PROGRAMMAZIONE II (A,B) - a.a. 2014-15
Primo Appello — 10 giugno 2015

Esercizio 1. Si consideri il seguente programma scritto in una sintassi Java-like.

```
public class ClassInitializationTest {
   public static void main(String[] args) throws InterruptedException {
       NotUsed o = null;
       Child t = new Child();
   }
}

public class Parent { ... }

public class NotUsed { ... }

public class Child extends Parent { ... }
```

1. Si descriva la politica di inizializzazione delle classi, motivando la risposta.

```
Assumendo che Parent non invochi altre classi,
l'ordine di inizializzazione delle classi e'
Object - Parent - Child - ClassInitializationTest
```

Esercizio 2. Si consideri il seguente programma scritto in OCaml.

```
let myOp x y = (x + y) * x in
  let rec f g n =  if n = 1 then g 0
                   else myOp (g 0) (f (fun x -> n) (n-1)) in
    f (fun x -> 10) 2;;
```

1. Si determini il tipo degli identificatori che compaiono nel programma scelto.

```
val myOp : int -> int -> int = <fun>
val f : (int -> int) -> int -> int = <fun>
```

2. Si descriva lo stato dello stack dei record di attivazione nella simulazione della valutazione del programma. In particolare, si indichino per ogni record di attivazione le informazioni relative al puntatore di catena statica, al puntatore di catena dinamica e alla struttura dell'ambiente locale.

3. Si determini il valore calcolato dal programma. `- : int = 120`

4. Si discuta la regola operazionale dell'interprete relativa all'invocazione di una funzione considerando come caso specifico l'invocazione `f (fun x -> 10) 2`.

Esercizio 3. Si consideri l'interfacaccia `Comparable` delle librerie standard Java.

```
public interface Comparable<T> {

//EFFECTS: if this minore di obj then result < 0
           else if this e' uguale a obj then result = 0
                else result = 1
int compareTo(T obj)

// Il metodo definisce un ordinamento totale sugli elementi del tipo generico T.
}
```

Si consideri il seguente frammento della classe `PriorityQueue`, anch'essa parte delle librerie standard Java.

```
public class PriorityQueue<E> {
// OVERVIEW: collezione mutabile di oggetti generici di tipo E nella quale
//           gli elementi sono disposti in ordine, dal piu' piccolo
//            (ovvero, l'elemento in posizione 0) al piu' grande.
//           Per l'ordinamento il tipo E deve implementare l'interfaccia Comparable.

   // EFFECTS: inizializza this alla coda vuota
   public PriorityQueue()

   // EFFECTS: restituisce il numero degli elementi nella coda
   public int size()

   // EFFECTS: restituisce true se x appartiene a this
   public boolean contains(E x)

   // EFFECTS: restituisce l'elemento di tipo E in posizione i
   public E get(int i)

   // EFFECTS: restituisce l'elemento di tipo E piu' grande nell'ordinamento
   //          o null nel caso this sia vuoto
   public E peek()

   // EFFECTS: restituisce l'elemento di tipo E piu' grande nell'ordinamento
   //          o null nel caso this sia vuoto
   //          tale elemento e' rimosso da this
   public E poll()

   // EFFECTS: x e' aggiunto a this
   // THROWS: NullPointerException if x == null
   public void add(E x)
}
```

1. Senza fare assunzioni sulla struttura dati sottostante, si scriva l'implementazione del metodo `peek()`.

```
public E peek() {
   if (this.size() == 0) return null;

   E max = this.get(0);
   for(int i = 0; i < this.size(); i++) {
      if(max.compareTo(this.get(i)) <= 0)
         max = this.get(i);

   return max;
}
// L'uso di "return this.get(this.size()-1)" e' stato
// accettato, ma sottintendeva una scelta implementativa
```

2. Supponendo di utilizzare come struttura di implementazione

   ```
   private Vector<E> coda;
   private E testa;
   ```

   si definiscano la funzione di astrazione e l'invariante di rappresentazione.

```
FA = <coda[0], coda[1], ..., coda[coda.size()-1]>

IR = (coda != null) && (coda.size() > 0 ==> testa == coda[coda.size()-1])
     && (forall i. 0 <= i < coda.size() ==> coda[i] != null)
     && (forall i. 0 < i < coda.size() ==> coda[i-1].compareTo(coda[i]) <= 0)
```

3. Si dia l'implementazione del metodo `add()` e si dimostri che preserva l'invariante di rappresentazione.

```
public void add(E x) {
   if (x == null) throw new NullPointerException();

   boolean found = false;
   for(int i = 0; i < this.size() && !found; i++) {
      if(x.compareTo(coda[i]) <= 0)  {
         coda.add(i, x); found = true;
      }
   }

   if (!found) { coda.add(x); testa = x };
}

// post = (x != null ==> x occorre in FA)
// L'uso della variabile booleana found risulta di aiuto
// per una prova formale che il metodo preserva IR
```

4. Si discuta se è possibile definire la classe `DistinctElements-PriorityQueue`, nella quale gli elementi della coda sono tutti differenti, come estensione di `PriorityQueue` in modo tale che rispetti il principio di sostituzione.

```
// IR' = IR
         && (forall i. 0 < i < coda.size() ==> coda[i-1].compareTo(coda[i]) < 0)

public void add(E x) {
   if (!this.contains(x)) super.add(x);
}

// pre' = (forall i. 0 <= i < coda.size() ==> x.compareTo(coda[i]) != 0)
// post' = (x != null ==> x occorre in FA)
// La regola delle proprieta' e' rispettata, dato che IR' ==> IR e i  metodi
// della sottoclasse preservano IR'
// Non e' rispettata la regola dei metodi, dato che il pre del metodo add della
// superclasse non implica il pre del metodo add sovrascritto
```