Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI 1121)
Final Examination

Instructor: Marcel Turcotte

April 2013, duration: 3 hours

Identification

Surname: _____ First name: _____

Student #: _____ Seat #: _____ Signature: _____

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 comments and assumptions to get partial marks
4. Write your answers in the space provided
   (a) Use the back of pages if necessary
   (b) You may not hand in additional pages
5. Do not remove the staple holding the examination pages together
6. Beware, poor hand writing can affect grades

Marking scheme

Question	Maximum	Result
1	10	
2	10	
3	25	
4	15	
5	20	
6	10	
7	10	
Total	**100**	

Question 1 (10 marks)

A. For a variable of a primitive type, the value is found at the address designated by the identifier.
True or **False**

B. A constructor has a return value and the same name as its class.
True or **False**

C. An abstract class contains only abstract methods.
True or **False**

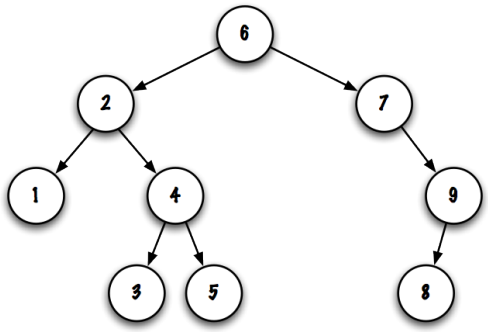
D. Redefining a method in a sub-class will hide the original method declaration in the super-class.
True or **False**

E. The keyword final for the declaration of a method implies that it cannot be redefined in a sub-class.
True or **False**

F. The value of the following (RPN) postfix expression is 67.
True or **False**

```
20 35 - 5 / 10 7 * +
```

G. When traversing the following binary search tree in post-order, the nodes are visited in the following order: 1, 3, 5, 2, 4, 8, 9, 7, 6.
True or **False**



H. When implementing the "head+tail" strategy presented in class for processing linked lists, the recursive method has always at least one more parameter than its public counterpart.
True or **False**

I. A method that throws unchecked exceptions must declare them using the keyword **throws**.
True or **False**

J. A nested non-static class has access to the methods and attributes of the outer class even if their visibility is private.
True or **False**

Question 2 (10 marks)

A. Which abstract data type is used for the syntactical analysis of an arithmetic expression?

(a) Queue

(b) List

(c) BinarySearchTree

(d) None of the above

Answer: d

B. Which of the following statements best describes the output of the program below.

(a) Displays "c = 0"

(b) Displays "c = Infinity"

(c) Displays "** caught Exception **", "c = 2"

(d) Displays "** caught ArithmeticException **", "c = 3"

(e) Produces a run-time error and displays a stack trace

(f) Gives a compile-time error: "exception java.lang.ArithmeticException has already been caught"

```
int a = 1, b = 0, c = 0;
try {
    c = a/b;
} catch ( Exception e ) {
    System.err.println( "** caught Exception **" );
    c = 2;
} catch ( ArithmeticException ae ) {
    System.err.println( "** caught ArithmeticException **" );
    c = 3;
}
System.out.println( "c = " + c );
```

Answer: f

C. For the **ArrayList** implementation of the interface **List**.

(a) Insertions at intermediate positions are always fast.

(b) Adding an element at the first position is always fast.

(c) Removing an element is always fast.

(d) Reading the value of an intermediate position is always fast.

Answer: d

D. For the **LinkedList** implementation of the interface **List**, adding an element at the rear of a singly-linked list can be made faster by:

(a) Adding the elements in the reverse order

(b) Adding a new instance variable to designate the last element of the list

(c) Adding a new instance variable to designate the second to the last node of the list

(d) (b) and (c)

(e) None of the above

Answer: b

E. Consider the implementation of the class **CircularQueue** below. Given a queue designated by **q** and containing the following elements: A, B, C, D, E, F, G, where A is the front element of the queue, following the call, **q.magic(4)**, what will be the content of the queue?

(a) E, F, G

(b) A, B, C, D

(c) E, F, G, A, B, C, D

(d) E, F, G, A, B, C, D, A, B, C, D

(e) None of the above

Answer: c

```
public class CircularQueue<E> {
    private E[] elems;
    private int front;
    private int rear;

    public CircularQueue(int capacity) {
        if (capacity < 0) {
            throw new IllegalArgumentException("negative number");
        }
        elems = (E[]) new Object[capacity];
        front = -1;
        rear = -1;
    }

    public void magic(int n) {
        if (rear != -1 && rear != front) {
            while (n > 0) {
                E current = elems[front];
                elems[front] = null;
                front = (front + 1) % elems.length;
                rear = (rear + 1) % elems.length;
                elems[rear] = current;
                n--;
            }
        }
    }
}
```

Question 3 (25 marks)

You must implement a software system for managing reference letters for the University of Ottawa. Candidates applying for the master or the doctorate degree must provide recommendation letters from former professors or employers. Write the implementation of the classes **Person**, **Student**, **Reference** and **Letter**. Make sure to add all the necessary constructors. Each attribute must have a getter method. Here is a test program to illustrate the use of these classes. In particular, make sure that your implementation would work for this test.

```
Person student, reference1, reference2;

student = new Student("Lionel ", "cr7@uottawa.ca", "incognito", "432534", 9.2);
reference1 = new Reference("Pepe", "pepe@uottawa.ca", "wrestling","IEEE");
reference2 = new Reference("Mourinho", "specialone@uottawa.ca","psgmission", "SITE");

Letter letter1 = new Letter("Student is excellent ", 5, 5, 5);
Letter letter2 = new Letter("A hard worker student, very innovative", 5, 3, 4);

student.add(letter1);
student.add(letter2);

reference1.add(letter1);
reference2.add(letter2);

for (Letter aLetter : student.getLetters()){
    if (aLetter != null) {
        System.out.println(aLetter);
    }
}
```

Executing the above test program will produce the following output.

```
Comment: Student is excellent; Performance = 5.0; Originality = 5.0;
  Potential = 5.0.
Comment: A hard worker student, very innovative; Performance = 5.0; Originality = 3.0;
  Potential = 4.0.
```

For this question, you are allowed to use the predifined class **java.util.ArrayList**. In particular, it has a constructor **ArrayList()**, stores an arbitrarily large number of elements, and implements the following methods.

```
public interface List<E> {
    // Appends the specified element to the end of this list
    public abstract boolean add(E o);
    // Returns the element at the specified position in this list.
    public abstract E get(int index);
    // Replaces the element at the specified position with the specified element
    public abstract E set(int index, E element);
    // Returns an array containing all of the elements in this list in proper order
    public abstract Object[] toArray();
    // Returns the number of elements in this list
    public abstract int size();
}
```

A. An object of the class **Person** has attributes for saving the name of the person, its E-mail address and password. A **Person** holds several letters. Make sure to include at least one constructor, as well as the necessary access methods.

```
// MODEL

import java.util.List;
import java.util.ArrayList;

public class Person {

    private String name, email, password;

    private List< Letter> letters;

    public Person (String name, String email, String password) {
        this.name = name;
        this.email = email;
        this.password = password;
        letters = new ArrayList< Letter >();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void add(Letter letter) {
        letters.add(letter);
    }

    public Letter get(int index) {
        return letters.get(index);
    }

    public Letter[] getLetters() {
        return (Letter[]) letters.toArray();
    }
```

```
}
```

B. A **Student** is a **Person** that also has a student number and an admission average. A student holds several letters. Make sure to include at least one constructor, as well as the necessary access methods.

```
// MODEL

public class Student extends Person {

    private String studentID;
    private double gpa;

    public Student(String name, String email, String password,
                                                String studentID, double gpa) {
        super(name, email, password);
        this.studentID = studentID;
        this.gpa = gpa;
    }

    public String getStudentID() {
        return studentID;
    }

    public double getGpa() {
        return gpa;
    }

}
```

C. A **Reference** is a **Person** that belongs to an organization. A **Reference** may write several letters (one per student). Make sure to include at least one constructor, as well as the necessary access methods.

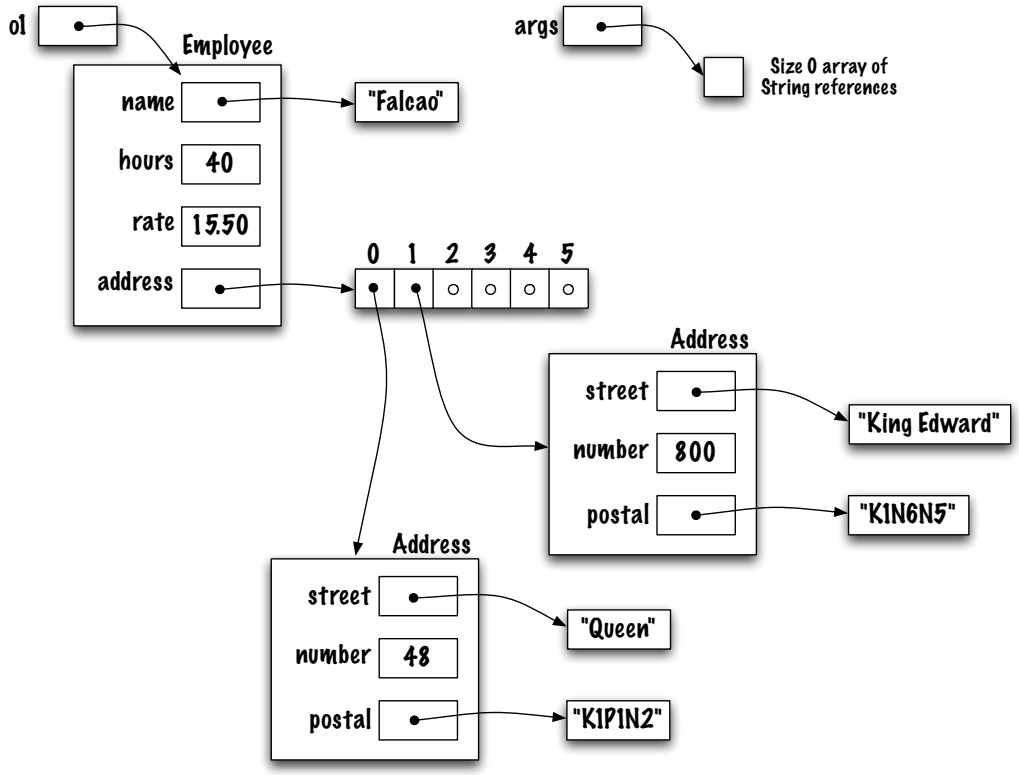
```
// MODEL

public class Reference extends Person {

    private String institution;

    public Reference(String name, String email, String password, String institution) {

        super(name, email, password);
        this.institution = institution;
    }

    public String getInstitution() {
        return institution;
    }

    public void setInstitution(String institution) {
        this.institution = institution;
    }

}
```

D. A recommendation **Letter** has an attribute to store general comments on the student, as well as rankings (0 (weak) to 5 (strong)) for academic performance, originality and research potential.

```
// MODEL

public class Letter {

    private String comment;
    private double performance, originality, potential;

    public Letter(String comment, int performance, int originality, int potential) {
        this.comment  =comment;
        this.performance = performance;
        this.originality= originality;
        this.potential = potential;
    }

    public String getComment() {
        return comment;
    }

    public double getPerformance() {
        return performance;
    }

    public double getOriginality() {
        return originality;
    }

    public double getPotential() {
        return potential;
    }

    public String toString() {
        StringBuffer str = new StringBuffer();
        str.append("Comment");
        str.append(": ");
        str.append(comment);
        str.append("; ");
        str.append("Performance = ");
        str.append(performance);
        str.append("; Originality = ");
        str.append(originality);
        str.append("; Potential = ");
        str.append(potential);
        return  str.toString();
    }

}
```

Question 4 (15 marks)

Reverse engineer the memory diagrams below. Specifically, give the implementation of all the classes, instance variables, and constructors.




```
// MODEL

public class Address {

        private String street;
        private String postal;

        private int number;

        public Address(String street, int number, String postal){
                this.street = street;
                this.number = number;
                this.postal = postal;
        }

}
```

```
// MODEL

public class Employee {

        private String name;
        private Address[] addresses;

        private int hours;
        private double rate;

        public Employee(String name, int hours, double rate){
                this.name = name;
                this.hours = hours;
                this.rate = rate;
                this.addresses = new Address[6];
                this.size= 0 ;
        }

        public void set( int, pos Address address ) {
                addresses[pos] = address ;
        }

        public static void main(String[] args) {
                Employee o1;

                o1 = new Employee("Falcao", 40, 15.50);

                o1.set(0, new Address("Queen", 48, "K1P 1N2"));
                o1.set(1, new Address("King Edward", 800, "K1N6N5"));
        }
}
```

Question 5 (20 marks)

The abstract data type **Deque** — pronounced "deck" — combines features of both a queue and a stack. In particular, a **Deque** ("Double-Ended QUEue") allows for

- efficient insertions at the front or rear;
- efficient deletions at the front or the rear.

Here are the descriptions of the four methods of this class.

- **void offerFirst(E item)**: adds an item at the front of this **Deque**;
- **void offerLast(E item)**: adds an item at the rear of this **Deque**;
- **E pollFirst()**: removes and returns the front item of this **Deque**, returns **null** if this **Deque** was empty;
- **E pollLast()**: removes and returns the rear item of this **Deque**, returns **null** if this **Deque** was empty.

Below, you will find a partial implementation of the class **LinkedDeque** that uses linked elements to store its elements.

- The linked elements are doubly-linked;
- A **LinkedDeck** has two instance variables, **front** and **rear**.

```
public class LinkedDeque<E> implements Deque<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;
        }
    }

    LinkedDeque() {
        front = null;
        rear = null;
    }

    private Node<E> front;
    private Node<E> rear;
```

Hint: Draw the memory diagrams. Consider all the special cases.

```
// MODEL

    public void offerFirst(E e) {

        front = new Node<E>(e, null, front);

        if (rear == null) {
            rear = front;
        } else {
            front.next.prev = front;
        }

    } // End of offerFirst
```

```
// MODEL

    public void offerLast(E e) {

        Node<E> newNode = new Node<E>(e, rear, null);

        if (rear == null) {
            front = newNode;
            rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }

    } // End of offerLast
```

```
// MODEL

    public E pollFirst() {

        if (front == null) {
            return null;
        }

        E saved = front.value;

        if (front == rear) { // front cannot be null
            front = null;
            rear = null;
        } else {
            front = front.next;
            front.prev = null;
        }

        return saved;

    } // End of pollFirst
```

```
// MODEL

    public E pollLast() {

        if (front == null) {
            return null;
        }

        E saved = rear.value;

        if (front == rear) { // front cannot be null
            front = null;
            rear = null;
        } else {
            rear = rear.next;
            rear.next = null;
        }

        return saved;

    } // End of pollLast
} // End of LinkedDeque
```

Question 6 (10 marks)

In the class **LinkedList** below, implement the method **LinkedList<E> partition(E elem)**. This instance method partitions this list in two parts. This intance retains all the **leftmost** elements of this list up to and including the first occurrence of **elem**. The rest of the elements are returned as a new list. If **elem** is not found in the list, then **this** list remains intact and the returned list is empty.

For instance, let **xs** designate a list containing the values **1, 2, 3, 4, 3, 5, 6**. After the call **ys = xs.partition(3)**, the list designated by **xs** contains the elements **1, 2, 3**, whereas **ys** now designates a list containing the elments **4, 3, 5, 6**.

The method must be implemented following the technique presented in class for implementing recursive methods inside the class, i.e. where a recursive method is made of a public part and a private recursive part, which we called the helper method. The public method initiates the first call to the recursive method.

```
public class LinkedList<E> {

    private static class Node<E> {

        private E value;
        private Node<E> next;

        private Node(E value, Node<E> next) {
            this.value = value;
            this.next = next;
        }
    }

    private Node<E> head = null;
```

```
    public LinkedList<E> partition(E elem) {

        // MODEL

        return partitionRec(head, elem);

    } // End of partition
```

```
    // MODEL

    private LinkedList<E> partitionRec( Node<E> p, E elem  ) {

        LinkedList<E> result;

        if (p == null) {

            result = new LinkedList<E>();

        else if (p.value.equals(elem)) {

            result = new LinkedList<E>();
            result.head = p.next;
            p.next = null;

        } else {

            result = partitionRec(p.next, elem);

        }

        return result;

    } // End of partitionRec
} // End of LinkedList
```

Question 7 (10 marks)

Implement the method **int countIf(E elem)** for the binary search tree presented in class. The method returns the number of elements in the tree that are greater than **elem**.

- The elements stored in a binary search tree implement the interface **Comparable<E>**. Recall that the method **int compareTo(E other)** returns a negative integer, zero, or a positive integer as the instance is less than, equal to, or greater than the specified object.
- A method that is visiting too many nodes will get a maximum of 9 marks.
- Given a binary search tree, **t**, containing the values **1, 2, 3, 4, 5, 6**, the call **t.countIf(2)** returns the value **4**.

```
public class BinarySearchTree<E extends Comparable<E> > {

    private static class Node<E> {

        private E value;

        private Node<E> left;
        private Node<E> right;

        private Node( E value ) {
            this.value = value;
            left = null;
            right = null;
        }
    }

    private Node<E> root = null;

    // MODEL

    public int countIf(E elem) {
        return countIf(root, elem);
    }

    private int countIf(Node<E> current, E elem) {

        int count = 0;

        if (current != null) {

            count = countIf(current.right, elem);

            if (elem.compareTo(current.value)<0) {
                count = cout + countIf(current.left, elem) + 1;
            }

        }

        return count;
    }

    // MODEL 2 (9/10) based on in-order traversal
```

```
public int countIf(E elem) {
    return countIf(root, elem);
}

private int countIf(Node<E> current, E elem) {

    int count = 0;

    if (current != null) {

        count = countIf(current.right, elem);

        if (elem.compareTo(current.value)<0) {
            count = cout + 1;
        }

        count = count + countIf(current.left, elem);

    }

    return count;
}
```