Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

Introduction to Computing II (ITI 1121)
Final Examination

Instructor: Marcel Turcotte

April 2011, 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. The statement "**cmp = compare(i, j)**" below will produce a compile-time error.
True or **False**.

```
public class Test {
    public static int compare( long a, long b ) {
        return a < b ? -1 : ( a == b ) ? 0 : 1;
    }
    public static void main( String[] args ) {
        int i, j, cmp;
        i = 5;
        j = 10;
        cmp = compare( i, j );
    }
}
```

B. If **p** is of type **int**, the test of the **if** statement below will produce a compile-time error.
True or **False**.

```
if ( p == null ) {
    System.out.println( "is empty" );
}
```

C. Two or more methods in a class may have the same name, as long as the types of return values are different.
True or **False**

D. Each instance of a class has its own set of instance variables.
True or **False**

E. When you write a constructor for a class, the default constructor that Java automatically provides is no longer present.
True or **False**

F. A reference variable of type **T** can reference an object of class **T** or any of its superclasses.
True or **False**

G. You are not required to catch exceptions that inherit from the class **RuntimeException**.
True or **False**

H. The **throws** clause causes an exception to be thrown.
True or **False**

I. In a singly linked list implementation, the reference **tail** facilitates the implementation of the method **addLast**.
True or **False**.

J. In a binary search tree, duplicated values are not allowed.
True or **False**.

Question 2: (10 marks)

A. The name of a reference variable that is always available to an instance method and refers to the object itself.

(a) self

(b) this

(c) object

(d) instance

(e) me

B. A method in a subclass that has the same signature as a method in the superclass is

(a) overloading

(b) overriding

(c) chaining

(d) an error

C. All the classes directly or indirectly inherit from this class.

(a) Object

(b) Class

(c) Instance

(d) Root

(e) Super

D. To remove the first node in a nonempty singly linked list, with no dummy node,

(a) move the successor reference in the **head** node one node forward:

```
head.next = head.next.next;
```

(b) set a reference **pred** to the predecessor of the node you want to remove, and set the successor of **pred** to the successor of the **head**

(c) move the **head** reference one node forward:

```
head = head.next;
```

(d) delete the node by setting the **head** reference to null:

```
head = null;
```

E. For the implementation of a queue using singly linked nodes,

(a) **front** designates the first element and **rear** designates the last element;

(b) **rear** designates the first element and **front** designates the last element;

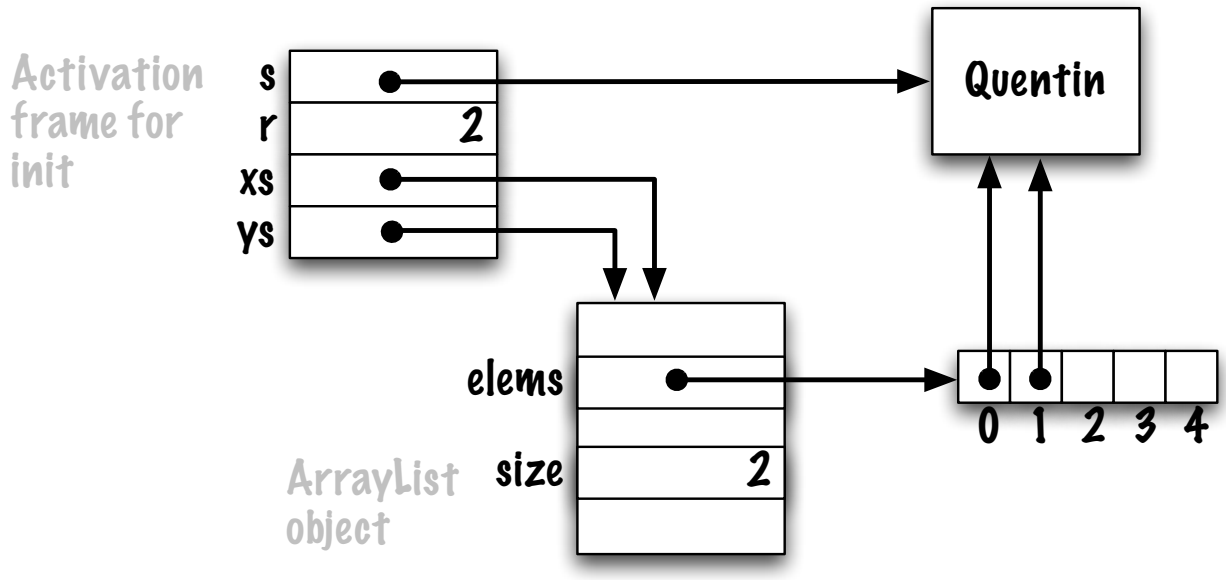
(c) It does not matter, (a) and (b) would both lead to efficient implementations;

(d) None of the above.

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 and all the local variables of the method **ArrayList.init** following the execution of the statement "**ys = xs**".

```
public class ArrayList<E> {

    private E[] elems;
    private int size;

    public ArrayList( E value, int range, int capacity ) {
        elems = (E[]) new Object[ capacity ];

        for ( int i=0; i<range; i++ ) {
            elems[ i ] = value;
        }

        size = range;
    }

    public static void init() {
        String s;
        s = new String( "Quentin" );

        int r;
        r = 2;

        ArrayList<String> xs, ys;
        xs = new ArrayList<String>( s, r, 5 );

        ys = xs;
    }

}
```



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

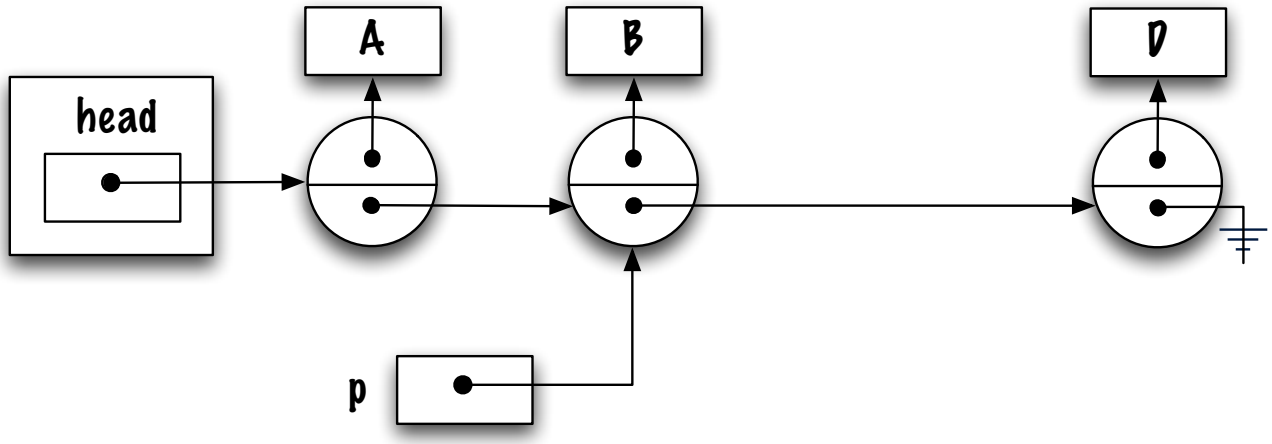
```
public class LinkedList {

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

    }

    private Node<E> head;

    public static void main( String[] args ) {

        private Node<E> p;

        p = head;

        while ( p != 0 ) {
             System.out.println p.value;
             p++;
        }

    }

}
// SOLUTION:
public class LinkedList<E> { // 1. missing type parameter

    private static class Node<E> {
      private E value;
      private Node<E> next;
      private Node( E value, Node<E> next ) { // 2. no return type (constructor)
          this.value = value;
          this.next = next;
      }

    }

    private Node<E> head;

    public static void main( String[] args ) {

        Node<E> p; // 3. can't have a visibility modifier for a local variable

        p = head; // 4. cannot access head from static context

        while ( p != null ) { // 5. p is of type Node, can't compare to 0
             System.out.println( p.value ); // 6. missing ()
             p = p.next; // 7. p is of type Node, cannot use ++ operator
        }

    }
}
```

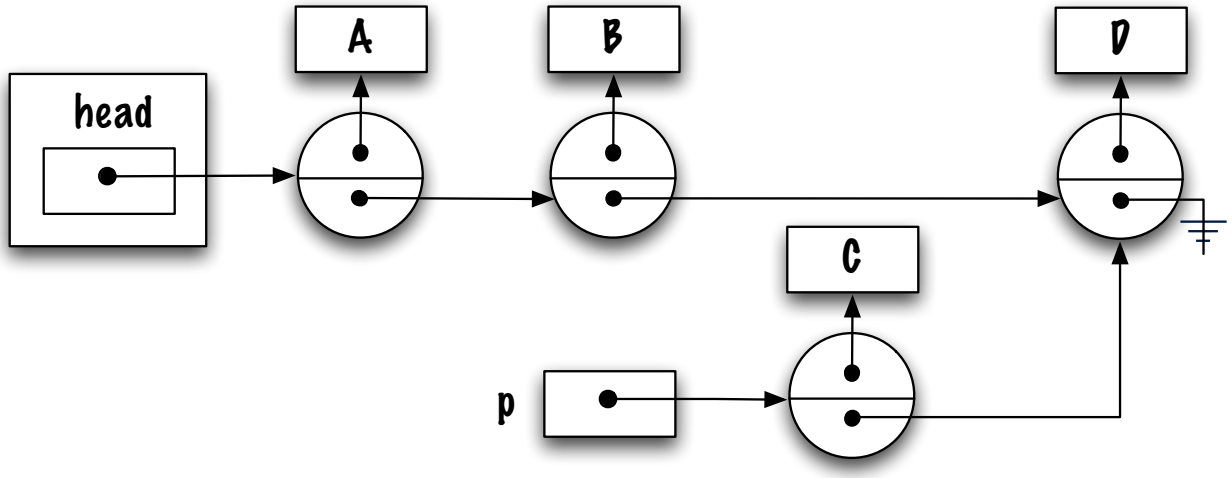
C. Given the following partial declaration of the class **LinkedList**.

```
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;

    // ...
}
```

Modify the memory diagram below to represent the content of the memory after the execution of the following statement [1]:

```
p = new Node<E>( C, p.next );
```





[1] Assume that **C** is reference variable designating an object.

D. Study the following Java program and tell what the program will output when run:

```
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( "ratio is " + (a/b) );
8          } catch( IllegalArgumentException e1 ) {
9              System.out.println( "caught IllegalArgumentException" );
10         } catch( ArithmeticException e2 ) {
11             System.out.println( "caught ArithmeticException" );
12         }
13     }
14     public static void main( String[] args ) {
15         displayRatio( 5, 0 );
16     }
17 }
```

(a) 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:15)
```

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

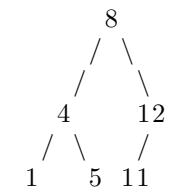
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.displayRatio(Test.java:7)
        at Test.main(Test.java:15)
```

(c) Displays "ratio is (5/0)"

(d) Displays "caught IllegalArgumentException"

(e) Displays "caught ArithmeticException"

E. Draw the binary search tree that results from inserting the following elements, in that order, when using the method add presented in class: 8, 12, 11, 4, 5, 1.



F. Beware, the following Java program has a bug! Tell what it will output when run.

```
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;
    public void addFirst( E elem ) {
        head = new Node<E>( elem, head );
    }
    public int size() {
        return size( head );
    }
    private int size( Node<E> current ) {
        int length = 0;
        if ( current == null ) {
            length = 0;
        } else {
            size( current.next );
            length++;
        }
        return length;
    }
    public static void main( String[] args ) {
        LinkedList<Integer> l;
        l = new LinkedList<Integer>();
        for ( int i=0; i<5; i++ ) {
            l.addFirst( i );
        }
        System.out.println( "The size of l is " + l.size() );
    }
}
```

(a) The size of l is 0

(b) The size of l is 1

(c) The size of l is 4

(d) The size of l is 6

(e) Infinite recusion causes a stack overflow

Question 4: (20 marks)

Write the Java implementation of the classes **Person**, **Customer**, and **PreferredCustomer** following all the instructions.

A. The class **Person** has fields for holding a person's name, address and telephone number. Make sure to include at least one constructor, as well as appropriate getter and setter methods.

```
// MODEL

public class Person {

    private String name;
    private String address;
    private String phone;

    public Person( String name, String address, String phone ) {
        this.name = name;
        this.address = address;
        this.phone = phone;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public String getPhone() {
        return phone;
    }

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

    public void setAddress( String address ) {
        this.address = address;
    }

    public void setPhone( String phone ) {
        this.phone = phone;
    }

}
```

B. A **Customer** is a **Person** with a customer number, a tally of the purchases for this customer, as well a field indicating if the customer has agreed to be on the mailing list. Write at least one constructor, as well as the appropriate accessor methods.

```
// MODEL

public class Customer extends Person {

    private int customerNumber;
    private static int nextCustomerNumber = 1;

    private double purchases;

    private boolean isOnMailingList;

    public Customer( String name, String address, String phone, boolean isOnMailingList )
        super( name, address, phone );
        this.isOnMailingList = isOnMailingList;
        customerNumber = nextCustomerNumber;
        nextCustomerNumber++;
    }

    public boolean isOnMailingList() {
        return isOnMailingList;
    }

    public void setIsOnMailingList( boolean isOnMailingList ) {
        this.isOnMailingList = isOnMailingList;
    }

    public void updatePurchases( double amount ) {
        purchases = purchases + amount;
    }

    public double getPurchases() {
        return purchases;
    }

}
```

C. A **PreferredCustomer** is a **Customer** that earns on discounts based on the tally of his/her purchases. Specifically, when a **PreferredCustomer** has spent $ 500, he or she gets a 5% discount on future purchases, when a **PreferredCustomer** has spent $ 1,000, he or she gets a 7.5% discount on future purchases, finally, when a **PreferredCustomer** has spent $ 2,000, he or she gets a 10% discount on future purchases. A **PreferredCustomer** has a method **double getDiscountLevel()** that returns a discount percentage based on the tally of the purchases of this customer. Make sure to include at least one constructor.

```
// MODEL

public class PreferredCustomer extends Customer {

    public PreferredCustomer( String name, String address, String phone, boolean isOML )
        super( name, address, phone, isOML );
    }

    public double getDiscountLevel() {

        double discountLevel = 0.0;

        if ( getPurchases() > 2000.0 ) {
            discountLevel = 0.1;
        } else if ( getPurchases() > 1000.0 ) {
            discountLevel = 0.075;
        } else if ( getPurchases() > 500.0 ) {
            discountLevel = 0.05;
        } else {
            discountLevel = 0.0;
        }

        return discountLevel;
    }

}
```

Question 5: (8 marks)

Implement the class method **public static <E> void swap(Stack<E> xs, Stack<E> ys)**. The method exchanges the content of two stacks, **xs** and **ys**.

- The method must work for any valid implementation of the interface **Stack**;
- You can assume the existence of the classes **DynamicArrayStack** and **LinkedStack**.

```
Stack<String> a, b;

a = new LinkedStack<String>();
a.push( "alpha" ); a.push( "beta" ); a.push( "gamma" );

b = new DynamicArrayStack<String>();
b.push( "blue" ); b.push( "green" ); b.push( "yellow" ); b.push( "black" );

System.out.println( a );
System.out.println( b );
swap( a, b );
System.out.println( a );
System.out.println( b );
```

In particular, the above statements should print the following.

```
[gamma,beta,alpha]
[black,yellow,green,blue]
[black,yellow,green,blue]
[gamma,beta,alpha]
```

```
public static <E> void swap( Stack<E> xs, Stack<E> ys ) {

    // MODEL
    Stack<E> xsReverse, ysReverse;

    xsReverse = new LinkedStack<E>();

    while ( ! xs.isEmpty() ) {
        xsReverse.push( xs.pop() );
    }

    ysReverse = new LinkedStack<E>();

    while ( ! ys.isEmpty() ) {
        ysReverse.push( ys.pop() );
    }

    while ( ! xsReverse.isEmpty() ) {
        ys.push( xsReverse.pop() );
    }

    while ( ! ysReverse.isEmpty() ) {
        xs.push( ysReverse.pop() );
    }
}
```

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 disignating 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 ) { // MODEL

        if ( from < 0 || from > to || to >= size ) {
            throw new IllegalArgumentException( "from:"+from+",to:"+to );
        }

        LinkedList<E> result;
        result = new LinkedList<E>();

        Node<E> first = head.next;
        for ( int i=0; i<from; i++ ) {
            first = first.next;
        }

        Node<E> last = first;
        for ( int i=0; i<(to-from); i++ ) {
            last = last.next;
        }

        Node<E> other = result.head; // dummy node

        other.next = first;
        other.previous = last;

        first.previous.next = last.next;
        last.next.previous = first.previous;

        first.previous = other;
        last.next = other;

        result.size = to - from + 1;
        size = size - result.size;

        return result;

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

Question 7: (12 marks)

Complete the implementation of the class **CircularStack**. Read all the directives. In particular, notice that this stack implementation uses a circular array.

```
public interface Stack<E> {

    // Adds an element onto the top of this stack
    public abstract void push( E element );

    // Removes and returns the top element of the stack
    public abstract E pop() throws java.util.EmptyStackException;

    // Returns true if and only if this stack is empty
    public abstract boolean isEmpty();
}
```

- This implementation uses a **fixed-size circular array**;
- When the stack is full, the method **push** replaces the bottom element with the new element to be added;
- Therefore, the method **push** can always add new elements to the stack, even when the stack is full, but the oldest elements, those at the bottom of the stack, are lost;
- If n is the capacity of the stack, then this stack memorises a maximum of n elements, the last ones added to the stack;
- The constructor has a single parameter, which defines the capacity of the stack;
- The **null** value is a valid value for this stack.

```
public class CircularStack<E> implements Stack<E> {

    private E[] elems;
    private int top, size;

    public CircularStack( int capacity ) {


        elems = (E[]) new Object[ capacity ] ;

        top = -1;
        size = 0;

    }

    public boolean isEmpty() {


        return size==0 ;

    }

    // continues on the next page
```

```
    // Question 7 continues

    public void push( E elem ) {

        top = (top + 1) % elems.length ;

        elems[ top ] = elem;

        if ( size < elems.length ) {

            size++;

        }
    }

    public E pop() {

        if ( isEmpty() ) {

            throw new java.util.EmptyStackException() ;
        }

        E saved = elems[ top ];

        elems[ top ] = null ;

        if ( size==1 ) {

            top = -1 ;

            size = 0;

        } else {
            size--;
            top--;

            if ( top == -1 ) {

                top = elems.length - 1 ;

            }
        }

        return saved;
    }
} // End of CircularStack
```

Question 8: (7 marks)

Complete the implementation of the instance method **isValid()** for the class **BinarySearchTree** below. This **recursive** method returns **true** if and only all the nodes of the tree are locally valid, and **false** otherwise.

```
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;

    public boolean isValid() {


        return isValid(  root  );

    }

    private boolean isValid( Node<E> current ) {
        boolean isValid = true;

        if ( current != null ) {

            if ( current.left != null ) {


                isValid = current.left.value.compareTo(current.value)< 0  && isValid(current.left) ;

            }

            if (  isValid && current.right != null  ) {


                isValid = current.right.value.compareTo( current.value )>0 && isValid( current.right) ;

            }

        }

        return isValid;
    }
}
```

(blank space)