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; }
    }
```

```
}
```