# ITI 1121. Introduction to Computer Science II

Laboratory 7
Summer 2018

- Introduction to the applications of stacks
- Introduction to the implementation of stacks
- Review of inheritance concepts

## Part I

# Stacks

## Introduction

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

## 1 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 );
}
```

## 2 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

## 3 Implement the class LinkedStack<E> that implements the above interface.

- Write a test class Test that inserts 10 Integers from 1to 10 (using s.push( new Integer( i ) )) in two stacks using the ArrayStack and LinkedStack. Then pop all the elements and display the outcome. Repeat the insertion again, then clear the stack.

## Q1) Submit the files Stack, ArrayStack, LinkedStack and Test

# 4 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 number of opening ones is the same as the closing ones. This suggests an algorithm:

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 .

```
public class Balanced {

    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;
    }

    public static void main( String[] args ) {
        for ( int i=0; i<args.length; i++ ) {
            System.out.println( "algo1( \"" + args[ i ] + "\" ) -> " + algo1( args[ i ] ) );
        }
    }
}
```

## 5 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)

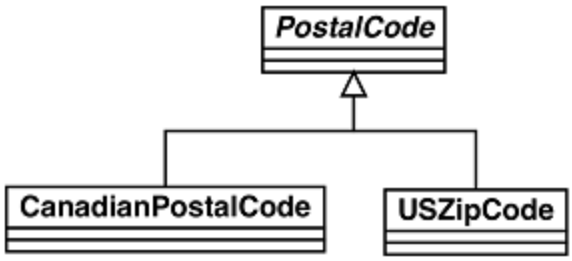
Thoroughly test your program using valid and invalid expressions. Is your algorithm handling the following case: "((()))"? How is it handling it?

### Q2) submit the class Balanced with the new alg2 method.

## Part II (optional)

# 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.

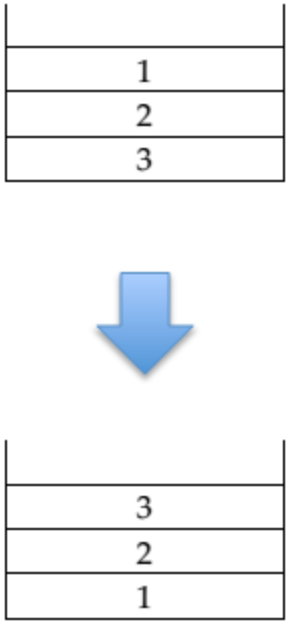
1. Write an implementation for the classes `PostalCode, CanadianPostalCode and USZipCode`. Make sure to include the instance variables and necessary constructors. Appendix section A lists some of the methods of the classes `String` and `Character`.
2. Write a test class for the above classes
   • Declare 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.

## Part III

# Additional question

Please, answer this question after finishing your lab tasks:

Implement a method that reverses a Stack of Integers:



# 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.