Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction à l'informatique II (ITI1521)
EXAMEN FINAL

Instructeurs: Guy-Vincent Jourdan et Marcel Turcotte

Avril 2018, durée: 3 heures

Identification

Nom de famille : _____ Prénom(s) : _____

Étudiant : _____ # Siège : _____ Signature : _____

Instructions

1. Examen à livres fermés.
2. L'utilisation de calculatrices, d'appareils électroniques ou tout autre dispositif de communication est interdit.
   (a) Tout appareil doit être éteint et rangé.
   (b) Toute personne qui refuse de se conformer à ces règles pourrait être accusée de fraude scolaire.
3. Répondez sur ce questionnaire.
   (a) Utilisez le verso des pages si nécessaire.
   (b) Aucune page supplémentaire n'est permise.
4. Écrivez vos commentaires et hypothèses afin d'obtenir des points partiels
5. Écrivez lisiblement, puisque votre note en dépend.
6. Ne retirez pas l'agrafe du livret d'examen.
7. Attendez l'annonce de début de l'examen.

Barème

Question	Maximum	Résultat
1	5	
2	10	
3	5	
4	15	
5	10	
6	4	
7	6	
8	10	
Total	**65**	

Question 1 : (5 points)

Analysez attentivement le programme Java ci-dessous et répondez à la question.

```
public class Boing {

    private void boom() {
        if (true) {
            throw new RuntimeException("Aouch!");
        }
        System.out.println("Saved!");
    }

    public void bam(){
        try {
             boom();
        } catch (RuntimeException e) {
            System.out.println(e);
        }
        System.out.println("Finished!");
    }

    public static void main(String[] args) {
        Boing boing = new Boing();
        boing.bam();
    }
}
```

Choisissez l'option qui correspond le mieux à l'exécution du programme Java ci-dessus.

A.

```
java.lang.RuntimeException: Aouch!
Finished!
Saved!
```

B.

```
java.lang.RuntimeException: Aouch!
Saved!
Finished!
```

C.

```
java.lang.RuntimeException: Aouch!
Finished!
```

D.

```
Saved!
Finished!
```

E. Ce programme produit une erreur de compilation.

F. Autre, expliquez votre réponse dans la boîte ci-dessous.

Question 2 : (10 points)

Analysez attentivement le code Java ci-dessous et répondez aux questions.

```
public class Alpha {

    public String toString(){
        return "Alpha";
    }

}
```

```
public class Beta extends Alpha implements Gamma {

    public void  gamma() {
        System.out.println("Gamma");
    }

    public String toString() {
        return "Beta";
    }

}
```

```
public interface Gamma {
    void gamma();
}
```

Pour chaque bloc de code, indiquez le résultat attendu.

A.

```
Alpha a1 = new Alpha();
System.out.println(a1);
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

B.

```
Alpha a2 = new Beta();
System.out.println(a2);
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

C.

```
Beta b1 = new Alpha();
System.out.println(b1);
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

D.

```
Beta b2 = new Beta();
System.out.println(b2);
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

E.

```
Alpha a3 = new Alpha();
a3.gamma();
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

F.

```
        Alpha a4 = new Beta();
        a4.gamma();
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

G.

```
Beta b3 = new Beta();
b3.gamma();
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

H.

```
Beta b4 = new Gamma();
b4.gamma();
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

I.

```
Gamma g1 = new Alpha();
g1.gamma();
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

(e) Autre

J.

```
Gamma g2 = new Beta();
g2.gamma();
```

(a) Affiche **Alpha**

(b) Affiche **Beta**

(c) Affiche **Gamma**

(d) Erreur de compilation

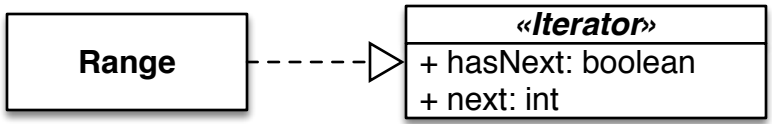
(e) Autre

Question 3 : (5 points)

Vous trouverez ci-dessous une expression en format **postfixe**. À l'aide de l'algorithme présenté en classe, vous devez transformer cette expression et donner l'expression **infixe** correspondante.

```
4   3   *   7   6   -   *   2   4   +   /
```

Question 4 : (15 points)



Un objet de la classe **Range** peut être vu («*can be seen*») comme un **Iterator**. Pour cette question, la méthode **next** de l'itérateur retourne une valeur de type **int**.

```
Iterator i;
i = new Range(5);

while (i.hasNext()) {
    System.out.println(i.next());
}
```

Sur la page suivante, donnez l'implémentation complète de **Range** et **Iterator** de sorte que l'exécution du programme Java ci-dessus produise la sortie suivante sur la console.

```
0
1
2
3
4
```

Suivez toutes les directives suivantes :

- La classe **Range** n'a qu'un constructeur. Ce dernier n'a qu'un paramètre et il est de type **int**.
- Le premier appel valide à la méthode **next** retourne la valeur **0**. Les appels subséquents retournent la valeur suivante dans l'itération. Le nombre maximal d'appels à la méthode **next** est spécifié par le paramètre du constructeur. Si le nombre d'appels excède cette valeur, la méthode lance l'exception **IllegalStateException**.
- La méthode **hasNext** retourne **true** si et seulement si le prochain appel à la méthode **next** est possible (ne lance pas une exception).

Question 5 : (10 points)

Un **arbre binaire de recherche** est construit en appelant sa méthode **add()** dans l'ordre suivant :

```
BinarySearchTree<Integer> b;
b = new BinarySearchTree<Integer>();
b.add(4);
b.add(8);
b.add(2);
b.add(6);
b.add(7);
b.add(5);
b.add(9);
b.add(1);
b.add(3);
```

A. Dans la boîte ci-dessous, dessinez l'arbre binaire désigné par la variable référence **b**.

B. Est-ce que l'arbre désigné par **b** est **balancé** ?
Oui ou **non**

C. Quelle est la **profondeur** de l'arbre désigné par **b** ?

D. Donnez l'ordre dans lequel les noeuds de l'arbre désigné par **b** seront visités en fonction de chacun des parcours suivants.

(a) Le parcours **préfixe** (pré-ordre) :

(b) Le parcours **infixe** (symétrique) :

(c) Le parcours **suffixe** (post-ordre) :

Question 6 : (4 points)

Analysez attentivement ce programme Java et donnez le résultat qui sera affiché sur la console. Supposez que l'interface **Stack** et la classe **LinkedStack** ont été définies et fonctionnent parfaitement, comme présentées en classe.

```
public class TestStack {

    public static void test(Stack<String> s) {

        Stack<String> tmp;
        tmp = new LinkedStack<String>();

        while (! s.isEmpty()) {
            tmp.push(s.pop());
        }

        s = tmp;
    }

    public static void main(String[] args) {

        Stack<String> s;
        s = new LinkedStack<String>();

        s.push("A");
        s.push("B");
        s.push("C");
        s.push("D");
        System.out.println("before:");
        System.out.println(s);
        test(s);
        System.out.println("after:");
        System.out.println(s);

    }
}
```

Complétez la sortie qui sera produite par l'exécution du programme ci-dessus.

```
before:
[A,B,C,D] <- top
```

Question 7 : (6 points)

Analysez attentivement ce programme Java et répondez à la question sur la page suivante.

```
public class LinkedList<E> {
    private static class Node<T> {
        private T value;
        private Node<T> next;
        private Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
        }
    }
    private Node<E> head;
    private int size;
    public int testRec(int i, int j) {
        return testRec(head, i, j);
    }
    private int testRec(Node<E> current, int i, int j) {
        if (j <= 0) {
            return 1;
        } else if (i <= 0) {
            return 1 + testRec(current.next, i, j-1);
        } else {
            return testRec(current.next, i-1, j-1);
        }
    }
    public void addLast(E elem) {
        if (elem == null) {
            throw new NullPointerException();
        }
        if (head == null) {
            head = new Node<E>(elem, null);
        } else {
            Node<E> p = head;
            while (p.next != null) {
                p = p.next;
            }
            p.next = new Node<E>(elem, null);
        }
        size++;
    }
    public String toString() {
        String res = "[";
        Node<E> cursor = head;
        while (cursor != null) {
            res = res + cursor.value;
            cursor = cursor.next;
            if(cursor != null) {
                res = res + ", " ;
            }
        }
        res = res + "]";
        return res;
    }

}
```

Étant donné l'implémentation de la classe **LinkedList** de la page précédente, donnez le résultat qui sera affiché sur la console lors de l'exécution du programme ci-dessous.

```
LinkedList<Integer> xs;
xs = new LinkedList<Integer>();
for (int i=0; i<10; i++) {
    xs.addLast(i);
}
System.out.println(xs);
System.out.println(xs.testRec(2,5));
```

Donnez votre réponse dans la boîte ci-dessous.

Question 8 : (10 points)

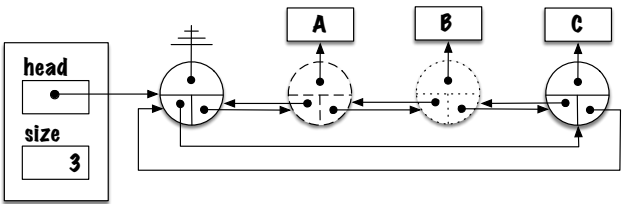
Implémentez la méthode d'instance **swapFirst** pour la classe **LinkedList** dont les caractéristiques sont les suivantes.

- La structure chaînée débute toujours par le noeud factice («*dummy node*») qui marque le début de la liste. Le noeud factice ne sauvegarde aucun élément d'information. La liste vide ne contient que le noeud factice.
- Pour cette question, les noeuds sont doublement chaînés.
- La liste est circulaire, ainsi la variable référence **next** du dernier noeud de la liste désigne le noeud factice et la variable référence **prev** du noeud factice désigne le dernier noeud de la liste. Dans le cas de la liste vide, les variables d'instance **prev** et **next** du noeud factice pointe vers le noeud factice lui-même.
- Puisqu'on accède facilement au dernier noeud de la liste en suivant la référence **prev** du noeud factice, il n'y a pas de référence **tail** dans cette implémentation.

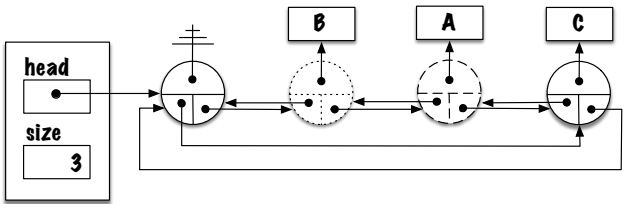
La méthode **swapFirst** échange l'ordre des deux premiers noeuds de la liste.

- Puisque cette question porte sur l'implémentation des structures chaînées, vous ne pouvez pas seulement échanger les valeurs, vous devez changer l'ordre des noeuds (objets de la classe **Node**).
- Également, vous ne pouvez pas utiliser les méthodes d'instance de la classe **LinkedList** . En particulier, vous ne pouvez pas utiliser les méthodes **add()** et **remove()**.
- Cette méthode lance l'exception **IllegalStateException** si la taille de la liste est inférieure à deux (2).

Voici le diagramme de mémoire de la liste avant et après l'appel à la méthode **swapFirst**. Nous avons utilisé deux types de lignes pour dessiner les deux premiers noeuds et bien identifier le travail à faire.



(avant)



(après)

```
public class LinkedList<E> implements List<E> {
   private static class Node<T> {
      private T value;
      private Node<T> prev;
      private Node<T> next;
      private Node(T value, Node<T> prev, Node<T> next) {
         this.value = value;
         this.prev = prev;
         this.next = next;
      }
   }
   private Node<E> head;
   private int size;

   public void swapFirst() {









  }
}
```