza il prezzo ad un valore di default. Il secondo costruttore prende un intero ed un double che utilizza per inizializzare il numero della camera e il prezzo. Al momento della costruzione la camera non ha un cliente associato.

- un metodo `double costo()` che restituisce il prezzo;
- un metodo `void entra(Cliente c):throws NotYourRoomException`: se `c` è il cliente *checkeIn* nella camera non ha effetto, altrimenti lancia `NotYourRoomException`

Alcune camere, di tipo `CameraConTV`, hanno la televisione ed un metodo:

- `void usaTV()`: alla prima invocazione, segnala che la televisione è stata usata.

`CameraConTV` è sottotipo di `Camera`. Nelle camere con TV, il metodo `costo()` restituisce il prezzo della camera sommato ad una costante `FORFAIT`, di tipo double, nel caso in cui cliente abbia usato la televisione.

I CLIENTI Ogni cliente ha un nome, di tipo `String`, ed è associato ad una camera. Il costruttore della classe `Cliente` ha un unico parametro che ne determina il nome: al momento della costruzione, pertanto, un cliente non è associato ad alcuna camera. Inoltre, i clienti hanno i seguenti metodi:

- `boolean checkIn(Albergo a)`: richiede all'albergo una camera. Se l'albergo ha camere libere, occupa la camera in questione e restitusce `true`. Altrimenti restituisce `false`. (nota: questo metodo *non* lancia `FullyBookedException`).
- `void checkOut(Albergo, ngiorni)`: richiede all'albergo il *checkOut* e stampa il valore del conto;
- `void entra() throws NotYourRoomException`: prova ad entrare in camera, se il cliente ha una camera associata;
- `boolean watchTV()`: utilizza la televisione, se la camera la fornisce. In questo caso restituisce `true`, altrimenti `false`.

In tutte le classi utilizzate i qualificatori `private` per tutti i campi, fornendo i metodi *get* e *set* se/quando necessario. Utilizzate i metodi forniti per le liste dalla libreria nel package `java.util`, ed in particolare:

```
boolean isEmpty();
boolean add(Object o);
Object remove(int index);
Object remove(Object o);
```

Nome: _____ Matricola: _____

Esercizio B1 Definite la classe `Albergo` e la classe `FullyBookedException`.

Nome: _____ Matricola: _____

Esercizio B2 Definite le classi `Camere` e `CamereConTV`, e la classe `NotYourRoomException`.

Nome: _____ Matricola: _____

Esercizio B3 Definite la class `Cliente`

Metodologie di Programmazione 2003 – 2004

SECONDO APPELLO: 9 Febbraio 2004

Nome: _____ Matricola: _____

Istruzioni

- Scrivete il vostro nome su tutti i fogli.
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- Giustificate le risposte: le risposte senza giustificazione non saranno considerate.
- Tempo a disposizione 2 ore e 30.
- No libri, appunti o altro.

LASCIATE IN BIANCO:

A1	
A2	
A3	
B1	
B2	
B3	
Totale	

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio A1 Considerate la seguente gerarchia di classi:

```
interface M { A m(); }
interface N { void n(); }
interface K { void k(); }

class A implements M, K {
   public A m() { return this; }
   public void k() {}
}
class B extends A {
   public void k() {}
}
class C implements N {
   public void n() {}
}
```

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

1. `M a = new B(); K b = (K)(a.m());`

2. `M a = new B(); B b = (B)(a.m());`

3. `M a = new B(); B b = ((B)a).m();`

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio A2 Considerate le seguenti classi.

```
class A {
  void show() { p((B)this); }
  void p(A x)
  { System.out.println("A"); }
}
class B extends A {
  void p(A x)
  { System.out.println("B"); }
}
```

1. Quale è il risultato della compilazione e della eventuale esecuzione delle istruzioni: `A a = new A(); a.show();`? **Motivate la risposta**

2. Quale è il risultato della compilazione e della eventuale esecuzione delle istruzioni: `A a = new B(); a.show();` ? **Motivate la risposta**

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio A3 Data la seguente definizione:

```
interface T {  void m();  }
```

definite il corpo del seguente metodo:

```
int iterateAndApply(List alist) {
  // scorre alist invocando il metodo m() su tutti
  // gli elementi che hanno tipo T









}
```

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Parte B In questo set di esercizi dovete costruire un sistema di classi per gestire una partita di scacchi.

La CASELLE e la SCACCHIERA La classe `Casella` definisce le posizioni sulla scacchiera. Ogni casella è caratterizzata da due coordinate (valori interi compresi tra 0 e 7), che ne individuano la posizione sulla scacchiera.

La classe `Scacchiera` gestisce una matrice (8×8) su cui sono disposti i pezzi, fornendo i seguenti metodi:

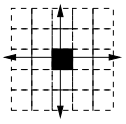
- `Pezzo get(Casella c)`: restituisce il pezzo che si trova alle coordinate corrispondenti alla casella `c`, se un tale pezzo esiste, altrimenti altrimenti `null`.
- `Pezzo mossa(Giocatore g, Casella from, Casella to) throws IllegalMove,NothingToMove:`

  Se la casella `from` non contiene alcun pezzo lancia l'eccezione `NothingToMove`. Se la mossa non è legale lancia una eccezione `IllegalMove`. Altrimenti muove il pezzo da `from` a `to` e restituisce il pezzo contenuto nella casella `to` *prima* della mossa. Il fatto che la mossa sia legale dipende dal tipo del pezzo che è oggetto della mossa e da due ulteriori condizioni: (1) il pezzo deve appartenere al giocatore, (2) la casella `to` non deve essere occupata da un altro pezzo dello stesso giocatore.
- `void demo(MoveGenerator mg)`: crea due giocatori associati alla schacchiera, uno di colore bianco, l'altro di colore nero. Esegue un loop in cui ad ogni passo `mg` viene richiesto di generare una nuova mossa; la mossa generata viene fatta eseguire alternativamente ai giocatori bianco e nero (iniziando dal bianco). Il loop termina non appena uno dei due giocatori mangia il `Re` dell'avversario. Se la mossa generata ad un passo non è legale il giocatore che ha tentato la mossa ripete il suo turno.

I PEZZI Tutti i pezzi hanno un colore, occupano una casella, ed hanno i seguenti metodi:

- `boolean appartiene(Giocatore p)`: `true` se il colore del pezzo e del giocatore coincidono
- `void muovi(Scacchiera s, Casella to) throws IllegalMove`: controlla che sia legale muovere il pezzo dalla casella corrente alla posizione `to`: se si, setta la casella corrente al valore `to`, altrimenti lancia una eccezione. Quando il metodo è invocato, la posizione `to` non contiene un pezzo dello stesso colore del pezzo corrente.

Il controllo che la mossa sia legale dipende dal tipo effettivo del pezzo. Ad esempio: nella classe `Torre`, una mossa è legale se, data la casella corrente, indicata dal quadrato nero, la casella `to` è raggiungibile lungo una delle quattro direzioni indicate nella figura a lato, e tutte le caselle intermedie tra la corrente e la casella `to` (eslusa) sono libere (non contengono alcun pezzo).



I GIOCATORI Ogni giocatore ha un colore, gioca su una scacchiera e mantiene la lista (inizialmente vuota) dei pezzi vinti nel corso di ciascuna partita. Oltre ad opportuni costruttori, la classe `Giocatore` fornisce il metodo:

- `Pezzo muovi(Casella from, Casella to)`: muove il pezzo da `from` a `to`. Se la mossa è legale e la posizione `to` è occupata da un pezzo dell'avversario, quest'ultimo viene *mangiato* e acquisito dal giocatore. Restituisce il pezzo mangiato o null, se nessun pezzo viene mangiato dalla mossa.

In tutte le classi, utilizzate i qualificatori `private` per i tutti campi, fornendo metodi *getter* e *setter* se necessario. Per i colori, utilizzate il tipo `Color` definito nel package `java.awt` ed in particolare le costanti `Color.white` e `Color.black`. Infine, assumete date le seguenti definizioni:

```
class IllegalMove extends Exception {}
class NothingToMove extends Exception {}

interface Mossa {
    public Casella from();
    public Casella to();
}
interface MoveGenerator {
    public Mossa nuovaMossa();
}
```

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio B1 Definite le classi `Casella` e `Scacchiera`

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio B2 Definite le classi `Pezzo` e `Torre`.

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio B3 Definite la classe `Giocatore`

Nome: _____ Matricola: _____

Metodologie di Programmazione 2003 – 2004
TERZO APPELLO: 8 GIUGNO 2004

Nome: _____ Matricola: _____

Istruzioni

- Scrivete il vostro nome su tutti i fogli.
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- Giustificate le risposte: le risposte senza giustificazione non saranno considerate.
- Tempo a disposizione 2 ore e 30.
- No libri, appunti o altro.

LASCIATE IN BIANCO:

A1	
A2	
A3	
B1	
B2	
B3	
Totale	

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio A1 Considerate la seguente gerarchia di classi:

```
interface M { M m(); }
interface N { void n(); }

class A implements M {
   public M m() { return new B(); }
}
class B extends A {
   public M m() { return new A(); }
}
class C extends A implements N {
   public void n() {}
}
```

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

1. `N x = new C(); M y = x.m();`

2. `M x = new A(); B y = (B)x.m();`

3. `M x = new B(); B y = (B)x.m();`

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio A2 Considerate le seguenti classi.

```
class A {
  void test(int d)
  { System.out.print("A"); }
}

class B extends A {
  void test(int i)
  { System.out.print("B-int"); }
  void test(char c)
  { System.out.print("B-char"); }
}
```

1. Cosa stampa `A a = new B(); a.test(1);`? Perchè?

2. Cosa stampa `A a = new B(); a.test('x');` ? Perchè?

3. Cosa stampa `B a = new B(); a.test('x');` ? Perchè?

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio A3 Data la seguente definizione:

```
interface T {  boolean m();  }

class A implements T {
   boolean m() { return false; }
}

class B implements T {
   boolean m() { return true; }
}
```

definite il corpo del seguente metodo:

```
int countAs(List alist) {
  // scorre alist e restituisce il numero di elementi
  // che hanno tipo A









}
```

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Parte B In questo set di esercizi dovete costruire un sistema di classi per simulare l'interprete di una semplice calcolatrice elettronica (CE) per valori reali. Ogni CE è dotata di

- un'area memoria, detta RAM, in cui memorizza i valori;
- un'area di memoria, detta PROG, in cui memorizza la sequenza di istruzioni che formano i programmi;
- quattro registri: A = accumula i risultati; IP = contiene l'indice dell'istruzione corrente; IR = contiene l'istruzione corrente; S = stato della macchina (stop o meno).

L'*instruction set* di una CE comprende le istruzioni nella tabella seguente.

Formato Simbolico	**Significato**
LOAD ind	A ← RAM[ind]
LOADC val	A ← val
STORE ind	RAM[ind] ← A
ADD ind	A ← A + RAM[ind]
MUL ind	A ← A * RAM[ind]
JUMP ind	IP ← ind
INCR	A ← A + 1
DECR	A ← A - 1
ALT	ferma l'esecuzione

Il ciclo di esecuzione è descritto dallo pseudo-codice seguente

```
procedure interpreta (start:integer)
begin
  IP ← start; S ← false
  while (not S)
    IR ← PROG[IP]
    if IR = (LOAD ind) then A ← RAM[ind]
    elsif IR = (STORE ind) then RAM[ind] ← A
    elsif ...
    ...
    elsif IR = ALT then S ← true
    endif
    if IR != (JUMP ind) then IP ← IP+1
end
```

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio B1

Definite una classe `CE` che rappresenti una calcolatrice con le caratteristiche descritte in precedenza. La classe definisce

- opportuni campi per rappresentare le componenti della calcolatrice, e (uno o più) costruttori per inizializzare tali campi;
- un metodo `interpreta(int start, boolean trace)` che simula il comportamento della funzione 'interpreta' definita in precedenza ed inoltre, se `trace == true`, stampa ogni istruzione dopo averla eseguita.

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio B2

Definite una classe astratta `Istruzione` che definisce un campo `String nome` e fornisce

- un costruttore che inizializza il campo nome al valore del parametro
- un metodo `void esegui(CE c)`, dove `CE` è la classe che rappresenta una calcolatrice
- un metodo `String toString()` che restituisce il valore del campo `nome`.

Definite inoltre tre sottoclassi `LOADC`, `INCR`, `ALT` della classe `Istruzione`, per rappresentare le corrispondenti istruzioni di una CE. Ognuna delle sottoclassi definisce almeno:

- un costruttore senza parametri che inizializza il campo `nome` alla stringa che corrisponde al nome della classe (ad esempio: nella classe `LOADC` il campo viene inizializzato alla stringa `"LOADC"`).
- il metodo `esegui(CE c)` che realizza il significato dell'operazione che corrisponde alla classe come indicato nella tabella a pag 4.

Nome: _____ Matricola: _____

Nome: _____ Matricola: _____

Esercizio B3

Definite una classe `Test` che testa la vostra implementazione creando una `CE` e facendogli eseguire la rappresentazione del seguente programma:

LOADI 0 INCR INCR ALT

e stampando la sequenza di istruzioni via via che le esegue.

Nome: _____ Matricola: _____

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());
    }
}
```

Metodologie di Programmazione 2005 – 2006
PRIMO APPELLO: 10 Gennaio 2006

Parte I – Java

Considerate le seguenti definizioni di classe.

```
class A {
  public void print(String s) { System.out.println(s); }
  public void m(int i)     { print("A.m(int)"); }
  public void m(boolean b) { print("A.m(boolean)"); }

  public A k() { return this; }
}

class B extends A {
  public void m(float f)  { print("B.m(float)"); }
  public void m(boolean b) { print("B.m(boolean)"); }
}
class C extends A {
  public void m(int i) { print("C.m(int)"); }
  public void m(double d) { print("C.m(double)");  }
}
```

e assumete le seguenti dichiarazioni di variabile.

```
A va  = new A();          A vab = new B();
B vb = new B();           A vac = new C();
C vc = 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`.

`vab.m(1);`	`A.m(int)`
`vab.m(1.2);`	`compiler error`
`vac.m(1);`	`C.m(int)`
`vb.m(1.2);`	`B.m(float)`
`((B)vab).m(1.2);`	`B.m(float)`
`((B)(vab.k())).m(1)`	`A.m(int)`
`((B)(va.k())).m(1)`	`runtime error`
`((C)(vac.k())).m(false)`	`A.m(boolean)`
`((B)vb.k()).m(true)`	`B.m(boolean)`
`vac.k().m(1.2)`	`compiler error`

Parte II – Datatypes

Un multinsieme è un insieme che può contenere più di una occorrenza di ciascuno dei suoi elementi: ad esempio, {a,a,b,b,b,c} è un multinsieme (ma non è un insieme!) con 2 occorrenze di a, 3 occorrenze di b e una occorrenza di c. L'ordine degli elementi di un multinsieme è irrilevante: quindi, ad esempio, {c,a,b,a} e {b,a,a,c} sono lo stesso multinsieme. All'inverso, {c,a,b,a} è diverso da {a,b,c} perchè non contengono lo stesso numero di a. Un multinsieme è definito formalmente come una coppia (A,m) dove A = {x1,...,xn} è l'*insieme sottostante*, e m : A -> N è la funzione di molteplicità, tale che per ogni x∈A, m(x) è il numero di occorrenze di x. Se x ∉ A, m(x) = 0.

È prassi comune indicare il multinsieme (A,m) come l'insieme di coppie {(x,m(x)) | x ∈ A}. Utilizzando tale notazione abbiamo: {a,a,b,b,b,c} = {(a,2),(b,3),(c,1)} e {c,a,b,a} = {b,a,a,c} = {(a,2),(b,1),(c,1)}. La dimensione di un multinsieme è data dalla somma delle molteplicità dei suoi elementi. Ad esempio: la dimensione di {(a,2),(b,3),(c,1)} è 6.

La seguente definizione di classe fornisce specifica e implementazione per il tipo IntMultiSet. L'invariante di rappresentazione e la funzione di astrazione sono definiti come segue:

$$AF(c) = \{ (c.\texttt{els}[i].\texttt{val}, c.\texttt{els}[i].\texttt{m}) \mid 0 \leq i < c.\texttt{els.size}() \}$$

$$IR(c) = \{ c.\texttt{els} \neq \texttt{null}, \forall i.0 \leq i < .\texttt{els.size}() \, (c.\texttt{els}[i] \neq \texttt{null} : \texttt{Element}, c.\texttt{els}[\texttt{i}].\texttt{m} \geq 0)$$
$$c.\texttt{els} \text{ non contiene duplicati}, \texttt{size} = \Sigma_{i=0}^{c.\texttt{els.size}()-1} c.\texttt{els}[i].\texttt{m} \}$$

```
class IntMultiSet {
    // OVERVIEW: un multinsieme che contiene valori interi

    // RAPPRESENTAZIONE
    private Vector els;
    private int size;

    private static class Element {
        int val, m;
        Element(int v) { val = v; m = 1;}
    }

    // COSTRUTTORE
    public IntMultiSet() {
        // EFFETTO: restituisce un multinsieme di interi vuoto
        els = new Vector(); size = 0;
    }

    // METODI
    private Element find(int x) {
        // EFFETTO: restituisce l'Element che associato ad x.
        // Null se x non appartiene a this
        for (int i = 0; i<els.size(); i++) {
            Element e = (Element)els.get(i); if (e.val == x) return e;
        }
        return null;
    }

    public int isIn(int x) {
        // EFFETTO: restituisce la molteplicita' di x in this
        // (zero se x non appartiene a this)
        Element e = find(x);
        return (e != null)? e.m: 0;
    }
```

```
    public void insert(int x) {
        // EFFETTO: modifica this inserendo una nuova occorrenza di x
        Element e = find(x);
        if (e == null)  els.add(new Element(x));
        else  e.m++;
        size++;
    }

    public void remove(int x) {
        // EFFETTO: modifica this rimuovendo una occorrenza di x
        Element e = find(x);
        if (e == null) return;
        else if (e.m >= 1) { e.m--; size--; }
    }

    public int size() {
        // EFFETTO: restituisce la dimensione di this
        return size;
   }
}
```

Parte III – Subtyping

La classe seguente definisce specifica e implementazione per il sottotipo `MaxDimIntMultiSet` di `IntMultiSet`.

```
class MaxDimIntMultiSet extends IntMultiSet {
    // OVERVIEW: un sottotipo di IntMultiSet con un metodo
    // maxDim che restituisce la dimensione massima raggiunta dal
    // multinsieme nella sequenza di inserzioni e rimozioni
    // effettuate a partire dalla creazione dell'insieme stesso.

    // RAPPRESENTAZIONE
    private int maxsize;

    public MaxDimIntMultiSet() {
        // EFFETTO: restituisce un insieme vuoto
        super(); maxsize = 0;
    }

    public int maxDim() {
        // EFFETTO; restituisce la dimensione massima assunta
        // dal multinsieme nella sua storia
        return maxsize;
    }

    public void insert(int x) {
        // EFFETTO; restituisce la dimensione massima assunta
        // dal multinsieme nella sua storia
        super.insert(x);
        maxsize = Math.max(size(), maxsize);
    }
}
```

Parte IV – Iteratori

```
class Filter implements Iterator {
      // OVERVIEW: un Filter(i,t)  e' un iteratore che, dati
      // i:Iterator e t:Test,  produce, in ordine, tutti gli elementi
      // 'e' prodotti da 'i' per cui t.test(e) restituisce true.
      // IR = { se next != null, next e' l'elemento da restituire
      //        altrimenti non ci sono elementi }
      Iterator i; Test t;
      Object next;

      public Filter(Iterator i, Test t) throws NullPointerException {
          // EFFETTO: costruisce un Filter(i,t). NullPointerException
          // se i e/o t sono nulli
          if (i == null || t == null) throw new NullPointerException();
          this.i = i; this.t = t;
          next = findNext();
       }

      private Object findNext() {
          // EFFETTO: setta next al prossimo elemento da restituire,
      // se esiste, altrimenti null;
          while(i.hasNext()) {
               Object e = i.next();  if (t.test(e)) return e;
       }
         return null;
     }

      public boolean hasNext() {
          // EFFETTO: standard
          return (next != null);
     }

      public Object next() throws NoSuchElementException{
          // EFFETTO: standard
          if (next == null)
              throw new NoSuchElementException();
          Object tmp = next;
          next = findNext();
          return tmp;
      }

      public void remove() { /* not implemented */ }

 }
```

Parte V – Progetto

Considerate la seguente definizione della struttura 'albero binario': un albero binario è vuoto oppure contiene un nodo con una etichetta e due figli, che a loro volta sono alberi binari. La dimensione di un albero binario è il numero di nodi dell'albero (la dimensione di un albero vuoto è zero). Esistono diverse possibili realizzazione degli alberi binari in Java (alcune le avete viste al corso di laboratorio).

In questo esercizio dovrete realizzare una implementazione basata su un gerarchia di tipi che include: un tipo `BT` che rappresenti un generico albero binario, e due sottotipi `EmptyBT` e `NodeBT` che rappresentino rispettivamente le due possibili forme di un albero: vuoto, e con un nodo etichettato da un valore di tipo `Object` e due sottoalberi.

Nella vostra implementazione un albero binario deve fornire almeno due metodi: `int size()` che calcola la dimensione dell'albero, e `boolean equals(...)` che realizza il predicato di uguaglianza tra alberi binari (due alberi binari sono uguali se hanno la stessa struttura e nodi corrispondenti hanno etichette uguali; fornite tutte le versione del metodi `equals` che ritenete opportune)

```
abstract class BT {
    public boolean equals(Object e) {
        if (e instanceof BT) return equals((BT) e);
        else return false;
    }
    abstract public boolean equals(BT e);
    abstract public int size();
}

class EmptyBT extends BT {
    public boolean equals(BT t) { return (t != null && (t instanceof EmptyBT)); }
    public int size() { return 0; }
}

class NodeBT extends BT {
    private Object label;
    private BT left, right;

    public NodeBT(Object label, BT left, BT right) throws NullPointeException {
        if (label == null) throw new NullPointeException();
        this.label = label;
        this.left = (left != null) ? left : new EmptyBT();
        this.right = (right != null) ? right : new EmptyBT();
    }

    public NodeBT(Object label) throws NullPointerException {
        this(label, new EmptyBT(), new EmptyBT());
    }

    public int size() {
        return 1+left.size()+right.size();
    }

    public boolean equals(BT t) {
        if (t == null) return false;
        try {
            NodeBT nt = (NodeBT) t;
            return (label.equals(nt.label) &&
                    left.equals(nt.left) &&
                    right.equals(nt.right));
        }
        catch (ClassCastException e) { return false; }
    }
```

```
}
```

Parte I – Java

Considerate le seguenti definizioni di classe.

```
class T {}
class S extends T {}

class A {
    public void print(String s) { System.out.println(s); }
    public void m(T t)      { print("A.m(T)"); }
    public void m(S s, T t) { print("A.m(S,T)"); }
}

class B extends A {
    public void m(S s)  { print("B.m(S)"); }
    public A id() { return this; }
}

class C extends B {
    public void m(T t) { print("C.m(T)"); }
    public void m(S s1, S s2) { print("C.m(S,S)");  }
    public void m(T t1, T t2) { print("C.m(T,T)");  }
}
```

e assumete le seguenti dichiarazioni di variabile.

```
A va  = new A();          A vab = new B();
B vb = new B();           A vac = new C();
C vc = new C();           T t = new T(); S s = new S();
```

Nella tabella seguente, indicate nella colonna di destra l'output prodotto dal comando riportato nella tabella di sinistra. Se il comando causa errore, indicate nella colonna il tipo di errore, indicando `errore di compilazione` oppure `errore run-time`.

`vab.m(t);`	`A.m(T)`
`vac.m((T)s, s);;`	`compiler error`
`vac.m(s, (T)s);`	`A.m(S,T)`
`vac.m(s, s);`	`A.m(S,T)`
`((C)vac).m((T)s, (T)s);`	`C.m(T,T)`
`((A)vb).m(s);`	`A.m(T)`
`((B)vb).m(s);`	`B.m(S)`
`vab.id().m(s,s);`	`compiler error`
`((B)vc.id()).m(t);`	`C.m(T)`
`((C)vb.id()).m(t);`	`run-time error`

Parte II – Datatypes

Una funzione parziale $A \rightharpoonup B$ è una funzione definita solo su un sottoinsieme di elementi dell'insieme D. Diciamo *dominio* della funzione il sottoinsieme di elementi di D su cui la funzione è definita. Considerate la seguente specifica.

```
  class PartialFun {
      // OVERVIEW: una PartialFun e‘ una funzione parziale a
      // dominio finito dal tipo Object al tipo Object

      public PartialFun()
      // EFFETTO: restituisce una funzione parziale con dominio vuoto
      // (ovvero, indefinita su qualunque argomento)

      public boolean defined(Object o)
      // EFFETTO: restituisce true se this e‘ definita su o,
      // false altrimenti

      public Object apply(Object o) throws UndefinedException
      // EFFETTO: restituisce l'oggetto che this associa ad "o",
      // se "o" e‘ nel dominio di this; altrimenti lancia l'eccezione

      public void update(Object o, Object v) throws IllegalBindingException
      // EFFETTO: modifica this, associando "o" a "v". Se "o" appartiene al
      // dominio di this, modifica l'associazione corrente. Altrimenti
      // aggiunge una nuova associazione. In entrambi i casi, "o" deve
      // deve essere diverso da null, altrimenti lancia l'eccezione

      public int size()
      // EFFETTO: restituisce la dimensione del dominio di this
}
```

Definite l'implementazione del tipo `PartialFun`, scegliendo la rappresentazione che ritenete più opportuna e descrivendo la funzione di astrazione e l'invariante di rappresentazione.

Soluzione:

```
import java.util.*;

class UndefinedException extends Exception {}
class IllegalBindingException extends Exception {}

class PartialFun {
    // OVERVIEW: una PartialFun e‘ una funzione parziale a
    // dominio finito dal tipo Object al tipo Object

    protected static class Pair {
        Object fst, snd;
        Pair(Object f, Object s) { fst = f; snd = s; }
    }
    private List graph;

    public PartialFun() {
        // EFFETTO: restituisce una funzione parziale con dominio vuoto
        // (ovvero, indefinita su qualunque argomento)
         graph = new ArrayList();
    }

    private Pair map(Object o) {
        // EFFETTO: restituisce l'elemento di graph che ha "o" come prima
```

```
        // componente, se "o" e' nel dominio di this;  null altrimenti
        Iterator i = graph.iterator();
        while (i.hasNext()) {
            Pair p = (Pair)i.next();
            if (p.fst.equals(o)) return p;
        }
        return null;
    }

    public boolean defined(Object o) {
        // EFFETTO: restituisce true se this e' definita su o, false altrimenti
        return (map(o) != null);
    }

    public Object apply(Object o) throws UndefinedException {
        // EFFETTO: restituisce l'oggetto che this associa ad "o",
        // se "o" e' nel dominio di this; altrimenti lancia l'eccezione
        Pair p = map(o);
        if (p == null) throw new UndefinedException();
        else return p.snd;
    }

    public void update(Object o, Object v) throws IllegalBindingException {
        // EFFETTO: modifica this, associando "o" a "v". Se "o" appartiene al
        // dominio di this, modifica l'associazione corrente. Altrimenti
        // aggiunge una nuova associazione. In entrambi i casi, "o" deve
        // deve essere diverso da null, altrimenti lancia l'eccezione
        if (o == null || v == null) throw new IllegalBindingException();
        else {
            Pair p = map(o);
            if (p != null) p.snd = v;
            else graph.add(new Pair(o,v));
        }
    }

    public int size() {
        // EFFETTO: restituisce la dimensione del dominio di this
        return graph.size();
    }

    // ITERATORE: soluzione all'esercizio Parte IV

    private static class Bindings implements Iterator {
        private Iterator it;

        Bindings (List l) { it = l.iterator(); }
        public boolean hasNext() { return it.hasNext(); }
        public Object next() { return it.next(); }
        public void remove() {}

    }
    public Iterator bindings() { return new Bindings(graph); }
}
```

Parte III – Subtyping

Considerate la seguente specifica del sottotipo `HomPartialFun` di `PartialFun`.

```
class HomPartialFun extends PartialFun {
 // OVERVIEW: un sottotipo di PartialFun che realizza una
 // funzione parziale omogenea, in cui gli elementi del dominio
 // hanno tutti lo stesso tipo, e la stessa proprieta' vale per il
 // codominio (i tipi del dominio e codominio possono essere
 // diversi).

 public HomPartialFun()
    // EFFETTO: restituisce una funzione parziale omogenea ovunque
    // indefinita

 public void update(Object o, Object v) throws IllegalBindingException
    // EFFETTO; come nel supertipo, ma controlla anche che i tipi
    // di "o" e \"\" siano omogenei con i tipi correnti del
    // dominio e codominio di this. In caso contrario lancia
    l'eccezione.
}
```

Completate la definizione, definendo l'implementazione di `HomPartialFun`. NB: l'implementazione può richiedere nuovi campi e di ridefinire i metodi del supertipo.

Soluzione:

```
class HomPartialFun extends PartialFun {
     // OVERVIEW: un sottotipo di PartialFun che realizza una
     // funzione parziale omogenea, in cui gli elementi del dominio
     // hanno tutti lo stesso tipo, e la stessa proprieta' vale per il
     // codominio (i tipi del dominio e codominio possono essere
     // diversi).

    private Pair first;
    public HomPartialFun() { super(); first = null; }

    public void update(Object o, Object v) throws IllegalBindingException {
        // EFFETTO; come nel supertipo, ma controlla anche che i tipi
        // di "o" e "v" siano omogenei con i tipi correnti del
        // dominio e codominio di this. In caso contrario lancia
        // l'eccezione.
        if (size() == 0) { first = new Pair(o,v); }
        else if (first.fst.getClass() != o.getClass() ||
                 first.snd.getClass() != v.getClass())
             throw new IllegalBindingException();
        super.update(o,v);
    }
}
```

Parte IV – Iteratori

Estendete l'implementazione della classe `PartialFun` implementando il metodo descritto dalla seguente specifica

```
public Iterator bindings()
// EFFETTO: restituisce un Iteratore che enumera,
// tutte le associazioni definite in this,
// restituendole in modo consecutivo ad ogni chiamata.
```

Soluzione: vedi soluzione Parte II.

Parte V – Progetto

Considerate la seguente definizione della struttura 'sequenza': una sequenza è vuota oppure è formata da un elemento etichettato da costante intera e da una sequenza.

Realizzate una implementazione della struttura 'sequenza' mediante una gerarchia di tipi che include esclusivamente: un tipo `Seq` che rappresenta una generica sequenza, e due sottotipi `NullSeq` e `FullSeq` che rappresentano rispettivamente le due possibili forme di una sequenza: vuota e formata a un elemento ed una sottosequenza.

Nella vostra implementazione una sequenza deve fornire opportuni costruttori e supportare i seguenti metodi:

```
public int length();
   // EFFETTO: calcola la lunghezza di this
public boolean equals(Object s);
   // EFFETTO: true solo se l'argomento s e' uguale a this
public boolean equals(Seq s);
   // EFFETTO: come sopra
public Seq append(Seq s)
   // EFFETTO: costruisce la nuova sequenza ottenuta concatenando
   // s alla fine di this.
```

Soluzione:

```
abstract class Seq {
    public boolean equals(Object e) {
        if (e == null || !(e instanceof Seq)) return false;
        else return equals((Seq) e);
    }
    abstract public boolean equals(Seq e);
    abstract public int length();
    abstract Seq append (Seq s);
}

class NullSeq extends Seq {
    public boolean equals(Seq s) { return (s instanceof NullSeq);}
    public int length() { return 0; }
    public Seq append(Seq s) { return s; }
}

class FullSeq extends Seq {
    private int label;
    private Seq tail;

    public FullSeq(int l) { label = l; tail = new NullSeq(); }
    public int length() { return 1+tail.length();   }
    public boolean equals(Seq s) {
        try {
            FullSeq fs = (FullSeq) s;
            return (label == fs.label) && tail.equals(fs.tail);
        }
        catch (ClassCastException e) { return false; }
        catch (NullPointerException e) { return false; }
    }
    public Seq append(Seq s) {
        if (tail instanceof NullSeq) tail = s;
        else  tail.append(s);
        return this;
    }
}
```

Metodologie di Programmazione 2006 – 2007
PRIMO APPELLO: 30 Gennaio 2007

Istruzioni

- Scrivete il vostro nome sul primo foglio.
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- Tempo a disposizione 2 ore e 30.
- No libri, appunti o altro.

LASCIATE IN BIANCO:

JAVA	
DATATYPES	
ITERATORI	
SUBTYPING	
TOTALE	

Parte I – Java

Considerate la seguente gerarchia di classi:

```
interface M { M m(); }
interface N { void n(); }

class A implements M {
   public M m() { return this; }
}
class B extends A {
   public void k() { }
}
class C extends A implements N {
   public void n() {}
   public void p() {}
}
```

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

1. `N x = new C(); M y = x.m();`

2. `M x = new A(); B y = (B)x.m();`

3. `A x = new C(); ((C)x).p();`

Considerate le seguenti classi.

```
class A {
  void test(double x)
  { System.out.print("A"); }
}
class B extends A {
  void test(double x)
  { System.out.print("B-double"); }
  void test(int x)
  { System.out.print("B-int"); }
}
```

Data la dichiarazione `B b = new B()`, quale è il risultato della compilazione e della eventuale esecuzione dei seguenti frammenti? **Motivate le risposte**

4. `((A)b).test(1)`

5. `b.test(1.0)`

6. `b.test(1)`

Parte II – Datatypes

Sia data la seguente specifica della classe `CircList` che realizza il tipo di dato *Lista Circolare*, ovvero una lista in cui il successore dell'ultimo elemento è nuovamente il primo elemento della lista).

```
class CircList {
// OVERVIEW: una CircList e' una lista circolare di Integers.
// Elemento tipico = [x1,...,xn,x1,...,xn, ....]

public CircList()
// POST: costruisce una CircList vuota

public Integer first() throws EmptyException
// POST: se this e' vuota solleva EmptyException, altrimenti
// restituisce il primo elemento di this

public boolean empty()
 // POST: se this e' vuota ritorna true, altrimenti ritorna false.

public void insert (Integer x) throws NullPointerException
 // POST: se x e' null solleva NullPointerexception, altrimenti
 // modifica this aggiungendo x come primo elemento

public Integer delete() throws EmptyException
 // POST se this e' vuota solleva EmptyException, altrimenti rimuove il
 // primo elemento di this, e lo restituisce

public void rightRotate() throws EmptyException;
 // EFFECTS: se this e' vuota solleva EmptyException, altrimenti
 // modifica this ruotando la lista a destra. Ovvero,
 // se this = [x1,...,xn,x1,...,xn,...],
 // this_post = [xn,x1,...,xn-1, xn,x1,...,xn-1,...]

public void leftRotate() throws EmptyException;
 // EFFECTS: se this e' vuota solleva EmptyException, altrimenti
 // modifica this ruotando la lista a sinistra. Ovvero,
 // se this = [x1,...,xn,x1,...,xn,...],
 // this_post = [x2,...,xn,x1, x2,...,xn,x1,...]
}
```

Completate la definizione della classe `CircList`, ovvero:

a) definite la rappresentazione della classe utilizzando una struttura a lista con nodi semplici
b) fornite la funzione di astrazione e l'invariante di rappresentazione
c) definite l'implementazione del costruttore e di tutti i metodi nella specifica

Parte III – Iteratori

Estendete la definizione della classe `CircList` con i due iteratori seguenti:

```
public Iterator elements()
// POST: restituisce un iteratore su this che produce la sequenza
// finita degli elementi del primo ciclo della lista. Ovvero, se
// this = [x1,....,xn,x1,....xn,....] l'iteratore produce la
// sequenza [x1,...,xn]
```

Parte IV – Subtyping

Completate seguente definizione del sottotipo `MinCircList` di `CircList`.

```
class MinCircList extends CircList {
 // OVERVIEW: una CircList con un metodo min() che determina
 // il minimo intero nella lista.

 public Integer min() throws EmptyException
    // POST: restituisce il minimo elemento contenuto nella lista
}
```

La vostra implementazione deve garantire una complessità costante per `min()`, ridefinendo quindi i metodi `insert()` e `delete()` della superclasse.

Metodologie di Programmazione 2007 – 2008
SECONDO APPELLO: 4 LUGLIO 2008

NOME COGNOME _____

MATRICOLA _____

ISTRUZIONI

- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- Giustificate le risposte: le risposte senza giustificazione non saranno considerate.
- Tempo a disposizione 2 ore e 30.
- No libri, appunti o altro.

LASCIATE IN BIANCO:

1	2	3	4	TOT

Java

Considerate la seguente gerarchia di classi:

```
interface M { M m(); }
interface N { void n(); }

class A implements M {
    public M m() { return this; }
}

class B extends A {
    public void k() { }
}

class C extends A implements N {
    public void n() {}
    public void p() {}
}
```

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

MOTIVATE LE RISPOSTE: RISPOSTE NON MOTIVATE NON SARANNO CONSIDERATE.

1. `N x = new C(); M y = (B)x;`

2. `M x = new B(); B y = x.m();`

3. `M x = new B(); ((B)x.m()).k();`

Datatypes

Completate la seguente definizione della classe `BiArrayList` che rappresenta una tabella bidimensionale dinamica, formata da un numero fisso di righe ed un numero variabile di colonne. Definite i campi per realizzare la rappresentazione interna della classe nei modi che ritenete più opportuni, ed implementate i metodi in accordo alla vostra scelta, rispettando la specifica data.

```
/**
 * Una matrice bi-dimensionale di oggetti di tipo T con un numero fissato
 * di righe ed un numero variabile di colonne (tutte le colonne hanno lo
 * stesso numero di posizioni).
 */
class BiArrayList<T>
{
    /**
     * @pre : n > 0
     * @post: crea una BiArrayList vuota, con n righe e 0 colonne
     */
    public BiArrayList(int n)

    /**
     * @pre: TRUE, @result = numero di righe di this
     */
    public int rows()

    /**
     * @pre: TRUE, @result = numero di colonne di this
     */
    public int cols()

    /**
     * @pre: c != null  && c.length == rows()
     * @post: aggiunge c come ultima colonna di this.
     *   Lancia l'eccezione se la precondizione e` violata
     */
    public void insert (T[] c) throws IllegalArgumentException

    /**
     * @pre:  x != null & 0 <= i < rows() && 0 <= j < cols()
     * @post: modifica la posizione (i,j) della matrice settandola
     *        ad x. Eccezione se la precondizione e` violata
     */
    public void set(int i, int j, T x) throws IllegalArgumentException

    /**
     * @pre:  0 <= i < rows() && 0 <= j < cols()
     * @result = il valore memorizzato in posizione (i,j)
     * @post: Eccezione se la precondizione e` violata
     */
    public T get(int i, int j)  throws IllegalArgumentException

    /**
     * @result = un iteratore che restituisce gli elementi di this
     *    enumerandoli in ordine di riga e di colonna
     */
    public Iterator<T> iterator()
}
```

Progetto

Sia data la seguente classe per rappresentare un buffer di n posizioni.

```
class Buffer {
    private char[] contents;

    public Buffer(int n)
    {
      if (n <= 0) throw new ArrayOutOfBoundsException();
      contents = new char[n];
    }

    public void set(int i, char c) throws ArrayIndexOutOfBoundsException
    {
      contents[i] = c;
    }

    public char get(int i) throws ArrayIndexOutOfBoundsException
    {
      return contents[i];
    }
}
```

Definite una sottoclasse `UndoBuffer` di `Buffer` che fornisca un metodo `undo()` che permetta di annullare l'effetto delle operazioni di `set` sulle posizioni del buffer. Il metodo `undo()` si comporta come l'operazione di *annulla* disponibile in un qualunque editor, ovvero: annulla l'effetto dell'ultima operazione di `set()` che non sia stata già annullata; se non ci sono `set()` da annullare, `undo()` non ha alcun effetto.

Esempio:

```
// OPERAZIONE           // STATO DEL BUFFER (_ = carattere nullo)
b = new UndoBuffer(4);  // _::_::_::_
b.set(1,'a');           // _::a::_::_
b.set(1,'b');           // _::b::_::_
b.undo();               // _::a::_::_
b.set(3,'c');           // _::a::_::c
b.undo();               // _::a::_::_
b.undo();               // _::_::_::_
b.undo();               // _::_::_::_
```

Esercitazioni

Vogliamo definire una gerarchia di classi e interfacce per rappresentare e valutare espressioni booleane con una struttura definita dalla seguente sintassi.

$$B ::= \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{IF}\ B\ \texttt{THEN}\ B\ \texttt{ELSE}\ B$$

Sia data la seguente interfaccia:

```
interface BoolExp
{
    /**
     * @result = il risultato della valutazione di this
     * @post: nochange
     */
     boolean double eval()

    /**
     * @result = la stringa che rappresenta this
     * @post: nochange
     */
     String toString()}
}
```

1. Realizzate:
   - una classe `Const` che implementa `BoolExp` e rappresenta una costante di tipo booleano
   - una classe `Cond` che implementa `BoolExp` e rappresenta una expressione condizionale della forma *if-then-else*.

   In entrambe le classi, la specifica dei metodi `eval()` e `toString()` è quella definita nell'interfaccia `BoolExp`.
2. Descrivete la sequenza di istruzioni che costruiscono la rappresentazione della espressione seguente, dove `a` e `b` sono (la rappresentazione di) arbitrarie espressioni booleane.

   ```
   if a then (if b then FALSE else TRUE) else FALSE
   ```

Metodologie di Programmazione 2008 – 2009
I APPELLO: 30 GENNAIO 2009

Nome: _____ Matricola: _____

Istruzioni

- Scrivete il vostro nome sul primo foglio.
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- Gli esercizi 1 e 2 sono obbligatori.
- Gli esercizi 3 e 4 sono ispirati dalle esercitazioni e sono
  - obbligatori per chi non ha sostenuto i quiz ovvero non ha ottenuto la sufficienza,
  - facoltativi per chi volesse migliorare il risultato dei quiz.

  La consegna di questi esercizi annulla comunque il risultato conseguito nei quiz.
- Il voto è il risultato della media pesata tra il punteggio dei primi due esercizi (70%), ed il punteggio degli esercizi 3 e 4 o dei quiz (30%)
- Due turni di consegna: dopo 1,5 ore per i primi due esercizi; dopo 2,5 per tutti gli esercizi. Chi non consegna entro il primo turno perde il punteggio dei quiz e viene valutato sugli esercizi 3 e 4.
- No libri, appunti o altro.

LASCIATE IN BIANCO:

1	2	3	4	TOT

Esercizio 1

Vogliamo definire la rappresentazione di una semplice unità di elaborazione. L'architettura dell'elaboratore include un'area di memoria MEM per i dati, un insieme REG di registri di supporto alle operazioni ed un'area codice PROG dove memorizzare i programmi. L'instruction set dell'elaboratore include le seguenti operazioni elementari.

ISTRUZIONE	**EFFETTO**
LOADC i, v	REG[i] ← v
LOAD i, j	REG[i] ← MEM[j]
STORE i, j	MEM[i] ← REG[j]
STOREC i, v	MEM[i] ← v
ADD i, j	REG[i] ← REG[i] + REG[j]
SUB i, j	REG[i] ← REG[i] - REG[j]
MUL i, j	REG[i] ← REG[i] * REG[j]
DIV i, j	REG[i] ← REG[i] / REG[j]
INC i	REG[i] ← REG[i] + 1
DEC i	REG[i] ← REG[i] - 1

Assumete date le seguenti due definizioni:

```
interface Instruction<T> {
    T unit();
    void exec() throws RuntimeException;
}
class WrongOperandException extends Exception {}
class WrongInstructionException extends Exception {}
class RuntimeException extends Exception {}
```

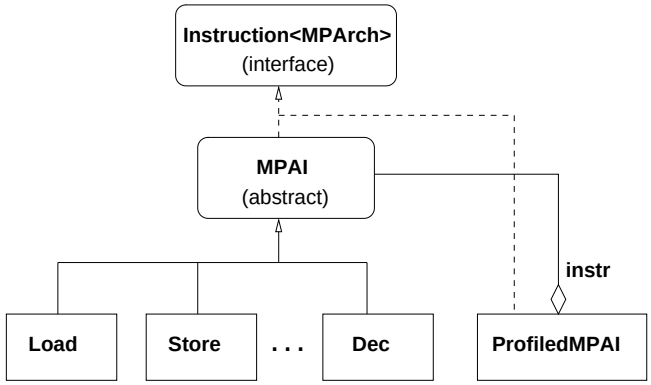
Un oggetto di tipo `Instruction<T>` rappresenta una operazione elementare, il cui effetto si ottiene invocando il metodo `exec()`. Ogni istruzione è associata ad una unità di elaborazione, cui si accede invocando il metodo `unit()`.

Definite una classe `MPArch` che realizzi l'architettura descritta sopra, attenendovi alle seguenti indicazioni:

- *Rappresentazione*
  - l'area di memoria ed i registri contengono valori `double`
  - l'area codice memorizza il programma come un lista di oggetti di tipo `Instruction<MPArch>`.
- *Costruttore*: costruisce una unità di elaborazione dimensionando l'area di memoria e l'insieme di registri come specificato dai parametri:
  ```
  public MPArch(int memsize, int regsize)
  ```
- *Istruzioni elementari*: per ciascuna istruzione la classe definisce un metodo che ne realizza l'effetto. Il metodo lancia `WronglOperandException` nel caso in cui gli argomenti che corrispondono a indirizzi nell'area di memoria e/o a indici di registro siano fuori dei rispettivi range. Implementate solamente i metodi `loadc()` e `store()` a titolo di esempio, attenendovi alle firme riportate qui di seguito:
  ```
  public void loadc(int i, double v) throws WrongOperandException
  public void store(int i, int j) throws WrongOperandException
  ```
- *Loader*: carica la lista di istruzioni nell'area codice, controllando che ciascuna istruzione sia associata all'unità `this`. In caso contrario, lancia l'eccezione.
  ```
  public void load(List<Instruction<MPArch>> prog)
                                        throws WrongInstructionException
  ```
- *Interprete*: esegue in sequenza ciascuna delle istruzioni dell'area codice. Se una delle istruzioni lancia `RuntimeExeption`, il metodo termina istantanamente senza eccezioni stampando un messaggio di errore.
  ```
  public void run()
  ```

Esercizio 2

Il seguente diagramma UML descrive una possibile implementazione delle istruzioni per l'architettura `MPArch` delineata nell'esercizio precedente.



Implementate le classi `MPAI`, `Store` e `ProfiledMPAI`, definendo campi, metodi e costruttori in modo da realizzare le funzionalità descritte qui di seguito.

La classe `MPAI`, astratta, definisce il formato generico di una istruzione, stabilendo l'associazione tra l'istruzione e l'unità di elaborazione per conto della quale sarà eseguita. Implementa i metodi dell'interfaccia `Instruction<MPArch>`, possibilmente definendoli `abstract` e definisce un ulteriore metodo pubblico `String profile()` che restituisce la rappresentazione testuale dell'istruzione.

Ciascuna delle sottoclassi `Load`, `Store`, ..., `Dec` è concreta e rappresenta una istruzione specifica. In ciascuna classe, il metodo `exec()` esegue l'istruzione, invocando il metodo corrispondente sull'unità di elaborazione associata.

La classe `ProfiledMPAI`, anch'essa concreta, realizza una versione "profiled" delle istruzioni: in tali istruzioni, l'invocazione del metodo `exec()` causa la stampa della rappresentazione testuale dell'istruzione stessa, e successivamente la sua esecuzione.

Esercizio 3

Siano date le seguenti definizioni.

```
public interface Measurable
{
   double measure();
}

public class DataStat
{
   public void add(Measurable x)
   {
     if (x == null) return;
     sum = sum + x.measure();
     if (count == 0 || max.measure() < x.measure())
        max = x;
     count++;
   }

   public Measurable max() { return max; }

   public double average() { return (count == 0) ? 0 : sum/count; }

   private double sum;
   private Measurable max;
   private int count;
}
```

Definite una sottoclasse `MoreDataStat` di `DataStat` che definisce i due metodi seguenti:

```
/**
 * @pre true
 * @post @nochange
 * @result = l'ampiezza dell'intervallo delle misure considerate,
 *  ovvero la differenza tra le misure degli elementi con misura
 *  massima e minima.
 */
public double size()

/**
 * @pre d >= 0
 * @post @nochange
 * @result = un iteratore che permette di ottenere in sequenza gli
 *  elementi la cui misura e' minore del valore d
 */
public Iterator<Measurable> lessThan(double d)
```

Esercizio 4

Considerate la seguente gerarchia di tipi:

```
interface M { M m(); }
interface N { }

class A implements M {
   public M m() { return new B(); }
}
class B extends A {
   public M m() { return new A(); }
}
class C extends A implements N {
   public M m() { return this; }
}
```

Indicate il tipo statico ed il tipo dinamico per ciascuna (sotto) espressione nei seguenti frammenti di codice.

Determinate il risultato della compilazione e, nel caso la compilazione non dia errori, dell'esecuzione.

1. `N x = new C(); M y = x.m();`

2. `M x = new A(); B y = (B)x.m();`

3. `M x = new B(); B y = (B)x.m();`

Metodologie di Programmazione 2008 – 2009
II APPELLO: 18 FEBBRAIO 2009

Nome: _____ Matricola: _____

Istruzioni

- Scrivete il vostro nome sul primo foglio.
- Scrivete le soluzioni nello spazio riservato a ciascun esercizio.
- Gli esercizi 1 e 2 sono obbligatori.
- Gli esercizi 3 e 4 sono ispirati dalle esercitazioni e sono
  - obbligatori per chi non ha sostenuto i quiz ovvero non ha ottenuto la sufficienza,
  - facoltativi per chi volesse migliorare il risultato dei quiz.

  La consegna di questi esercizi annulla comunque il risultato conseguito nei quiz.
- Il voto è il risultato della media pesata tra il punteggio dei primi due esercizi (70%), ed il punteggio degli esercizi 3 e 4 o dei quiz (30%)
- Due turni di consegna: dopo 1,5 ore per i primi due esercizi; dopo 2,5 per tutti gli esercizi. Chi non consegna entro il primo turno perde il punteggio dei quiz e viene valutato sugli esercizi 3 e 4.
- No libri, appunti o altro.

LASCIATE IN BIANCO:

1	2	3	4	TOT

Esercizio 1

Sia data la seguente specifica di una classe che realizza la struttura dati *coda*.

```
class Queue<T>
{
  /** costruisce una coda vuota */
  public Queue()

  /** aggiunge elem sulla coda. @pre: elem != null   */
  public void enqueue(T elem)

  /** rimuove la testa della cosa e la restituisce */
  T dequeue()

  /** true se la coda e' vuota */
  public boolean empty()

  /** il numero degli elementi in coda */
  public int size()

}
```

Vogliamo realizzare una classe `UQueue` che estende `Queue` con un metodo `undo()` che permette di annullare l'effetto di ciascuna operazione `enqueue()` o `dequeue()` sulla coda. Il metodo `undo()` si comporta come l'operazione di *annulla* disponibile in un editor, ovvero: annulla l'effetto dell'ultima `enqueue()` o `dequeue()` che non sia stata già annullata; se non ci sono operazioni da annullare, `undo()` non ha alcun effetto.

Esempio:

```
// OPERAZIONE                 // STATO DELLA CODA
q = new UQueue<Integer>();    // []
q.enqueue(1);;                // [1]
q.enqueue(4);                 // [4,1]
q.undo();                     // [1]     -- annulla q.enqueue(4)
q.enqueue(3);                 // [3,1]
q.enqueue(8);                 // [8,3,1]
q.dequeue();                  // [8,3]
q.undo();                     // [8,3,1] -- annulla q.dequeue()
q.undo();                     // [3,1]   -- annulla q.enqueue(8)
```

Realizzate la classe `UQueue` definendo tutti i campi, strutture dati e metodi che ritenete necessari per ottenere le funzionalità richieste.

Esercizio 2

Dovete realizzare una applicazione per l'impaginazione di documenti. Supponiamo che i documenti possano essere solo di due tipi – `Article` e `Draft` – ciascuno con un suo specifico stile di impaginazione, o *layout*.

È data la seguente definizione della classe `Page`, che rappresenta una pagina generica.

```
class Page
{
     protected int np;          // numero della pagina
     private String text;       // il testo della pagina

     /** inizializza i campi ai valori dei parametri corrispondenti */
     public Page(int np, String text){ this.np = np; this.text = text; }

     /**  crea il layout minimo: testo e numero di pagina   */
     String layout() { return (text + "\n" + np); }
}
```

A) Realizzate due sottoclassi `ArticlePage` e `DraftPage` di `Page`. Le due classi sovrascrivono il metodo `layout()`: in `ArticlePage`, il metodo aggiunge un frontespizio (una linea che precede il testo) formato come segue: nelle pagine dispari, contiene i nomi degli autori, nelle pagine pari contiene il titolo del documento. Nella classe `DraftPage`, il metodo aggiunge il frontespizio `Draft - non diffondere`.

B) Completate la definizione della seguente classe che rappresenta un generico documento.

```
abstract class Document<T extends Page>
{
     private List<String> authors;  // autori
     private String title;          // titolo
     private List<T> pages;         // lista delle pagine

     /** crea una pagina con numero n e testo text */
     abstract T createPage(int n, String text);

     /** restituisce il titolo */
     public String title() { return title;  }

     /**
      costruisce un documento con authors come autori, title come titolo
      ed una lista di pagine, opportunamente numerate, che corrisponde
      alla lista ts che contiene il testo delle pagine
     */
     public Document(List<String> authors, String title,  List<String> ts)
     { /* COMPLETARE */ }

     /** restituisce un iteratore sulla lista di autori */
     public Iterator<String> authors() { /* COMPLETARE */ }

     /**
      restituisce la stringa ottenuta concatenando il layout di ciascuna delle pagine
     */
     String print() { /* COMPLETARE */ }
 }
```

C) Realizzate due classi concrete `Article` e `Draft` che estendono `Document` istanziando opportunamente il parametro di tipo della superclasse e implementano il metodo `createPage()` creando, rispettivamente, la `ArticlePage` e la `DraftPage` associate alla stringa che ricevono come parametro.

D) Rispondete alle seguenti domande:

1. Come definireste la classe `Document` senza utilizzare i *generics*?
2. Quale delle due definizioni, con e senza generics, vi pare più appropriata? Perchè?

Esercizio 3

Considerate la seguente gerarchia di classi:

```
interface I { void m(J x); }

interface J { void n(); }

class A implements I {
    public void m(J x) { System.out.println("A.m()"); }
}

class B extends A {
    public void m(C x) { System.out.println("B.m()"); }
}

class C extends A implements J {
    public void m(C x) { System.out.println("C.m()"); }
    public void n() { System.out.println("C.n()"); }
}
```

A) Indicate il tipo statico ed il tipo dinamico per ciascuna (sotto) espressione nei seguenti frammenti di codice.

B) Determinate il risultato della compilazione e, nel caso la compilazione non dia errori, dell'esecuzione.

- `I x = new C(); x.m((J)x);`

- `I x = new A(); x.m((C)x);`

- `B x = new B(); x.m(new C());`

Esercizio 4

Vogliamo definire una gerarchia di classi e interfacce per rappresentare e valutare espressioni booleane con una struttura definita dalla seguente sintassi.

$$B \ ::= \ \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{IF}\ B\ \texttt{THEN}\ B\ \texttt{ELSE}\ B$$

Sia data la seguente interfaccia:

```
interface BoolExp
{
    /**
     * @result = il risultato della valutazione di this
     * @post: nochange
     */
     boolean double eval()

    /**
     * @result = la stringa che rappresenta this
     * @post: nochange
     */
     String toString()}
}
```

1. Realizzate:
   - una classe `Const` che implementa `BoolExp` e rappresenta una costante di tipo booleano
   - una classe `Cond` che implementa `BoolExp` e rappresenta una expressione condizionale della forma *if-then-else*.

   In entrambe le classi, la specifica dei metodi `eval()` e `toString()` è quella definita nell'interfaccia `BoolExp`.
2. Descrivete la sequenza di istruzioni che costruiscono la rappresentazione della espressione seguente, dove `a` e `b` sono (la rappresentazione di) arbitrarie espressioni booleane.

   ```
   if a then (if b then FALSE else TRUE) else FALSE
   ```

Programmazione a Oggetti
Metodologie di Programmazione
1 Giugno 2011

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.

LASCIATE IN BIANCO:

1	2	3	4	5	6	7	8	9	10	TOT

1 Esercizio

Definite l'implementazione del metodo `equals` per la classe `Point` qui di seguito.

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


    // ... altri metodi
}
```

La definizione deve assicurare che, dato un riferimento `p : Point`, il risultato dell'invocazione `p.equals(q)` sia `true` se e solo se `q` è un `Point` con coordinate uguali a quelle di `p`. [3pt]

2 Esercizio

Siano date le seguenti definizioni

```
class SpaceShip
{
  public void CollideWith(Asteroid a) { a.CollideWith(this); }
}

class GiantSpaceShip extends SpaceShip {}

class Asteroid
{
  public void CollideWith(SpaceShip s)
  { system.out.println("Asteroid hit a SpaceShip"); }

  public void CollideWith(GiantSpaceShip s)
  { system.out.println("Asteroid hit a GiantSpaceShip"); }
}

class ExplodingAsteroid extends Asteroid
{
  public void CollideWith(SpaceShip s)
  { system.out.println("ExplodingAsteroid hit a SpaceShip"); }

  public void CollideWith(GiantSpaceShip s)
  { system.out.println("ExplodingAsteroid hit a GiantSpaceShip"); }
}
```

Considerate ora le seguenti dichiarazioni:

```
  Asteroid theAsteroid = new Asteroid();
  ExplodingAsteroid theExplodingAsteroid = new ExplodingAsteroid();

  SpaceShip theSpaceShip = new SpaceShip();
  GiantSpaceShip theGiantSpaceShip = new GiantSpaceShip();

  Asteroid theAsteroidReference = theExplodingAsteroid;
  SpaceShip  theSpaceShipReference = theGiantSpaceShip;
```

Descrivete l'output delle seguenti chiamate: [3pt]

```
  theAsteroid.CollideWith(theSpaceShipReference);

  theAsteroidReference.CollideWith(theSpaceShipReference);

  theSpaceShipReference.CollideWith(theAsteroid);

  theSpaceShipReference.CollideWith(theAsteroidReference);
```

3 Esercizio

Scrivete il codice di una classe `TwoStripes` che realizza una immagine rettangolare composta da due fasce verticali, contigue e di uguale larghezza, associate ciascuna ad un colore. Il costruttore della classe ha la seguente firma:

```
public TwoStripes(int width, int height,
                  PictureColor firstStripe,
                  PictureColor secondStripe);
```

La classe deve implementare l'interfaccia definita qui di seguito:

```
public interface Picture
{
   // restituisce il colore della coordinata (x,y)
   public PictureColor getColor(int x, int y);
   // restituisce la larghezza dell'immagine
   public int getWidth();
   // restituisce l'altezza dell'immagine
   public int getHeight();
}
```

[2pt]

4 Esercizio

Considerate il seguente sistema di classi:

```
public interface StockBroker
{
   public abstract String getAdvice();
}

public class Moodys implements StockBroker
{
   public String getAdvice() { return "buy Microsoft!";}
}

public class MerrillLynch implements StockBroker
{
  public String getAdvice() { return "sell Google!"; }
}

public class Investor
{
  private Stockbroker _sb;
  public Investor(Stockbroker sb) { _sb = sb; }

  public void setStockBroker(Stockbroker sb) {_sb = sb; }

  public String tradeStock()
  {
    return "I am going to "+_sb.getAdvice();
  }
 }
```

Descrivere quale(i) *design pattern(s)* viene (vengono) realizzato (i) dal sistema di classi appena descritto. [2pt]

Quale è l'output del seguente frammento di codice?

```
Investor i = new Investor(new Moodys());
System.out.println(i.tradeStock());
i.setStockbroker(new MerrillLynch());
System.out.println(i.tradeStock());
```

[1pt]

5 Esercizio

Consideriamo la realizzazione del concetto di funzione ottenuta mediante la seguente definizione di interfaccia.

```
public interface Fun
{
  // Applicata all'input x, restituisce un oggetto
  // come  risultato.
  public Object apply(Object x);
}
```

Data `f:Fun` ed `o:Object`, l'espressione `f.apply(o)` denota il risultato ottenuto applicando `f` all'argomento `o`. Ad esempio, consideriamo le due "funzioni" seguenti:

```
public class F implements Fun
{
  public Object apply(Object x)
  { return "F("+x.toString()+")"; }
}
public class G implements Fun
{
  public Object apply(Object x)
  { return "G("+x.toString()+")"; }
}
```

Ora, l'espressione `new F().apply("x")` restituisce la stringa `F(x)`; similmente, l'espressione `new G().apply("x")` restituisce la stringa `G(x)`.

Sia ora data l'interfaccia `Logical`, che rappresenta oggetti con la capacità di applicare una di due possibili funzioni:

```
public interface Logical
{
  // Applica trueF or falseF ad x
  public Object select(Fun trueF, Fun falseF, Object x);
}
```

Definite due classi concrete, `True` e `False` che implementino l'interfaccia `Logical` e realizzino il seguente comportamento. Dato `b:Logical`, l'espressione `b.select(new F(), new G(), x)` restituisce "`F(x)`" se `b` è una istanza di classe `True`, mentre restituisce "`G(x)`" se `b` è una istanza di classe `False`.

NOTA BENE: Il vostro codice per le classi `True` e `False` non deve contenere alcuna costante di tipo stringa. [3pt]

6 Esercizio

Vogliamo definire un sistema di classi ed interfacce che rappresentino le relazioni tra le componenti di un circuito elettronico descritte qui di seguito:

- una *resistenza* è una componente che regola la quantità di corrente da cui è attraversata;
- un *condensatore* è una componente che immagazzina carica;
- un *transistor* è un amplificatore di segnale elettrico;
- un *circuito integrato* è una componente che internamente contiene resistenze, condensatori, transistors e possibilmente altri circuiti integrati che agiscono come un unico elemento;
- una *scheda* contiene una collezione di componenti.

isegnate il diagramma UML delle classi. [3pt]

7 Esercizio

Sia `Pred<T>` l'interfaccia definita come segue:

```
interface Pred<T>
{
     public boolean good(T x);
}
```

Completate il codice delle due classi `EList<T>` (la lista sempre vuota) e `NEList<T>` (una lista sempre non vuota) che implementano l'interfaccia `IList<T>` e realizzano l'interfaccia `IList<T>` definita qui di seguito:

```
public interface IList<T>
{
    // restituisce la lista di tutti gli elementi di
    // this che soddisfano p;  ovvero la lista degli
    // x tali che p.good(x) = true
    IList<T> filter(Pred<T> p);
}

public class EList<T> implements IList<T>
{
    public static final EList singleton = new EList();


}


public class NEList<T> implements IList<T>
{
    private T first;
    private IList<T> rest;

    public NEList(T f, IList<T> r) { first = f; rest = r; }

    public IList<T> filter(Pred<T> p)
    {




    }
}
```

[3pt]

8 Esercizio

I neuroni presenti nel cervello realizzano un sistema bio-elettrico molto sofisticato che trasmette l'effetto di uno stimolo mediante un impulso elettrico. La propagazione dell'impulso corrisponde ad una modifica dello "stato elettrico" del neurone, che si svolge in quattro fasi, ognuna delle quali ha una durata di circa un 1 millisecondo.

- *resting*: il neurone è a riposo, voltaggio = -70.0mV); se stimolato passa nello stato *rising*
- *rising*: il neurone passa ad un voltaggio = 0.0 mV; in questa fase eventuali stimoli sono ignorati; dopo un millisecondo passa nello stato *falling*; in
- *falling*: fase calante dello stimolo, corrispondente ad un voltaggio = -25.0mV; anche in questa fase il neurono non reagisce a stimoli; dopo un millisecondo passa nella fase *undershoot*
- *undershoot*: fase di riequilibrio, ad un voltaggio = -80.0mV; se stimolato passa allo stato *resting*, a cui passa comunque dopo un millisecondo;

Possiamo rappresentare i neuroni come oggetti della classe `Neuron` descritta qui di seguito in termini di tre metodi:

- `stimulate()`: stimola il neurone per attivare l'impulso elettrico
- `getVoltage()`: misura il voltaggio della membrana
- `wait1ms`: simula il passaggio di tempo ed il conseguente passaggio delle diverse fasi nella trasmissione dello stimolo;

```
public class Neuron
{
  private State _state = RestingState.Singleton;

  void setState(State state) { _state = state;  }
  public double getVoltage() { return _state.getVoltage(this); }
  public void wait1ms()   { _state.wait1ms(this);   }
  public void stimulate() { _state.stimulate(this); }
}
```

Definite la classe astratta `State` e quattro classi concrete `RestingState`, `RisingState`, `FallingState` e `UndershootState` che realizzano il comportamento descritto in precedenza; [3pt]

9 Esercizio

Data l'interfaccia albero binario:

```
public interface AlberoBinario<T> {

    AlberoBinario<T> getLeft();
    AlberoBinario<T> getRigth();
    T getInfo();
    boolean isEmpty();

    List<T> preordine();
    Iterator<T> iterator();
}
```

Definiamo la *visita in preordine* la visita dell'albero in cui viene visitata prima l'informazione della radice, poi ricorsivamente visitato in preordine il sottoalbero sinistro e successivamente il sottoalbero destro. Implementare i metodi `getLeft, getRight, getInfo` e `isEmpty` definendo la classe `AlberoBinarioImpl` con le variabili d'istanza necessarie. [2pt]

Implementare il metodo ricorsivo `preordine` che ritorna una lista con le informazioni dell'albero in preordine. [3pt]

10 Esercizio

Con riferimento all'interfaccia `AlberoBinario` dell'Esercizio 9, implementate il metodo `iterator` in modo restituisca un iteratore che visita l'albero in preordine **non usando** una lista di supporto in preordine ma usando uno stack. [5pt]

Disegnare il diagramma delle classi nel caso si adotti il Design Pattern NullObject. [2pt]

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
    {




    }
}
```

Programmazione a Oggetti
Metodologie di Programmazione
14 Maggio 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 scelga 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

Correggere l'errore nel seguente codice Java: [1,5pt]

```
class Base
{
      private int b;
      public Base(int b) { this.b = b; }
}

class Derived extends Base
{


}
```

Completate la definizione della classe `Derived` e del metodo `main()` nella classe `Test` in modo che l'esecuzione restituisca l'output `"AB"`. **NB: non potete utilizzare comandi di stampa.** [1,5pt]

```
abstract class Base
{
      public Base() { System.out.print("A"); }
}

class Derived extends Base
{
      int a = ... ;

      int m()
      {
        System.out.print("B");
        return 0;
      }
}

class Test
{
      public static void main(String[] args)
      {
        ...

      }
}
```

2 Esercizio

Implementare il metodo `equals` per la classe `Nil` definita qui di seguito. [2pt]

```
class Nil
{




}
```

Implementare il metodo `equals` per la realizzazione della classe `Nil` definita qui di seguito. [2pt]

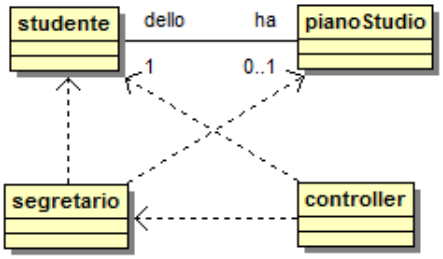
```
class Nil
{
    private static Nil singleton = new Nil();
    public Nil getSingleton(){ return singleton; }




}
```

3 Esercizio

Illustrate con un esempio, il concetto di polimorfismo in Java. (5 righe max) [3pt]

4 Esercizio

Un'applicazione desktop permette a un utente segretario la stampa di un piano di studi relativo a uno studente. L'implementazione di tale funzionalità è descritta qui di seguito mediante un diagramma UML ed il corrispondente schema di classi Java.




```
class Studente
{
   public PianoDiStudio getPiano(){...}
   ...
}

public class PianoDiStudio
{
   public String getDescription(){...}
   ...
}

public class Segretario
{
  public void printPianoStudio(Studente s)
  {
    PianoDiStudio pds = s.getPiano();
    System.out.println("Piano di Studio per "+s.getName());
    System.out.println( pds.getDescription() );
    ...
  }
}

public class Controller
{
   public void onEvent(Event e)
   {
      ...
      utente.printPianoStudio(s);
      ...
   }
}
```

Modificare il codice in modo da implementare il pattern Information Expert. [2pt]
Disegnare il nuovo diagramma delle classi. [2pt]

5 Esercizio

Considerate la seguente classe:

```
public class Giocatore
{
     public String getNome()  { return nome;    }

     private String nome;
}
```

Completate il codice della classe `TabelloneMonopoli` descritta qui di seguito, utilizzando la struttura dati che ritenete più opportuna. [3pt]

```
public class TabelloneMonopoli
{
 // inizialmente le caselle sono tutte senza proprietario
 // le caselle sono indicate da un intero da 0 a 39.

     // il giocatore g diviene proprietario della casella
     public void compra(Giocatore g, int casella)
     {
      ......................................
      ......................................

     }
     // il giocatore g vende la casella se la possiede
     // altrimenti non fa niente e ritorna false
     public boolean vendi(Giocatore g, int casella)
     {
       ......................................
       ......................................

     }
     // ritorna il giocatore proprietario della casella
     // null se la casella non ha proprietario
     public Giocatore getProprietario (int casella)
     {
       ......................................
       ......................................

     }

     private ............... tabellone =  ..................;
}
```

6 Esercizio

Vogliamo definire un sistema di classi e interfacce che rappresentino le relazioni tra le componenti di un circuito elettronico descritte qui di seguito:

- una *resistenza* è una componente che regola la quantità di corrente da cui è attraversata;
- un *condensatore* è una componente che immagazzina carica;
- un *transistor* è un amplificatore di segnale elettrico;
- una *scheda* contiene una collezione di componenti.

Dato il seguente diagramma UML che rappresenta una possibile soluzione del problema, scrivete il corrispondente codice Java che rappresenti le relazioni del diagramma. [3pt]

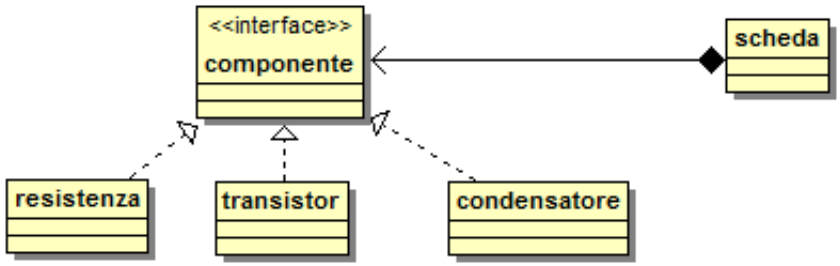


Figure 1: Diagramma delle classi

7 Esercizio

Si implementi il seguente diagramma delle classi in modo che:

- l'invocazione del metodo `impostaSuoneria` imposti come suoneria la suoneria passata come parametro.
- l'invocazione del metodo `onKeyPressed` comporti l'alternanza tra `radio` e `allarme` come impostazione della suoneria.
- l'invocazione del metodo `onTimer` comporti l'attivazione della suoneria (invocazione del metodo `suona` della suoneria impostata).
- la `radio` quando attivata stampi nello standard output `"musica"`.
- l'`allarme` quando attivato stampi nello standard output `"bee bee"`.
- il metodo `alterna` di `suoneria` per ogni sua implementazione cambi l'impostazione della sveglia passata come parametro, impostando l'altra suoneria (se chiamato su `allarme` imposta `radio`, viceversa se chiamato su `radio` imposta `allarme`).
- alla creazione la sveglia sia impostata su `allarme`.

[3pt]

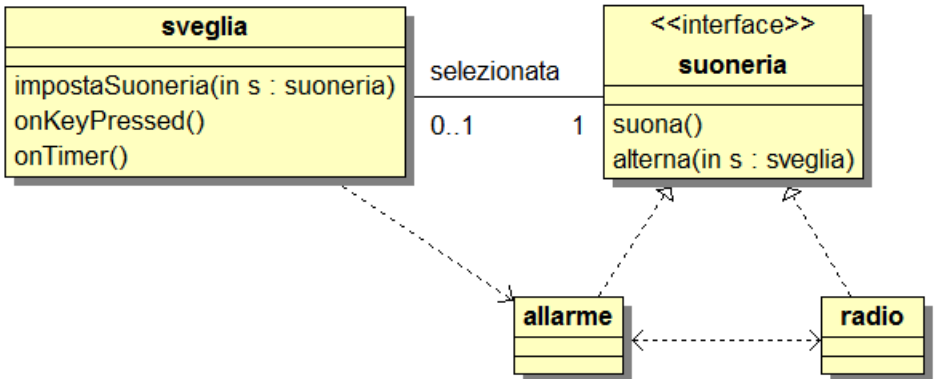


Figure 2: Diagramma delle classi

8 Esercizio

Data l'interfaccia lista:

```
interface Lista<T> {

    boolean isEmpty();
    T getFirst();
    Lista<T> getTail();

}
```

Implementare l'interfaccia Lista usando il pattern NullObject.

[3pt]

9 Esercizio

Con riferimento all'interfaccia `Lista` dell'Esercizio 9, implementate il metodo `public Iterator<T> iterator()` in modo che restituisca un iteratore che visita tutti gli elementi della lista nell'ordine naturale. [3pt]

10 Esercizio

Data la definizione alternativa di Lista:

```
interface ListaM<T>{

    T getFirst();
    void addFirst(T t);
    T removeFirst();

}
```

Mostrare perché non è possibile applicare il pattern NullObject. [3pt]

Programmazione a Oggetti
Metodologie di Programmazione
8 Giugno 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 scelga 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

Descrivere a cosa serve e come funziona il design patter Observer (o Listener). Completate l'esercizio con il codice di una classe di esempio che implementa il ruolo dell'Observer. [3pt]

2 Esercizio

1. In quale situazione e' necessario definire un costruttore *private*? In quale situazione può essere utile definire un costruttore *protected*? Per entrambe le situazioni date un esempio. [2pt]

2. Descrivete le differenze tra metodi `static` e non `static` in Java. Date una versione corretta della seguente definizione di classe:

```
class StaticTest
{
      private int a;
      static private int b;

      public int m1(int c) { return c * (a + b); }
      public static int m2(int d) { return d * (a - b); }
}
```

[2pt]

3 Esercizio

Sia data la seguente gerarchia:

```
    interface M { M m(); }
    interface K { void k(); }

    class A implements M, K {
       public M m() { System.out.print(" A "); return this; }
       public void k() {}
    }
    class B extends A {
       public M m() { System.out.print(" B "); return super.m(); }
}
```

Indicate il tipo statico ed il tipo dinamico per ciascuna (sotto) espressione nei seguenti frammenti di codice. Determinate inoltre il risultato della compilazione e, nel caso la compilazione non dia errori, dell'esecuzione. [3pt]

A. `M a = new B(); B b = ((B)a).m();`

B. `M a = new B(); K b = (K)(a.m());`

C. `M a = new A(); B b = (B)(a.m());`

4 Esercizio

Date le seguenti classi:

```
public class A {
}

public class B {
   public boolean equals(Object o){
      if (o==null || !( o instanceof  B))
        return false;
      return true;
   }
   public int hashCode () {return 0;}
}
```

Scrivere a fianco di ogni istruzione di stampa cosa viene stampato. Nel caso un' istruzione lanci un'eccezione, si assuma che l'esecuzione prosegua alla riga successiva. [3pt]

```
   Set<Object> set = new HashSet<Object>();
   A a = new A();
   set.add(a);
   System.out.println(set.size());   //.........
   set.add(new A());
   System.out.println(set.size());   //.........
   A a2 = a;
   set.add(a2);
   System.out.println(set.size());   //.........
   B b = new B();
   set.add(b);
   System.out.println(set.size());   //.........
   set.add(new B());
   System.out.println(set.size());   //.........
   B b2 = b;
   set.add(b2);
   System.out.println(set.size());   //.........
```

5 Esercizio

Considerate le seguenti classi che descrivono le monete della valuta americana. In ciascuna classe, il metodo `value()` restituisce il valore della moneta in centesimi (di dollaro).

```
class Penny   { public int value(){ return 1;  } }

class Nickel  { public int value(){ return 5;  } }

class Dime    { public int value(){ return 10; } }

class Quarter { public int value(){ return 25; } }
```

Definite un nuovo tipo `Coin` che vi permetta di completate il codice della classe `PiggyBank` (salvadanaio) descritto qui di seguito, utilizzando la classe che ritenete più opportuna del framework collection di Java. [4pt]

```
public class PiggyBank
{
     // Inizialmente il salvadanaio e' vuoto
     public PiggyBank()
     {
      ....................................
     }

     // Aggiunge al salvadanaio il coin c
     public void save(Coin c)
     {
      ....................................
      ....................................

     }

     // Restituisce tutto il contenuto del salvadanaio
     public Iterator<Coin> break()
     {
       ....................................
       ....................................

     }

     // Restituisce il valore totale del contenuto espresso in centesimi
     public int total()
     {
       ....................................
       ....................................
       ....................................
     }

     private ............... contents =  ................;
}
```

6 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]

7 Esercizio

Considerate la seguente rappresentazione degli insiemi.

```
class Set<T>
{
     private ArrayList<T> contents;

     public Set() { this.contents = new  ArrayList<T>(); }

     public Iterator<T> getContents(){ return contents.iterator(); }

     public void add(T val)
     { if (!contents.contains(val)) contents.add(val); }

     public void remove(T val){ contents.remove(val); }
}
```

Completate la definizione della classe `MaxSet<T>` seguendo le indicazioni date nei commenti. [4pt]

```
class MaxSet<T extends Comparable<T>> extends Set<T>
{
     private T 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(T val)
     {
       ...................
       ...................
     }
     // rimuove una occorrenza di val da this, aggiornando il
     // max in tutti i casi in cui sia necessario
     public void remove(T val)
     {
       ...................
       ...................
       ...................
       ...................
       ...................
     }
}
```

Ricordiamo qui di seguito la specifica dell'interfaccia `Comparable<T>`.

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

8 Esercizio

Siano date le seguenti classi per la gestione di un file system.

```
class File
{
   private String contents;
   private String name, owner;
   private int size;

   public File(String name, String owner)
   {
     this.name = name; this.owner = owner;
   }

   public int size()     { contents.length * 2; }
   public String name()  { return name; }
   public String owner() { return owner; }
}

class Directory
{
   private List<File> contents;
   private String name, owner;

   public Directory(String name, String owner)
   {
     this.name = name; this.owner = owner;
     contents = new LinkedList();
   }
   public String name()  { return name; }
   public String owner() { return owner; }
   public size()
   {
     // restituisce le somme delle dimensioni di ciascun
     // file contenuto nella directory
   }
   public void add(File f) { contents.add(f); }
}
```

Dovete modificare la struttura delle classi in modo che una directory possa contenere non solo file ma anche directory, che a loro volta conterranno file e directory. In particolare:

A. Definite una nuova classe astratta `Resource` per rappresentare file e directory. [1pt]

B. Ridefinite le classi `File` e `Directory` come sottoclassi di `Resource`. La nuova classe `File` mantiene le funzionalità della classe originale. La nuova classe `Directory` dovrà invece gestire una collezione di oggetti `Resource` invece che `File`. Inoltre, il metodo `size()` dovrà restituire la dimensione totale dei file (solo dei file) contenuti nella directory corrente e, ricorsivamente, nelle eventuali directory contenute nella directory corrente. [4pt]

9 Esercizio

Siano date le seguenti definizione di classe ed interfaccia.

```
interface Measurable
{ /** oggetti misurabili */
  public double measure() ;
}

class Queue<T>
{
  /** costruisce una coda vuota */
  public Queue()

  /** aggiunge elem sulla coda. @pre: elem != null   */
  public void enqueue(T elem)

  /** rimuove la testa della cosa e la restituisce */
  T dequeue()

  /** true se la coda e' vuota */
  public boolean empty()

  /** il numero degli elementi in coda */
  public int size()

}
```

Vogliamo definire una nuova classe `BoundedMeasureQueue`, che si comporta come una coda, ma: (i) può contenere solo elementi di tipo `Measure`, e (ii) il totale delle misure degli elementi al suo interno non può superare un valore `double` dato. Nel caso in cui l'aggiunta di un elemento in doda faccia superare la soglia, una chiamata ad `enqueue()` non modifica la coda.

Definite la nuova classe `BoundedMeasureQueue` come sottoclasse della classe `Queue` data.

10 Esercizio

Sia data la sequente definizione.

```
public class BlinkPanel extends JPanel
{
    public BlinkPanel()
    {
       btn = new JButton("Click me");
       label = new JLabel("");
       add(btn); add(label);
       btn.addActionListener(new Blinker());
    }

    public JLabel getLabel() { return label; }
    private JButton btn;
    private JLabel label;

    private class Blinker implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            if (label.getBackground() == Color.WHITE)
                label.setBackground(Color.BLACK);
            else
                label.setBackground(Color.WHITE);
        }
        private Color col = Color.WHITE;
    }
}
```

Modificate l'implementazione della classe `BlinkPanel` come richiesto nei due punti qui di seguito. NB: le modifiche richieste sull'implementazione sono distinte, non cumulative e non devono modificare le funzionalità pubbliche della classe `BlinkPanel`.

A. Eliminate la classe interna `Blinker` e ridefinitela come classe esterna.

B. Eliminante la classe interna `Blinker` rendendo la classe `BlinkPanel` un `ActionListener`.