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: Solutions

Instructor: Marcel Turcotte

April 2010, duration: 3 hours

Identification

Student name: _____

Student number: _____ Signature: _____

Instructions

1. This is a closed book examination;
2. No calculators or other aids are permitted;
3. Write comments and assumptions to get partial marks;
4. Beware, poor hand writing can affect grades;
5. Do not remove the staple holding the examination pages together;
6. 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	15	
4	15	
5	12	
6	13	
7	25	
Total	**100**	

Question 1: (10 marks)

For each statement, you must circle the correct answer, either true or false.

A. In Java, `total`, `ToTal` and `TOTAL` are all different identifiers.
Answer: true false

B. A single class may implement multiple interfaces.
Answer: true false

C. In Java, a subclass can only extend one parent class.
Answer: true false

D. A child class is allowed to define a method with the same name and parameter list as a method in the parent class.
Answer: true false

E. An abstract class must contain abstract methods.
Answer: true false

F. Unchecked exceptions must be caught or propogated, or a program will not compile.
Answer: true false

G. A breadth-first search uses a stack as a supporting data structure.
Answer: true false

H. It is possible to implement a stack and a queue in such a way that all the operations take a constant amount of time.
Answer: true false

I. A program with infinite recursion will act similarly to a program with an infinite loop.
Answer: true false

J. In a tree, a node that does not have any children is called a leaf.
Answer: true false

Question 2: (10 marks)

A. The Java Standard Edition Development Kit includes a program that that can package several Java files into a single file. What is the name of this program?

(a) zip

(b) tar

(c) compress

(d) jar

(e) WinZip

Answer: d

B. Which of the following statements and declarations is **not valid**?

(a) `next() = next + 1;`

(b) `a = b / c % d;`

(c) `int money, dollars = 0, cents = 0;`

(d) `float pi = 3.1415F;`

(e) all of the above

Answer: a

C. The following Java program:

(a) will result in a compile-time error

(b) will result in a run-time error

(c) displays **true**

(d) displays **false**

(e) displays **"5 == 5"**

Answer: d

```
public class Test {
    public static void incr( int value ) {
        value = value + 1;
    }
    public static void incr( Integer value ) {
        value = value + 1;
    }
    public static void main( String[] args ) {
        int i1;
        Integer i2;
        i1 = 5;
        incr( i1 );
        i2 = 4;
        incr( i2 );
        System.out.println( i1 == i2 );
    }
}
```

Question 2: (continued)

D. A class declared as final _____.

(a) cannot be changed
(b) cannot have subclasses
(c) cannot have superclasses
(d) has several abstract methods
(e) cannot be used in a program

Answer: b

E. To invoke a parents constructor in a subclass, we use the _____ method.

(a) abstract
(b) construct
(c) parent
(d) super
(e) extends

Answer: d

F. A _____ variable is shared among all instances of a class.

(a) static
(b) final
(c) public
(d) private
(e) none of the above

Answer: a

G. Which of the following refers to a set of data and the operations that are allowed on that data?

(a) abstract data type
(b) generic
(c) collection
(d) bag
(e) none of the above

Answer: a

Question 2: (continued)

H. Which of the following is an advantage of an array-based implementation over a linked-based implementation?

(a) the structure grows and shrinks dynamically in the array-based implementation

(b) the underlying structure is a fixed size array-based implementation

(c) the elements can be accessed directly in an array-based implementation

(d) all of the above

(e) neither a, b, nor c

Answer: c

I. In a binary search tree, the elements in the right subtree of the root are _____ the root element.

(a) greater than

(b) less than

(c) greater than or equal to

(d) less than or equal to

(e) equal to

Answer: a

J. When removing an element from a binary search tree that is a leaf, _____ will ensure that the resulting tree is still a binary search tree.

(a) replacing it with its only child

(b) replacing it with its inorder successor

(c) simply deleting it

(d) all of the above

(e) neither a, b, nor c

Answer: c

Question 3: (15 marks)

A. Write out the order of the elements that are contained in a stack after the following operations are performed. Make sure to indicate which element is the top element.

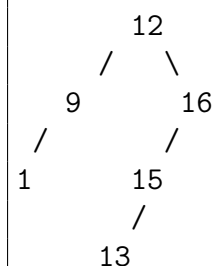
```
s.push( new Integer( 8 ) );
s.push( new Integer( 6 ) );
Integer num = s.pop();
s.push( new Integer( 3 ) );
s.push( new Integer( 4 ) );
s.push( new Integer( 15 ) );
s.push( new Integer( 12 ) );
s.pop();
s.pop();
s.pop();
s.push( new Integer( 19 ) );
```

Answer:

The resulting stack is: 19 (top), 3, 8 (bottom).

B. Draw the binary search tree that results from inserting the following elements, in that order, when using the method **add** presented in class: 12, 16, 9, 1, 15, 13.

Answer:



C. Give the outcome of the program.

```
public class Test {

    public static int bar( int i ) {
        if ( i == 2 ) {
            throw new Exception( "oups!" );
        }
        return i;
    }

    public static int foo() {
        int result;
        result = bar( 1 );
        try {
            result = bar( 2 );
        } catch ( Exception e ) {
            result = bar( 3 );
        }
        return result;
    }

    public static void main( String[] args ) {
        System.out.println( foo() );
    }

}
```

Answer:

```
TestException.java:10: unreported exception java.lang.Exception;
            must be caught or declared to be thrown
            throw new Exception( "oups!" );
            ^
1 error
```

D. The class **Sequence** implements a linked list with additional methods to write efficient recursive list processing methods outside of the implementation of the list. Here are the characteristics of the class **Sequence**.

- **boolean isEmpty()**; returns **true** if and only if **this** list is empty;
- **Sequence<E> split()**; returns the tail of **this** sequence, **this** sequence now contains a single element. It throws **IllegalStateException** if the **Sequence** was empty when the call was made;
- **void join(Sequence<E> other)**; appends **other** at the end of **this** sequence, **other** is now empty;
- **deleteFirst()** removes the first element of this **Sequence** .

```
public static <E> int april2010( Sequence<E> l ) {

    int result = 0;

    if ( ! l.isEmpty() ) {

        Sequence<E> tail = l.split();

        result = 1 + april2010( tail );

        if ( result % 2 == 0 ) {
            l.deleteFirst();
            result = 0;
        }

        l.join( tail );
    }

    return result;
}
```

Let **s** designate a **Sequence** containing the following elements, in that order, {0,1,2,3,4}, what will the content of **s** after the call **april2010(s)**?

Answer: {0, 2, 4}

Question 4: (15 marks)

In the class **SinglyLinkedList** below, write a **recursive** (instance) method that returns a list of all the positions where a given object is found. The method must be implemented following the technique presented in class for implementing recursive methods inside the class. Briefly, there will be a public method that initiates the first call to the recursive private method.

Let **l** designate a list containing the elements {A, B, A, A, C, D, A}, then a call **l.indexOfAll(A)** would return the following list of positions {0, 2, 3, 6}.

```
public class SinglyLinkedList<E> {

    private static class Node<F> {

        private F value;
        private Node<F> next;

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

    // Instance variable
    private Node<E> first;

    // Instance methods
    public void addFirst( E item ) { ... }
    public void addLast( E item ) { ... }
```

```
 // Complete the implementation of the method below

 public                            indexOfAll( E elem ) {

 } // End of indexOfAll

 // MODEL:

public LinkedList<Integer> indexOfAll( E elem ) {
    if ( elem == null ) {
        throw new NullPointerException( "Illegal argument" );
    }
    return indexOfAll( first, 0, elem );
}
```

```
 // Complete the implementation of the method below

 private                      indexOfAllRec(                              ) {

 } // End of indexOfAllRec

 // MODEL:

private LinkedList<Integer> indexOfAll( Node<E> current, int pos, E elem ) {

    LinkedList<Integer> result;

    if ( current == null ) {

        result = new LinkedList<Integer>();

    } else {

        result = indexOfAll( current.next, pos+1, elem );

        if ( elem.equals( current.value ) ) {
            result.addFirst( new Integer( pos ) );
        }

    }

    return result;
}
```

} // End of SinglyLinkedList

Question 5: (12 marks)

In the partial implementation of the class **CircularQueue** below, complete the implementation of the instance method **void rotate(int n)**.

- The method **rotate(int n)** removes **n** elements from the front of the queue and adds them to the rear of the queue. The elements are added at the rear of the queue in the same order as they are removed from the front of the queue. For example, given a queue **q** containing the elements "A, B, C, D, E", where the element **A** is at the front of the queue, following the method call **q.rotate(2)**, the content of the queue will be "C, D, E, A, B";
- **CircularQueue** uses the circular array technique seen in class, i.e. the front and rear of the queue are allowed to wrap around the boundaries of the array;
- The instance variable **front** designates the cell of the array that contains the front element of the queue or -1 if the queue is empty;
- The instance variable **rear** designates the cell of the array that contains the rear element of the queue or -1 if the queue is empty;
- The array **elems** is used to store the elements of this queue;
- If the value of the parameter is negative, the method must throw the exception **IllegalArgumentException**;
- For your implementation of the method **rotate**, you are not allowed to use any instance method, in particular, you cannot use the methods **enqueue** and **dequeue**. Your solution must make explicit use of the variables **front** and **rear**.

Complete the implementation of the methods **rotate(int n)** on the next page.

```
public class CircularQueue<E> implements Queue<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 rotate( int n ) {

      // MODEL

      if ( n < 0 ) {
          throw new IllegalArgumentException( "negative value: " + 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--;
          }

      }

    }

} // End of CircularQueue
```

Question 6: (13 marks)

Implement the method **rotate**, as described for Question 5, but this time for the implementation of a queue based on linked elements.

- **LinkedQueue** uses linked elements to store the values of this queue;
- The instance variable **front** designates the front element of the queue or has value **null** if the queue is empty;
- The instance variable **rear** designates the rear element of the queue or is **null** if the queue is empty;
- If the value of the parameter is negative, the method must throw the exception **IllegalArgumentException**;
- For your implementation of the method **rotate**, you are not allowed to use any instance method, in particular, you cannot use the methods **enqueue** and **dequeue**. Your solution must explicitly change the structure of the list by changing the links of the list (the references **next**).

Complete the implementation of the methods **rotate(int n)** on the next page.

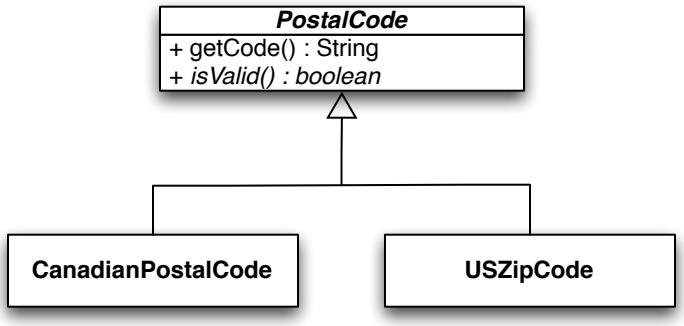
```
public class LinkedQueue<E> implements Queue<E> {

    private static class Elem<F> {

        private F value;
        private Elem<F> next;

        private Elem( F value, Elem<F> next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Elem<E> front;
    private Elem<E> rear;

    public LinkedQueue() {
        front = rear = null;
    }
```

```
    public void rotate( int n ) {

       // MODEL

       if ( n < 0 ) {
           throw new IllegalArgumentException( "negative value: " + n );
       }

       if ( front != null && front.next != null ) {

           while ( n > 0 ) {

               Elem<E> current = front;
               front = front.next;

               rear.next = current;
               rear = rear.next;
               rear.next = null;

               n--;
           }

       }
```

```
    }

} // End of LinkedQueue
```

Question 7: (25 marks)

The UML diagram below shows a hierarchy of classes to represent postal codes of different countries.



Knowing that:

- All postal codes have a method **getCode** that returns the code (of type **String**) represented by this instance;
- All postal codes have a method **isValid** that returns **true** if this code is valid and **false** otherwise;
- A Canadian postal code is valid if positions 0, 2 and 5 are letters, positions 1, 4 and 6 are digits, and the position 3 is a blank character;
- A valid US zip code consists of two letters, followed by a blank character, followed by 5 digits.

Write an implementation for the classes **PostalCode**, **CanadianPostalCode** and **USZipCode**. Make sure to include the instance variables and necessary constructors. Appendix A lists some of the methods of the classes **String** and **Character**.

```
// Implement the class PostalCode here

// MODEL

public abstract class PostalCode {

    private String code;

    public PostalCode( String code ) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }

    public abstract boolean isValid();
}
```

Question 7: (continued)

```
// Implement the class CanadianPostalCode here

// MODEL

public class CanadianPostalCode extends PostalCode {

    public CanadianPostalCode( String code ) {
        super( code );
    }

    public boolean isValid() {
        String myCode = getCode();
        boolean valid = true;

        if (myCode == null || myCode.length() != 7)
            valid = false;

        if ( ! Character.isLetter( myCode.charAt( 0 ) ) )
            valid = false;
        if ( ! Character.isLetter( myCode.charAt( 2 ) ) )
            valid = false;
        if ( ! Character.isLetter( myCode.charAt( 5 ) ) )
            valid = false;

        if ( ! Character.isDigit( myCode.charAt( 1 ) ) )
            valid = false;
        if ( ! Character.isDigit (myCode.charAt( 4 ) ) )
            valid = false;
        if ( ! Character.isDigit( myCode.charAt( 6 ) ) )
            valid = false;

        if (! Character.isWhitespace( myCode.charAt( 3 ) ) )
            valid = false;

       return valid;
    }
}
```

Question 7: (continued)

```
// Implement the class USZipCode here

// MODEL

public class USZipCode extends PostalCode {
    public USZipCode( String code ) {
        super( code );
    }
    public boolean isValid() {
        String myCode = getCode();
        boolean valid = true;
        if (myCode == null || myCode.length() != 8)
            valid = false;
        if ( ! Character.isLetter( myCode.charAt( 0 ) ) )
            valid = false;
        if ( ! Character.isLetter( myCode.charAt( 1 ) ) )
            valid = false;
        if ( ! Character.isWhitespace( myCode.charAt( 2 ) ) )
            valid = false;
        if ( ! Character.isDigit( myCode.charAt( 3 ) ) )
            valid = false;
        if ( ! Character.isDigit( myCode.charAt( 4 ) ) )
            valid = false;
        if ( ! Character.isDigit( myCode.charAt( 5 ) ) )
            valid = false;
        if ( ! Character.isDigit( myCode.charAt( 6 ) ) )
            valid = false;
        if ( ! Character.isDigit( myCode.charAt( 7 ) ) )
            valid = false;
        return valid;
    }
}
```

A Appendix

The class **String** includes the following methods.

char charAt(int pos): Returns the character at the specified index.

int length(): Returns the length of this string.

The class **Character** includes the following methods.

static boolean isDigit(char ch): Determines if the specified character is a digit.

static boolean isLetter(char ch): Determines if the specified character is a letter.

static boolean isWhitespace(char ch): Determines if the specified character is white space according to Java.