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
   ```