# ITI 1121. Introduction to Computer Science II

Laboratory 7

Winter 2014

- Further understanding of interfaces
- Introduction to the applications of stacks
- Introduction to the implementation of stacks
- Review of inheritance concepts

# Part I
# Interface

## 1 Combination

Let's revisit the class **Combination** from laboratory 1. Make all the necessary changes to the class **Combination** (see below) so that it implements the interface `java.lang.Comparable`.

```
public interface Comparable<E> {

     // Compares this object with the specified object for order.
     // Returns a negative integer, zero, or a positive integer
     // as this object is less than, equal to, or greater than the
     // specified object.

    public int compareTo( E other );
}
```

When two **Combination** objects are compared, the first pair of values is compared first, if the value of this object is smaller than that of the other then the method will return -1, if it is greater, then it will return 1, if those values are equal, then the second pair of values is considered, the value -1 or 1 will be returned if the second value of this object is smaller than that of the second, finally, if those values are also equal then the third pair of values is considered.

**Comparable** is part of a package, called **lang**, that is imported by default. Therefore, no need to import **Comparable**.

Write a test program for thouroughly testing your implementation.

**File:**

- Combination.java

# Part II
# Stacks

## Introduction

This laboratory is about an implementation of the interface **Stack** as well as an application

## 2 Modifying the interface Stack: adding a method clear()

Modify the interface **Stack** below adding an abstract method **public void clear()**.

```
public interface Stack<E> {
    public abstract boolean isEmpty();
    public abstract E peek();
    public abstract E pop();
    public abstract void push( E element );
}
```

**File:**

- Stack.java

## 3 Implementing the method clear() in the class ArrayStack

The class **ArrayStack** uses a fixed-size array and implements the interface **Stack**. Now that the interface **Stack** has been modified to have a method **clear()**, the current implementation of the class **ArrayStack** is broken (try compiling it without making any change, what is the error message displayed?).

Since the class **ArrayStack** implements the interface **Stack**, it has to provide an implementation for all the methods that are declared by the interface. Consequently, write an implementation for the method **void clear()**. It removes all of the elements from this **ArrayStack**. The stack will be empty after this call returns.

**Files:**

- ArrayStack.java
- L4Q2.java

## 4 Quiz (1 mark)

- Write out the order of the elements that are contained in a stack (designated by **s**) 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:
```

Submit your answer using Blackboard Learn:

- https://uottawa.blackboard.com/

## 5 Algo1

For this part of the laboratory, you will experiment with two algorithms for validating expressions containing parentheses (round, curly and square parentheses).

The class **Balanced** contains a simple algorithm to validate expressions. Observation: for an expression consisting of well balanced parentheses the the number of opening ones is the same as the closing ones. This suggest an algorithm:

```
public static boolean algo1( String s ) {

    int curly = 0;
    int square = 0;
    int round = 0;

    for ( int i=0; i<s.length(); i++ ) {

        char c = s.charAt( i );

        switch ( c ) {
        case '{':
            curly++;
            break;
        case '}':
            curly--;
            break;
        case '[':
            square++;
            break;
        case ']':
            square--;
            break;
        case '(':
            round++;
            break;
        case ')':
            round--;
```

```
            }
        }
        return curly == 0 && square == 0 && round == 0;
    }
```

Compile this program and experiment. First, experiment with valid expressions, such as these ones "()[]()", "([][()])". Notice that the algorithm also works for expressions that contain operands and operators: "(4 * (7 - 2))".

Next, find invalid expressions that are not handled well by this algorithm, i.e. expressions that are not well balanced and yet the algorithm returns true.

**File:**

- Balanced.java

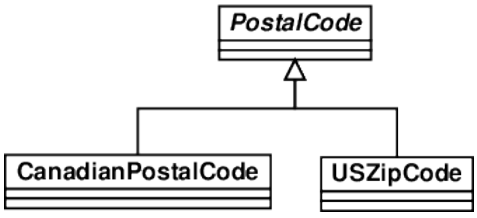
## 6 Algo2

Hopefully, you have been able to identify examples where algo1 fails. A well-balanced expression is an expression such that for each type of parentheses (round, curly and square), the number of opening and closing parentheses are the same. Furthermore, when reading this expression from left to right, every closing parenthesis that you come across is of the same type as the last un-matched opening parenthesis read. Implement a stack-based algorithm to validate an expression. Furthermore, make sure the analysis is carried out in a single pass. Implement this algorithm in the class **Balanced** and call it **algo2**. (My solution is 15 lines long)

Thoroughly test your program using valid and invalid expressions. Is your algorithm handling the following case: "((()))"? How is it handling it?

# Part III
# Review: inheritance

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.

1. 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`.
2. Write a test class for the above classes.
    - Declare an array, called `codes`, to hold exactly 100 postal codes.
    - Create $n = 10$ postal codes, some are USZipCodes, some are CanadianPostalCodes, some must be valid and some must not be valid. Store these codes in the $n$ left most positions of the array;
    - Knowing that exactly `n` postal codes are currently stored in the left most cells of the array, write a `for` loop to count the number of valid codes.

## A Appendix

The class `String` contains 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` contains 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.

**Last Modified: February 12, 2014**