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 Computer Science II (ITI 1121)
Midterm Examination

Instructor: Marcel Turcotte

February 2008, duration: 2 hours

Identification

Student name: ______________________________

Student number: ______________ Signature: ______________________

Instructions

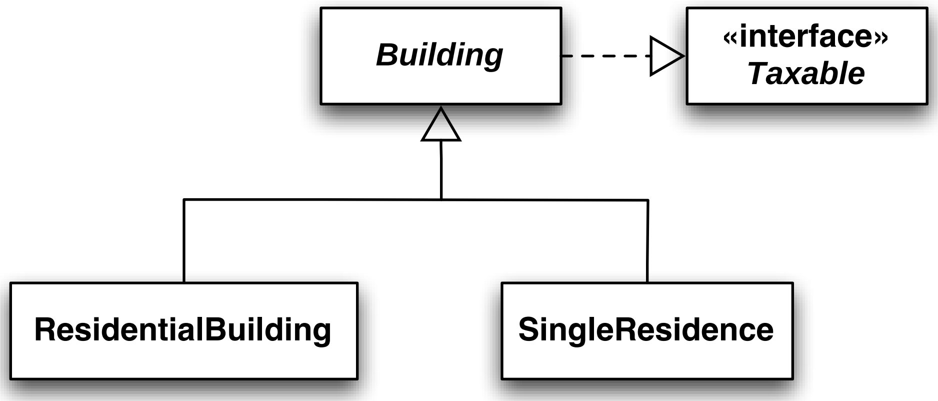
1. This is a closed book examination;
2. No calculators or other aids are permitted;
3. Write comments and assumptions to get partial marks;
4. Beware, poor hand writing can affect grades;
5. Do not remove the staple holding the examination pages together;
6. Write your answers in the space provided.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

Marking scheme

Question	Maximum	Result
1	35	
2	10	
3	10	
4	10	
5	10	
Total	**75**	

Question 1: (35 marks)

For this question, there is an interface, named **Taxable**, as well as three classes, **Building**, **ResidentialBuilding** and **SingleResidence**.



Knowing that:

1. all buildings have an address (a character string);
2. all buildings have a valuation (a double);
3. the residential buildings have a number of residential units (an integer);
4. the single residences have a swimming pool or not (a boolean);
5. all the buildings are **Taxable**, where **Taxable** is an interface that declares a method named **getTaxAmount**;
6. the tax amount for a residential building is the sum of two values. First, there is a general tax, which is a fixed rate (0.75%) times the valuation of the property. Second, there is a garbage collection tax, which is a fixed amount ($149.77) times the number of residential units;
7. the tax amount for a single residence is the sum of the following values: first, there is a general tax, which is a fixed rate (0.8%) times the valuation of the property. Second, there is a garbage collection tax, which is a fixed amount ($159.77). Thirdly, there is a tax of $89.01 for a swimming pool.

In Java, implement the above three classes, as well as the interface. Include the necessary constructors. Declare constants whenever necessary. For all the attributes, create the necessary access methods. However, the objects must be immutable (a concept seen in lecture 10).

A. In Java, implement the above three classes, as well as the interface. (27 marks)

B. Create a static method, named **tallyTaxes**. Its first parameter is a reference to an array of **Taxable** objects. The second parameter is the number of occupied cells in the array. You can assume that i) the parameter number of occupied cells is less than or equals to the number of cells in this array, ii) the elements are stored in the leftmost contiguous cells. The method returns the sum of the tax amount of all the objects stored in the array. (5 marks)

C. Declare and create an array, named **city**, to hold 23,000 buildings. (3 marks)

Question 2: Using a stack (10 marks)

The class **Calculator** (next page) implements a postfix evaluator similar to the one seen in class. Show the information that will be printed onto the screen when the following two statements are executed:

```
Calculator c = new Calculator();
c.execute( "6 4 5 print * print 2  3 print ^ print ~ print + print" );
```

Notes:

- For this question, there is an implementation of the interface **Stack** named **LinkedStack**;
- The implementation of the classes **Token** and **Reader** can be found in Appendix B and C.

Using a stack (continued)

```
public class Calculator {
    private Stack<Token> operands = new LinkedStack<Token>();
    public void execute( String program ) {
        Reader r = new Reader( program );
        while ( r.hasMoreTokens() ) {
            Token t = r.nextToken();
            if ( ! t.isSymbol() ) {
                operands.push( t );
            } else if ( t.sValue().equals( "+" ) ) {
                Token b = operands.pop();
                Token a = operands.pop();
                Token res = new Token( a.iValue() + b.iValue() );
                operands.push( res );
            } else if ( t.sValue().equals( "-" ) ) {
                Token b = operands.pop();
                Token a = operands.pop();
                Token res = new Token( a.iValue() - b.iValue() );
                operands.push( res );
            } else if ( t.sValue().equals( "*" ) ) {
                Token b = operands.pop();
                Token a = operands.pop();
                Token res = new Token( a.iValue() * b.iValue() );
                operands.push( res );
            } else if ( t.sValue().equals( "~" ) ) {
                Token o = operands.pop();
                Token res = new Token( - o.iValue() );
                operands.push( res );
            } else if ( t.sValue().equals( "^" ) ) {
                Token b = operands.pop();
                Token a = operands.pop();
                Token res = new Token( (int) Math.pow( a.iValue(), b.iValue() ) );
                operands.push( res );
            } else if ( t.sValue().equals( "print" ) ) {
                System.out.println( "-top-" );
                Stack<Token> tmpStack = new LinkedStack<Token>();
                while ( ! operands.isEmpty() ) {
                    t = operands.pop();
                    System.out.println( t );
                    tmpStack.push( t );
                }
                while (! tmpStack.isEmpty()) {
                    operands.push( tmpStack.pop() );
                }
                System.out.println( "-bottom-" ); System.out.println();
            }
        }
    }
}
```

Question 3: (10 marks)

The class **DynamicArrayStack** below uses the technique seen in class to increase its physical size according to the needs of the application.

- **DynamicArrayStack** uses an array to store the elements of this stack;
- The interface **Stack** and its implementation, **DynamicArrayStack**, have a formal type parameter (in other words, the implementation uses the concept of generics types, introduced in Java 1.5);
- The initial capacity of this array is given by the constant **INITIAL_CAPACITY**;
- The physical size of the array is increased by a **FACTOR** (an other constant) whenever the method **void push(E elem)** is called and the array is full;
- The instance variable **top** designates the top element (i.e. the cell where the last element was inserted, or -1 if the stack is empty);
- The description of the methods can be found in the Appendix A.

A. Complete the implementation of the method **compact**. The method decreases the physical size of the array so that there are no unoccupied cells. However, the physical size of the array must never be less than **INITIAL_CAPACITY**.

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

    // Constants

    public static final int INITIAL_CAPACITY = 10;
    public static final double FACTOR = 1.5;

    // Instance variables

    private E[] elems;            // Stores the elements of this stack
    private int top = -1;         // Designates the top element

    public DynamicArrayStack() {
        elems = (E[]) new Object[ INITIAL_CAPACITY ];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public E peek() {
        return elems[ top ];
    }

    // Continues on the next page...
    public void push( E element ) {
        if ( ( top + 1 ) == elems.length ) {
            increaseSize();
        }
        top++;
        elems[ top ] = element;
    }

    private void increaseSize() {
        // The implementation of the method increaseSize would be here.
        // It has been removed to avoid giving away too much information.
    }

    public E pop() {
        E saved;
        saved = elems[ top ];
        elems[ top ] = null;
        top--;
        return saved;
    }

    public void compact() { // Complete the implementation of this method




    } // End of compact
} // End of DynamicArrayStack
```

Question 4: (10 marks)

Complete the implementation of the instance methods **size()** and **swap()** within the class **Linked-Stack** below.

A. The method **size()** returns the number of elements that are currently stored into this stack;

B. The method **swap** exchanges the first two elements (**not the values**); the first element becomes the second and the second element becomes the first. The method returns **false** if there are less than 2 elements in the list. You cannot use the methods **push** and **pop**, instead the links of the structure (references) must be transformed.

```
public class LinkedStack<T> implements Stack<T> {

    private static class Elem<E> { // Implements the nodes of the list
        private E info;
        private Elem<E> next;
        private Elem( E info, Elem<E> next) {
            this.info = info;
            this.next = next;
        }
    }

    private Elem<T> top; // Instance variable, designates the top element

    public int size() {






















    }

    // Continues on the next page...
    public boolean swap() {
















    } // End of swap

    // The other instance methods, such as push and pop would go here;
    // but they cannot be used to answer this question.

} // End of LinkedStack
```

Question 5: (10 marks)

A. Convert the following infix expression to postfix notation (Reverse Polish Notation — RPN).

$$((5 + 3) / 2) \times (4 - 1)$$

B. This question is about the interface **Valuable**, the class **Number** and the class **Integer**, as defined on page 14.

Which of the following 4 outcomes characterizes best each of the 8 test cases found on the next page.

(1) Produces a compile time error?

(2) Produces a run time error?

(3) Prints "if";

(4) Prints "else";

For each test case, circle one answer.

(a) **1 2 3 4**

(b) **1 2 3 4**

(c) **1 2 3 4**

(d) **1 2 3 4**

(e) **1 2 3 4**

(f) **1 2 3 4**

(g) **1 2 3 4**

(h) **1 2 3 4**

(a)
```
Number n;

n = new Integer( 3 );

if ( n.getType().equals( "Number" ) ) {
    System.out.println( "if" );
} else {
    System.out.println( "else" );
}
```

(b)
```
Valuable v;

v = new Valuable( 3 );

if ( v.getValue() == 3 ) {
    System.out.println( "if" );
} else {
    System.out.println( "else" );
}
```

(c)
```
Number n;

Integer i;

n = new Integer( 3 );

i = n;

if ( n.getValue() == 3 ) {
     System.out.println( "if" );
} else {
     System.out.println( "else" );
}
```

(d)
```
Integer i, j, k;

i = new Integer( 1 );
j = new Integer( 2 );

k = i.plus( j );

if ( k.getValue() == 3 ) {
    System.out.println( "if" );
} else {
    System.out.println( "else" );
}
```

(e)
```
Object o = new String( "3" );

Integer i;

i = (Integer) o;

if ( i.getValue() == 3 ) {
    System.out.println( "if" );
} else {
    System.out.println( "else" );
}
```

(f)
```
Number n;

n = new Integer( 3 );

if ( n.toString().equals( "Integer: 3" ) ) {
    System.out.println( "if" );
} else {
    System.out.println( "else" );
}
```

(g)
```
Integer i, j;

i = new Integer( 3 );
j = new Integer( 3 );

if ( i == j ) {
    System.out.println( "if" );
} else {
    System.out.println( "else" );
}
```

(h)
```
Integer i, j;

i = new Integer( 3 );
j = new Integer( 3 );

if ( i.equals( j ) ) {
    System.out.println( "if" );
} else {
    System.out.println( "else" );
}
```

Declarations for Question 5: on page 12.

```
public interface Valuable {
    public int getValue();
}

public abstract class Number implements Valuable {

    public String toString() {
        return "Number: " + getValue();
    }

    public String getType() {
        return "Number";
    }
}

public class Integer extends Number {

    private int value;

    public Integer( int value ) {
        this.value = value;
    }

    public int getValue () {
        return value;
    }

    public Integer plus( Integer other ) {
        int value = 0;
        return new Integer( value + other.value );
    }

    public String toString() {
        return "Integer: " + getValue();
    }

    public String getType() {
        return "Integer";
    }
}
```

A Stack

```
public interface Stack<T> {

    /**
     * Tests if this Stack is empty.
     *
     * @return true if this Stack is empty; and false otherwise.
     */

    public abstract boolean isEmpty();

    /**
     * Removes and returns the top element.
     *
     * @return The top element.
     */

    public abstract T pop();

    /**
     * Puts an element onto the top of this stack.
     *
     * @param element the element be put onto the top of this stack.
     */

    public abstract void push( T element );

}
```

B Token

```
public class Token {
    private static final int INTEGER = 1;
    private static final int SYMBOL = 2;

    private int iValue;
    private String sValue;

    private int type;

    public Token( int iValue ) {
        this.iValue = iValue;
        type = INTEGER;
    }
    public Token( String sValue ) {
        this.sValue = sValue;
        type = SYMBOL;
    }
    public int iValue() {
        // pre-condition: this Token represents an integer value
        return iValue;
    }
    public String sValue() {
        // pre-condition: this Token represents a symbol
        return sValue;
    }
    public boolean isInteger() {
        return type == INTEGER;
    }
    public boolean isSymbol() {
        return type == SYMBOL;
    }
    public String toString() {
        switch ( type ) {
        case INTEGER:
            return "INTEGER: " + iValue;
        case SYMBOL:
            return "SYMBOL: " + sValue;
        default:
            return "INVALID";
        }
    }
}
```

C Reader

```
import java.util.StringTokenizer;

public class Reader {

    private StringTokenizer st;

    public Reader( String s ) {
        st = new StringTokenizer( s );
    }
    public boolean hasMoreTokens() {
        return st.hasMoreTokens();
    }

    public Token nextToken() {

        String t = st.nextToken();

        try {

            return new Token( Integer.parseInt( t ) );

        } catch ( NumberFormatException e ) {

            return new Token( t );

        }
    }
}
```

(blank space)