Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI 1121)
FINAL EXAMINATION

Instructors: Guy-Vincent Jourdan and Marcel Turcotte

April 2018, duration: 3 hours

Identification

Last name: ______________________ First name: ______________________________

Student #: ___________ Seat #: ____ Signature: ____________________ Section: A or B or C

Instructions

1. This is a closed book examination.
2. No calculators, electronic devices or other aids are permitted.
   (a) Any electronic device or tool must be shut off, stored and out of reach.
   (b) Anyone who fails to comply with these regulations may be charged with academic fraud.
3. Write your answers in the space provided.
   (a) Use the back of pages if necessary.
   (b) You may not hand in additional pages.
4. Do not remove pages or the staple holding the examination pages together.
5. Write comments and assumptions to get partial marks.
6. Beware, poor hand-writing can affect grades.
7. Wait for the start of the examination.

Marking scheme

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

Question 1 (5 marks)

Consider the Java program below.

```
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();
    }
}
```

Executing the code will output:

A.

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

B.

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

C.

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

D.

```
Saved!
Finished!
```

E. The code will not compile

F. Other, explain your answer in the box below.

Question 2 (10 marks)

Consider the Java code below and answer the 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();
}
```

For each block of code below, indicate the resulting outcome:

A.

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

(a) Prints **Alpha**

(b) Prints **Beta**

(c) Prints **Gamma**

(d) Does not compile

(e) Other

B.

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

(a) Prints **Alpha**

(b) Prints **Beta**

(c) Prints **Gamma**

(d) Does not compile

(e) Other

C.

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

(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

D.

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

(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

E.

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

(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

F.

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

(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

G.

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

(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

H.

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

(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

I.

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

(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

J.

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

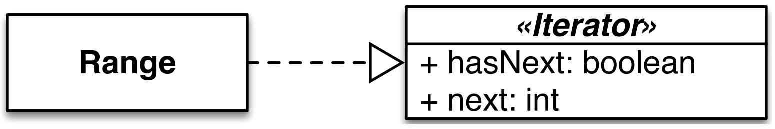
(a) Prints **Alpha**
(b) Prints **Beta**
(c) Prints **Gamma**
(d) Does not compile
(e) Other

Question 3 (5 marks)

You will find below an expression written in the **postfix** format. Using the algorithm seen in class, you need to give the corresponding expression written in the **infix** format.

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

Question 4 (15 marks)



An object of the class **Range** can be seen as an **Iterator**. For this question, the method **next** of the **Iterator** returns a value of type **int**.

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

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

On the next page provide a complete implementation of **Range** and **Iterator** so that the execution of the Java program above produces the following result on the console.

```
0
1
2
3
4
```

Follow all the following directives.

- The class **Range** has one constructor. It has a single parameter, which is of type **int**.
- The first valid call to the method **next** returns 0. Subsequent calls will return the next value in the iteration. The method can only be called the number of times specified by the value of the parameter of the constructor. If it is called more than that, it throws **IllegalStateException**.
- The method **hasNext** returns true if and only if a call to the method **next** would succeed (i.e. would not throw an exception).

Question 5 (10 marks)

A **binary search** tree is constructed by calling its **add()** method in the following orde r:

```
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. In the box below, draw the tree designated by the reference variable **b**.

B. Is **b** a **balanced** tree ?
Yes or **No**

C. What is the **depth** of the tree designated by **b**?

D. What will be the order in which the nodes of **b** are traversed following:

(a) The **pre-order** strategy:

(b) The **in-order** strategy:

(c) The **post-order** strategy:

Question 6 (4 marks)

Carefully analyze the following Java program and give the result printed out on the console. Assume that the interface **Stack** and the class **LinkedStack** have been defined and work as expected, i.e. as presented in class.

```
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);

    }
}
```

Complete the print out resulting from the execution of the code above.

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

Question 7 (6 marks)

Carefully analyze the following Java and answer the question on the next page.

```
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;
    }

}
```

Given the implementation of the class **LinkedList** from the previous page, give the result that will be printed out on the console when executing the program below:

```
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));
```

Give your answer in the box below:

Question 8 (10 marks)

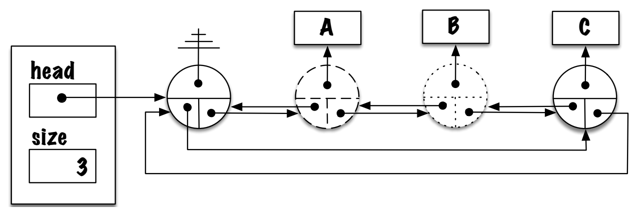
Implement the instance method **swapFirst** for the class **LinkedList** that has the following characteristics.

- An instance always starts off with a dummy node, which serves as a marker for the start of the list. The dummy node is never used to store data. The empty list consists of the dummy node only;
- In the implementation for this question, the nodes of the list are doubly linked;
- In this implementation, the list is circular, i.e. the reference **next** of the last node of the list is pointing at the dummy node, the reference **prev** of the dummy node is pointing at the last element of the list. In the empty list, the dummy node is the first and last node of the list, its references **prev** and **next** are pointing at the node itself;
- Since the last node is easily accessed, because it is always the previous node of the dummy node, the header of the list does not have (need) a tail pointer.

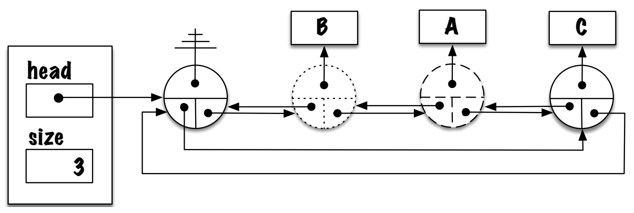
The method **swapFirst** exchanges the order of the first two nodes of the list.

- This question assesses your understanding of linked structures. Because of that, you cannot simply exchange the values. You need to exchange the order of the two objects of type **Node**.
- Likewise, you cannot use the methods of the class **LinkedList**. In particular, you cannot use the methods **add()** or **remove()**.
- The method throws the exception **IllegalStateException** if the size of the list is less than two (2).

Here is the memory diagram of the list before and after a call to the method **swapFirst**. We have used two different kinds of dotted lines to draw the first two nodes to identify the work to be done.



(before)



(after)

}

```
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() {









  }
}
```