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;
    }
}
```