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 to Computing II (ITI 1121)
Final Examination

Instructors: Oana Frunza and Rafael Falcon

April 15th, 2012. Duration: 3 hours

Identification

Last name, first name: ______________________________

Student number: ______________ Signature: ______________________

Instructions

1. **Read these instructions;**
2. This is a closed book examination;
3. No calculators or other aids are permitted;
4. Write comments and assumptions to get partial marks;
5. Beware, poor hand writing can affect grades;
6. Do not remove the staple holding the examination pages together;
7. Write your answers in the space provided.
   Use the back of pages if necessary.
   You may **not** hand in additional pages.

Marking scheme

Question	Maximum	Result
1	10	
2	10	
3	20	
4	20	
5	8	
6	13	
7	12	
8	7	
Total	**100**	

Question 1: (10 marks)

A. In Java, there are two ways of comparing objects, by identity or by content. Which of the following two statements compares objects by identity?

(a) **x == y**

(b) **x.equals(y)**

B. In Java, there are two paradigms allowing for the creation of data structures that can be used to store objects of various types, generics or inheritance. Which of the following two paradigms allows for the detection of type errors at compile time?

(a) Inheritance

(b) Generics

C. The statement "**cmp = eq(i, j)**" below produces a compile-time error.
True or **False**.

```
public class Test {
    public static boolean eq( int a, int b ) {
        boolean result;
        if ( a == b ) {
            result = true;
        } else {
            result = false;
        }
        return result;
    }
    public static void main( String[] args ) {
        long i, j;
        i = 5;
        j = 10;
        boolean cmp = eq( i, j );
    }
}
```

D. Two or more methods in a class may have the same name as long as their return types are different.
True or **False**

E. The following program displays **true**.
True or **False**

```
public class Counter {
    private static int value = 0;
    public void incr() {
        value = value+1;
    }
    public int getValue() {
        return value;
    }
    public static void main( String[] args ) {
        Counter a, b;
        a = new Counter();
        a.incr();
        b = new Counter();
        b.incr();
        System.out.println( b.getValue() == 1 );
    }
}
```

F. A constructor cannot have a return value/type.
True or **False**

G. A reference variable of type **T**, where **T** is an interface, can reference an object of class **S** if **S**, or one of its superclasses, implements **T**.
True or **False**

H. The name **this** refers to a reference that is always available to an instance method and refers to the object itself.
True or **False**

I. A method that throws exceptions of the class **RuntimeException**, or one of its subclass, **must** declare this exception using a clause **throws**.
True or **False**

J. In a binary search tree, duplicated values are allowed.
True or **False**.

Question 2: (10 marks)

A. What term is used to refer to a method that is automatically executed when an object of a class is created?

(a) Setter
(b) Getter
(c) Initializer
(d) Constructor
(e) Mutator

Answer:

B. The ____________________ of an object define(s) its potential behaviors.

(a) attributes
(b) white spaces
(c) variables
(d) methods
(e) names

Answer:

C. In a given class, if a local variable has the same name as an instance variable:

(a) This causes a compile-time error
(b) This causes a run-time error
(c) The local variable masks the instance variable
(d) The instance variable masks the local variable

Answer:

D. Which of the keywords below indicates that a new class is being derived from an existing class?

(a) super
(b) final
(c) extends
(d) inherits
(e) expands

Answer:

E. Regarding the **ArrayList** implementation of the interface **List**.

(a) Inserting an element at a random location with this implementation is always fast.
(b) Adding an element in the first position of the list is always fast with this implementation.
(c) Retrieving an element from a random location is always fast with this implementation.
(d) Removing the first element is always fast with this implementation.

Answer:

Question 3: (20 marks)

A. Following the guidelines presented in class, as well as the lecture notes, draw the **memory diagrams** for all the objects, all the local variables, and parameter of the method **Property.main before** the execution of the statement "aProperty = null".

```
public class Property {

    private String name;
    private double value;

    public Property( String name, double value ) {
        this.name = name;
        this.value = value;
    }

    public static void main( String[] args ) {

        Property aProperty;
        String aName;
        double aValue;

        aName = new String( "pi" );
        aValue = 3.14159265;

        aProperty = new Property( aName, aValue );

        aProperty = null;

    }
}
```

B. Identify five (5) compile-time warnings/errors in the Java program below.

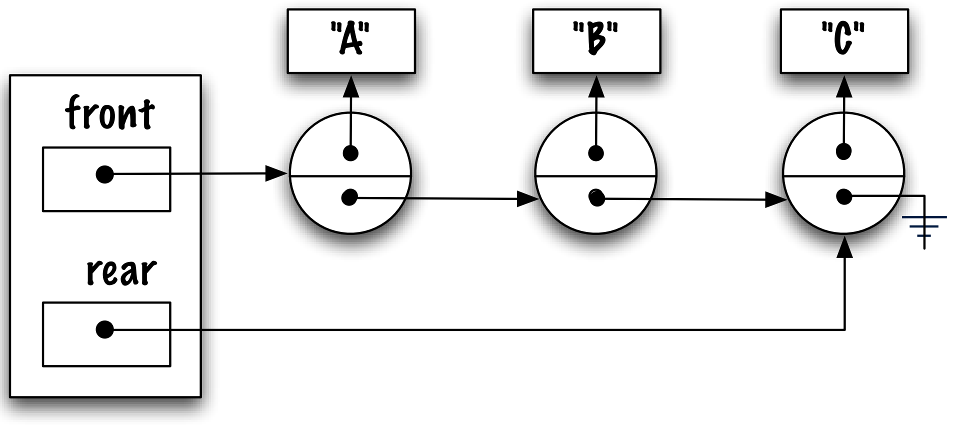
```
public class LinkedStack<E> {

    private static class Node<T> {

        private E value;
        private Node<T> next;

        private Node( T item, Node next ) {
            value = item;
            this.next = next;
        }

    }

    private Node<E> head;

    head == null;

    public static void main( String[] args ) {

        Node<E> p;
        p = head;

        while (p != null ) {
            System.out.println( p.value );
            p.next();
        }

    }

}
```

C. Given the following partial declaration of the class **LinkedQueue**.

```
public class LinkedQueue {

    private static class Node {
        private String value;
        private Node next;
        private Node( String value, Node next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Node front, rear;

    // ...
}
```

Modify the memory diagram below to represent the content of the memory after the execution of the following statement:

```
rear.next = new Node( "D", rear );
```



D. Analyze the following Java program and indicate what its output will be:

```
 1 public class Test {
 2     public static void displayRatio( int a, int b ) {
 3         if ( b == 0 ) {
 4             throw new IllegalArgumentException( "zero" );
 5         }
 6         try {
 7             System.out.println( "displayRatio: ratio is " + (a/b) );
 8         } catch( IllegalArgumentException e1 ) {
 9             System.out.println( "displayRatio: caught IllegalArgumentException" );
10         } catch( ArithmeticException e2 ) {
11             System.out.println( "displayRatio: caught ArithmeticException" );
12         }
13     }
14     public static void main( String[] args ) {
15         try {
16             displayRatio( 5, 0 );
17         } catch (RuntimeException e) {
18             System.out.println( "main: caught RuntimeException: " + e );
19         }
20     }
21 }
```

(a) Displays: “main: caught RuntimeException: java.lang.IllegalArgumentException: zero”

(b) The program terminates abruptly and displays the following stack trace:

```
Exception in thread "main" java.lang.IllegalArgumentException: zero
        at Test.displayRatio(Test.java:4)
        at Test.main(Test.java:16)
```

(c) The program terminates abruptly and displays the following stack trace:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.displayRatio(Test.java:7)
        at Test.main(Test.java:16)
```

(d) Displays “displayRatio: ratio is (5/0)”

(e) Displays “displayRatio: caught IllegalArgumentException”

(f) Displays “displayRatio: caught ArithmeticException”

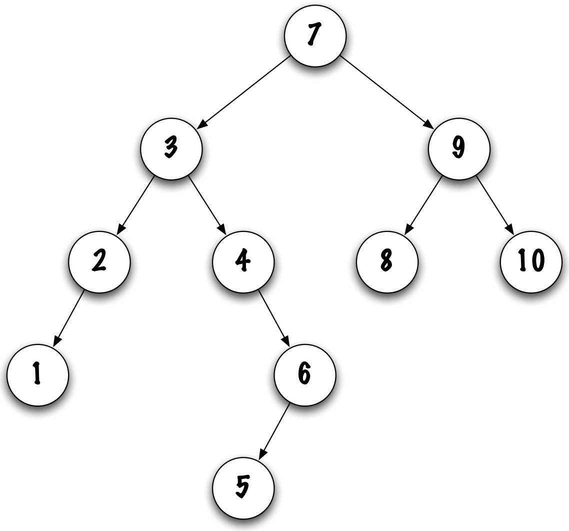
E. Complete the implementation of the recursive instance method **sum** using the "head+tail" strategy presented in class. The method **sum** returns the sum of all the elements of the list. In particular, the execution of the main method would display the value 10.

```
public class LinkedList {

    private static class Node {
        private int value;
        private Node next;
        private Node( int value, Node next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Node head;

    public void addFirst( int elem ) {
        head = new Node( elem, head );
    }

    public int sum() {

        int result = ______________________________;

        return result;
    }

    private _______________ sum( ______________________________ ) {

        _______________ result;

        if ( ______________________________ ) { // Base case

            result = ______________________________;

        } else { // General case

            _______________ s = ______________________________; // Recursion

            result = ______________________________;
        }
        return result;
    }

    public static void main( String[] args ) {
        LinkedList l;
        l = new LinkedList();
        for (int i=0; i<5; i++) {
            l.addFirst(i);
        }
        System.out.println( l.sum() );
    }

}
```

F. The following implementation corresponds to the **add** method for a binary search tree:

```
private boolean add( Integer obj, Node current ) {

    boolean result;
    int test = obj.compareTo( current.value );

    if ( test == 0 ) {
        result = false; // already exists, not added
    } else if ( test < 0 ) {
        if ( current.left == null ) {
            current.left = new Node( obj );
            result = true;
        } else {
            result = add( obj, current.left );
        }
    } else {
        if ( current.right == null ) {
            current.right = new Node( obj );
            result = true;
        } else {
            result = add( obj, current.right );
        }
    }
    return result;
}
```

By repeatedly invoking the **add** method with different integer values, the binary search tree displayed to the left has been generated. Write in the box to the right the sequence in which the integer numbers were added to the tree (e.g.: 1 2 3 4 5 6 7 8 9 10).



Question 4: (20 marks)

There are 52 cards in a deck; each belongs to one of four suits and one of 13 ranks. The suits from the lowest to the highest importance are: *Spades, Hearts, Diamonds* and *Clubs*. The ranks are: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King. The suits and ranks are mapped to two arrays of values as follows:

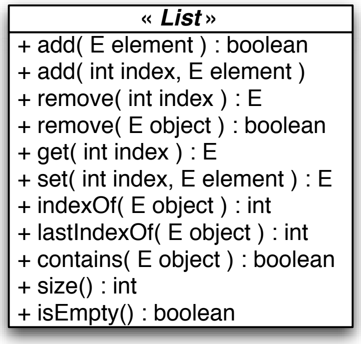
- int [ ] suits = { 1, 2, 3, 4 }; where 1 maps Spades; 2 maps Hearts; 3 maps Diamond; 4 maps Clubs
- int [ ] ranks = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 }; where 1 maps Ace; 2 maps 2; ... 11 maps Jack, ...

Write the Java implementation of the classes **Card**, **Deck**, and **Hand** following the instructions below.

A. The class **Card** implements **Comparable<Card>** and has fields for holding the **rank** and **suit** information of a card. Implement the method: **public int compareTo(Card other)** that returns -1, 0, or 1 as this object is less than, equal to, or greater than **other**. Make sure to include at least one constructor, as well as appropriate getter and setter methods.

B. The class **Deck** uses the **ArrayList** implementation of the abstract data type **List** to store the 52 cards. The class has a constructor that takes no parameter and creates a deck of 52 playing cards. The deck is set up in the order: *Spades, Hearts, Diamonds* and *Clubs*. Implement the methods:

- **public void shuffle()** that randomly shuffles a deck of cards. Each card in the deck will be shuffled with a randomly selected card from the deck; **Hint: Use the method random() of the java.lang.Math class, which returns a value of type double in the range [0; 1).**
- **public Card dealCard()** removes the last card from the deck and returns it. The method will throw an **IllegalStateException** object if the deck is empty.



```
« List »
+ add( E element ) : boolean
+ add( int index, E element )
+ remove( int index ) : E
+ remove( E object ) : boolean
+ get( int index ) : E
+ set( int index, E element ) : E
+ indexOf( E object ) : int
+ lastIndexOf( E object ) : int
+ contains( E object ) : boolean
+ size() : int
+ isEmpty() : boolean
```

C. The class **Hand** uses the ArrayList implementation of the abstract data type **List** to store the cards in the hand. Implement the method: **public int total ()** that returns the total number of cards in the hand.

Question 5: (8 marks)

For this question, there is an interface called **Queue** and its implementation, a class called **MysteryQueue**.

```
public interface Queue<E> {
    public abstract boolean isEmpty();
    public abstract void enqueue( E value );
    public abstract E dequeue();
}
```

The class **MysteryQueue** implements the interface **Queue**. The constructor (**public MysteryQueue()**) creates an empty queue. The implementation can hold an arbitrarily large number of elements. Make no other assumption about this implementation; in particular, you cannot assume that it uses an array or a linked-list.

For the class **Utils** below, write a class (static) method that returns true if the parameters **q1** and **q2** designate queues having the same elements in the same order, and false otherwise. Furthermore, the queues designated by the parameters **q1** and **q2** must not be changed (i.e. after the method call, they must contain the same elements, in the same order, as before the call).

```
public class Utils {
    public static <E> boolean eq(Queue<E> q1, Queue<E> q1) {

    } // End of the method eq
} // End of the class Utils
```

Question 6: (13 marks)

Implement the method **remove(int from, int to)** for the class **LinkedList**. This instance method removes all the elements in the specified range from this list and returns a new list that contains all the removed elements, in their original order. The implementation of **LinkedList** 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 **previous** 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 **previous** 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.

Example: if **xs** is a reference designating a list containing the following elements [**a,b,c,d,e,f**], after the method call **ys = xs.remove(2,3)**, the list designated by **xs** contains [**a,b,e,f**], and **ys** designates a list containing [**c,d**].

Write your answer in the class **LinkedList** on the next page. **You cannot use the methods of the class LinkedList. In particular, you cannot use the methods add() or remove().**

Hint: draw detailed memory diagrams.

```
public class LinkedList<E> {
    private static class Node<T> { // implementation of the doubly linked nodes
        private T value;
        private Node<T> previous;
        private Node<T> next;
        private Node( T value, Node<T> previous, Node<T> next ) {
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }
    private Node<E> head;
    private int size;
    public LinkedList() {
        head = new Node<E>( null, null, null );
        head.next = head.previous = head;
        size = 0;
    }

    public LinkedList<E> remove( int from, int to ) {




    } // End of remove
} // End of LinkedList
```

Question 7: (12 marks)

This question is about circular queue and iterator. The class **CircularQueue** uses the implementation technique called "circular array" to implement a fixed-size queue. This class also provides an implementation of an iterator.

```
public interface Iterator<E> {
    // Returns the next element in the iteration.
    public abstract E next();

    // Returns true if the iteration has more elements.
    public abstract boolean hasNext();
}
```

- This implementation uses a **fixed-size circular array**;
- An object of the class **CircularQueue** has a method **iterator** that returns an object of the class **CircularQueueIterator**. This class implements the interface **Iterator**;
- A call to the method **hasNext** of an iterator object returns **true** if there are more elements (cyclically) in the queue, and **false** otherwise;
- A call to the method **next** of an iterator object returns the next element in the queue (cyclically). Specifically, the first call to **next** returns the front element, the second call returns the element immediately after the first element, etc. The value of the "current" index of the iterator must wrap around the fixed-size array in the class **CircularQueue**. Eventually, a call to the method **next** will return the rear element of the queue. At this point, a call to the method **hasNext** returns **false** and a call to the method **next** will cause **NoSuchElementException** to be thrown.

Complete the implementation of the class **CircularQueue** given below.

```
public class CircularQueue<E> implements Queue<E> {

    private static final int DEFAULT_CAPACITY = 100;
    private int front, rear, size;
    private E[] elems;

    public CircularQueue( int capacity ) {
            elems = (E[]) new Object[ capacity ];
            front = 0;
            rear = -1;
            size = 0;
    }

    // continues on the next page
    public boolean isEmpty() {
            return ( size == 0 );
    }

    public void enqueue( E value ) {
            rear = ( rear+1 ) % elems.length;
            elems[ rear ] = value;
            size = Math.min( size + 1, elems.length ) ;
    }

    public E dequeue() {
        E savedValue = elems[ front ];
        elems[ front ] = null;
        size--;
        front = ( front+1 ) % elems.length;
        return savedValue;
    }

    private ___________ CircularQueueIterator implements Iterator<E> {

        private ___________ current = ___________;

        public E next() {

            if ( __________________________________ ) {
                throw new NoSuchElementException();
            }











            return ______________________;
        }

        public boolean hasNext() {
            boolean result;

            result = __________________________________________________;

            return result;
        }
    } // End of CircularQueueIterator


    public _________________ iterator() {

        return __________________________________;
    }

} // End of CircularQueue
```

Question 8: (7 marks)

For the class **BinarySearchTree** below, write an implementation for the instance method **public int getHeight(E elem)** that returns the number of links to follow from the root to the node containing the value **elem**, or -1 if **elem** was not found in this tree.

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

    private static class Node<F extends Comparable<F> > {

        private F value;
        private Node<F> left;
        private Node<F> right;

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

    private Node<E> root = null;

























}
```

(blank space)