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)
MIDTERM EXAMINATION

Instructors: Guy-Vincent Jourdan and Marcel Turcotte

March 2018, duration: 2 hours

Identification

Last name: _____ First name: _____

Student #: _____ Seat #: _____ Signature: _____ Section: A or B or C

Instructions

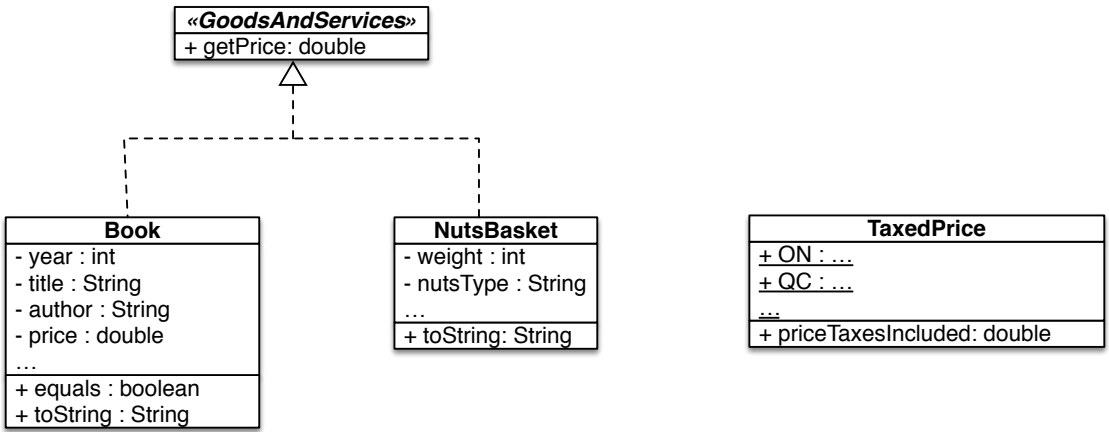
1. This is a closed book examination.
2. No calculators, electronic devices or other aids are permitted.
   (a) Any electronic device or tool must be shut off, stored and out of reach.
   (b) Anyone who fails to comply with these regulations may be charged with academic fraud.
3. Write your answers in the space provided.
   (a) Use the back of pages if necessary.
   (b) You may not hand in additional pages.
4. Do not remove pages or the staple holding the examination pages together.
5. Write comments and assumptions to get partial marks.
6. Beware, poor hand-writing can affect grades.
7. Wait for the start of the examination.

Marking scheme

Question	Maximum	Result
1	35	
2	15	
3	15	
4	5	
Total	**70**	

Question 1 (35 marks)

This question is about object-oriented programming in Java. Specifically, this is a small part of a software system that you are developing for a supermarket chain. We are focussing on 2 types of goods, "books" and "baskets of nuts", and the part of the software system that computes the price of good and services after taxes for all items sold by the chain, in every province.



In our software system, every object representing a good or a service, which includes objects of the classes **Book** and **NutsBasket**, **can be seen** as a **GoodsAndServices**.

- A **GoodsAndServices** has a method called **getPrice**, which returns a **double** value that is the price of that good or service **before taxes**.
- **Book** has 4 instance variables: the price before taxes of the book, of type **double**, the year of publication, of type **int**, and then two instances of **String** for the title and the author of the book. The **constructor** of the class receives these 4 elements of information as parameters.

  You need to also include three methods in **Book**: first, the instance method **toString**, which should behave as shown in the example below. Second, the instance method **equals**. Two books should be considered the same if they have the same title and author, and are published the same year. Finally, for **getPrice**, the price before tax of a book has been received as a parameter to the constructor.
- **NutsBasket** has 2 instance variables, the **nutsType**, of type **String**, and the **weight** in grams, which is an **int**. The **constructor** of the class receives these 2 elements of information as parameters. The class will also need an implementation of the instance method **toString**. In the case of a **NutsBasket**, the price before taxes is $0.1\$$ per gram.

The Java source code below shows the intended use of the classes and their methods.

```
Book b1, b2, b3;

b1 = new Book(1996, "John Smith", "Java for dummies", 85.0);
b2 = new Book(1996, "John Smith", "Java for dummies", 40.0);
b3 = new Book(2010, "Jane Adam", "Python for dummies", 50.0);

System.out.println(b1);
System.out.println(b2);
System.out.println(b3);
System.out.println(b1.equals(b2));
System.out.println(b1.equals(b3));

NutsBasket n1, n2;

n1 = new NutsBasket("Pistachios", 150);
n2 = new NutsBasket("Almonds", 400);

System.out.println(n1);
System.out.println(n2);
```

The execution of the above program should produce the following output on the console.

```
"Java for dummies" by John Smith, sold for 85.0$ plus taxes
"Java for dummies" by John Smith, sold for 40.0$ plus taxes
"Python for dummies" by Jane Adam, sold for 50.0$ plus taxes
true
false
basket of Pistachios costing 15.0$ plus taxes
basket of Almonds costing 40.0$ plus taxes
```

In the boxes next pages, you need to provide all the code necessary for this to work as described.

Lastly, you need to create a class **TaxedPrice**. This class is used to have the price of a good or service with taxes included, in various provinces.

We will only support 2 provinces at the moment: in Ontario, goods and services are taxed at 13%, and in Quebec, goods and services are taxed at 14,975%.

TaxedPrice has a **class** method **priceTaxesIncluded** that accepts two parameters, the good or service and the province. It returns the price of the good or service, including the taxes for that province.

The Java source code below shows the intended use of that class.

```
Book b1;

b1 = new Book(1996,"John Smith","Java for dummies", 85.0);

NutsBasket n1;

n1 = new NutsBasket("Pistachios", 150);

System.out.println("With taxes: " + b1 + " costs "
        +TaxedPrice.priceTaxesIncluded(b1,TaxedPrice.ON) + " in Ontario and "
        +TaxedPrice.priceTaxesIncluded(b1,TaxedPrice.QC) + " in Quebec.");

System.out.println("With taxes: " + n1 + " costs "
        +TaxedPrice.priceTaxesIncluded(n1,TaxedPrice.ON) + " in Ontario and "
        +TaxedPrice.priceTaxesIncluded(n1,TaxedPrice.QC) + " in Quebec.");
```

The execution of the above program should produce the following output on the console.

```
With taxes: "Java for dummies" by John Smith, sold for 85.0$ plus taxes
costs 96.05 in Ontario and 97.72875 in Quebec.

With taxes: basket of Pistachios costing 15.0$ plus taxes costs 16.95 in
Ontario and 17.24625 in Quebec.
```

In the box next page, provide the code for the class **TaxedPrice**.

Question 2 (15 marks)

In assignment 1, we introduced the concept of curve fitting through a linear regression. Smoothing is a similar concept, which finds applications in statistics and image processing. Whereas curve fitting uses an explicit function, smoothing gradually changes the values of a data set to remove noise and approximate some unknown function. For the class **Smoothing** on the next page, implement the class methods **printArray** and **smooth**.

- The method **printArray** receives the reference of an **array of doubles** as a parameter, and prints the values of that array as shown in the example below.
- The method **smooth** also receives the reference of an **array of doubles** as a parameter, and "smoothes" its value, which means that each value will be replaced by the average of the value before, the value itself, and the value after in the array. The first value is replaced by the average of the first value and the second value, and the last value is replaced by the average of the last value and the one before the last. For instance, an array $[2.0, 3.0, 1.0, 5.0]$ will be smoothed into $[2.5, 2.0, 3.0, 3.0]$.

The Java source code below shows the intended use of the class.

```
double[] test;

test = new double[]{0.0,5.0,0.0,5.0};
Smoothing.testSmooth(test);

test = new double[]{0.0,5.0,5.0,0.0};
Smoothing.testSmooth(test);

test = new double[]{0.0};
Smoothing.testSmooth(test);

test = new double[]{0.0,1.0};
Smoothing.testSmooth(test);

test = new double[]{2.0, 3.0, 1.0, 5.0};
Smoothing.testSmooth(test);

test=null;
Smoothing.testSmooth(test);
```

The execution of the above program should produce the following output on the console.

```
Running testSmooth on: [0.0, 5.0, 0.0, 5.0]
After smoothing: [2.5, 1.6666666666666667, 3.3333333333333335, 2.5]
Running testSmooth on: [0.0, 5.0, 5.0, 0.0]
After smoothing: [2.5, 3.3333333333333335, 3.3333333333333335, 2.5]
Running testSmooth on: [0.0]
After smoothing: [0.0]
Running testSmooth on: [0.0, 1.0]
After smoothing: [0.5, 0.5]
Running testSmooth on: [2.0, 3.0, 1.0, 5.0]
After smoothing: [2.5, 2.0, 3.0, 3.0]
Not a valid value, null, for the method testSmooth.
```

Complete the implementation of the class **Smoothing** on the next page.

```
public class Smoothing {

    private static void testSmooth(double[] a) {
        if (a == null) {
            System.out.println("Not a valid value, null, for the method testSmooth.");
            return;
        }
        System.out.print("Running testSmooth on: ");
        printArray(a);
        smooth(a);
        System.out.print("After smoothing: ");
        printArray(a);
    }

    private static void printArray(double[] a) {









    }
```

```
    private static void smooth(double[] a) {









    }
}
```

Question 3 (15 marks)

For the class **StackInspector** on the next page, complete the implementation of the method **findLast(E elem)**.

- The method **findLast** returns the index of the last occurrence of the specified element in the stack. If there is more than one occurrence, the last occurrence is the one closest to the bottom of the stack. This is also the one with the highest index.
- The method returns -1 if the stack does not contain the element.
- The value of the index increases from top to bottom. The index of the top element is 0.
- The method **findLast** does not change the state of the stack. That is before and after a call to the method, the stack must contain the same elements, in the same order.
- The value **null** is not a valid value for the parameter **elem**.

The example below illustrates the behaviour of the method **findLast**.

```
Stack<String> stack;
stack = new StackImplementation<String>();

stack.push("one");
stack.push("two");
stack.push("one");
stack.push("two");
stack.push("three");

StackInspector<String> inspector;
inspector = new StackInspector<String>(stack);

System.out.println(inspector.findLast("one"));
System.out.println(inspector.findLast("two"));
System.out.println(inspector.findLast("three"));
System.out.println(inspector.findLast("four"));
```

Executing the above Java code produces the following output on the console.

```
4
3
0
-1
```

For this question, there is class called **StackImplementation** that implements the interface **Stack**. An instance of the class **StackImplementation** can store an arbitrarily large number of elements. You do not need to write its implementation, it has been provided to you.

```
public class StackInspector<E> {

    private Stack<E> stack;

    public StackInspector(Stack<E> stack) {
        this.stack = stack;
    }









}
```

```java
/**
 * Stack Abstract Data Type. A Stack is a linear data structure following
 * last-in-first-out protocol, i.e. the last element that has been added
 * onto the Stack, is the first one to be removed.
 */

public interface Stack<E> {

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

    boolean isEmpty();

    /**
     * Returns a reference to the top element; does not change the state
     * of this Stack.
     *
     * @return The top element of this stack without removing it.
     */

    E peek();

    /**
     * Removes and returns the element at the top of this stack.
     *
     * @return The top element of this stack.
     */

    E pop();

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

    void push(E element);

}
```

Question 4 (5 marks)

You will find below an expression written in the **postfix** format. Using the algorithm seen in class, you need to give the corresponding expression written in the **infix** format.

4 3 * 7 6 − * 2 4 + /