COMP 2522 Object oriented programming 1
Assignment 3

Christopher Thompson
cthompson98@bcit.ca

BCIT CST — **Due Sunday March the 8th at or before 23:59:59**

Introduction

For assignment 3, we will examine abstraction and inheritance through the lens of mathematics.

In 1924, a Polish mathematician named Jan Łukasiewicz invented something called Polish notation. It was refined in the early 1960s by Edsger Dijkstra who developed Reverse Polish Notation to take advantage of the Stack data structure.

We've already talked about binary infix operators in COMP 1510 and COMP 2522. Binary infix operators are called binary because they require two operands. They are called infix because the operator is placed **between** the operands:

```
2 + 2 = 4
4 - 2 = 2
5 / 3 = 1 (assuming we are using ints)
```

Reverse Polish notation (RPN) is a mathematical notation where operators **follow** their operands. Instead of using a binary infix operator, RPN uses a binary **postfix** operator:

```
2 2 + = 4
4 2 - = 2
5 3 / = 1
```

Consider our usual notation. When we mix our operations using binary infix operators, we must implement rules of precedence. The use of parentheses can result in dramatically different results:

```
2 - 3 * 4 = -10
(2 - 3) * 4 = -4
```

RPN (postfix) notation removes the need for parentheses! RPN's greatest advantage is clear when we consider expressions that contain more than one operand. If we want an operation to take precedence, we just put the operator immediately to the right of the two operands:

```
2 3 4 * - = -10
2 3 - 4 * = -4
```

RPN doesn't just eliminate parentheses and keystrokes. It's flexible. If we use a Stack by pushing operands onto the Stack until we reached an operator, we can pop operands off the Stack, transform them with the operator, and push the result back onto the Stack as we go, letting us calculate complex partial results without having to save them in multiple locations.

That's exactly what we're going to do. We'll use a bit of inheritance and abstraction to create a hierarchy of operations. Then we'll build an RPNCalculator class that contains just a few methods.

Ordinarily for a project like this, we would use as many existing classes as possible. Why re-invent the wheel, right? Ordinarily we would use the java.util.Stack class to implement our Stack.

Ordinarily.

But not for a3. For a3, in order to gain some more experience with data structures and exceptions, you will do something extraordinary. You will implement your own Stack. You've already created an implementation during an exam, a seriously stressful situation, so this will surely be a cinch.

Fun will be had by all. Read on, and get started now!

1 Submission requirements

This assignment must be completed by Sunday March the 8th at or before 23:59:59.

Your code must all be in the ca.bcit.comp2522.assignments.a3 package of your COMP 2522 IntelliJ project.

We will mark the code as it appears during the final commit before the due date and time.

2 Grading scheme

Your assignment will be marked out of 10. A rubric will be made available on the Learning Hub to help guide your priorities.

3 Implementation requirements

Implement the following:

1. We are going to build a Reverse Polish Notation calculator. The RPNCalculator will accepts two command line arguments. The first command line argument must be an int which represents the initial size of the Stack we will use to store operands. The second command line argument must be a String in double quotes that contains a valid Reverse Polish Notation expression. We will place the main method inside a class called RPNCalculator, and we will invoke the program like this:

   ```
   java RPNCalculator 10 "1 2 3 4 5 6 7 8 9 10 + + + + + + + + +"
   ```

   The program must respond by printing the expression inside a set of square brackets followed by the result:

   ```
   [1 2 3 4 5 6 7 8 9 10 + + + + + + + + +] = 55
   ```

2. **Start by defining an interface called Operation**. An Operation is a function that has a symbol. It requires two operands. The Operation interface contains two public methods, char getSymbol(), which returns the operation symbol to the user, and int perform(int operandA, int operandB), which is a blueprint for performing a math operation. That's all we need to put inside the Operation.

3. Note that the Operation doesn't contain any instance variables because it's an interface. An interface just tells us how to interact with something. It makes no demands about implementation. Since it's an interface, it doesn't have a constructor either. We need to create a level of abstraction, another layer of the onion if you will, that adds the instance variable that will store the actual symbol being used.

4. **Define an abstract class called AbstractOperation that implements Operation**. Ensure AbstractOperation is an abstract class. It must contain a single protected instance variable called operationType, a char. Add a constructor which accepts a char and assigns it to operationType. Add an

accessor that returns the char. The accessor must be called getSymbol() and it must be final. We don't want any subclasses overriding it.

5. We are going to confine our exploration to the basic operations: +, -, *, and /. **Create four classes that extend AbstractOperation. They must be called AdditionOperation, SubtractionOperation, MultiplicationOperation, and DivisionOperation**. Each of these classes must contain a static constant char called ADDITION_CODE or SUBTRACTION_CODE or MULTIPLICATION_CODE or DIVISION_CODE, each of which is assigned the value '+', '-', '*', or '/' (I will let you decide which goes with which!). The constructor must pass the constant to the superclass constructor. Ensure each class provides a concrete implementation of perform too. That's all you need inside each class.

6. The hierarchy we just created is really quite elegant. We defined the concept of an Operation, and we can interact with an Operation by getting its symbol and passing it two operands to operate on. We further added an abstract implementation of Operation which added an instance variable, a constructor to assign it, and an accessor to acquire it. Then we created four concrete implementations of Operation. Each one is many things at once. For example, an AdditionOperation is-an AbstractOperation and it is-an Operation, too.

7. So we have this wonderful little inheritance hierarchy. It's easy to add more operations. In our project, any operation that can be represented as a single char can be added. Do I smell a bonus? Perhaps...

8. **I'd like you to implement the Stack next**. We will implement a fixed size, non-resizable Stack using an olde fashioned array of int:

   (a) The Stack must have two instance variables, an array of int called stackValues, and an int called count.

   (b) The Stack constructor must accept an integer representing the size of the array to create. If the size is less than one, throw an IllegalArgumentException.

   (c) Add a capacity( ) method that returns the size of the Stack.

   (d) Add a size( ) method that returns the number of elements in the Stack, i.e., the count.

   (e) Add a method called unused( ) which returns the amount of space left in the Stack.

   (f) Add a method called push(int value) which accepts an integer called value. This method pushes the value onto the Stack. If the Stack is already full, this method must throw a **checked exception called StackOverflowException**. You must create this exception. Pass the message "This stack is full!" to the StackOverflowException constructor. If, however, there is room, push the value onto the Stack. The next call to pop will remove this item( ) and return it. The next call to peek( ) will return a reference to this item without removing it.

   (g) Did someone say pop? Add a method called pop( ) which accepts no parameters and returns an int. If someone tries to pop a value from an empty Stack, ensure this method throws a **checked exception called StackUnderflowException**. You will have to add this exception to your project too. Pass the message "Cannot call Stack.pop() on an empty stack!" to the StackUnderflowException exception.

   (h) Finally, add a method called peek( ) which does NOT remove anything from the Stack, but does return the value on top of it. If the Stack is empty, throw a new StackUnderflowException and pass the message "Cannot call Stack.peek() on an empty stack!".

9. The pieces are in place. We just need to create the main class. **Implement a class called RPNCalculator**. It's time for the fun stuff!

   (a) RPNCalculator contains a single integer constant called MIN_STACK_SIZE which stores two. The smallest RPN calculation is two operands followed by a single operation.

   (b) RPNCalculator contains a single instance variable of type Stack called (wait for it) stack.

   (c) The constructor must accept an integer called stackSize. If this integer is less than MIN_STACK_SIZE the constructor must throw an IllegalArgumentException. Otherwise instantiate a Stack of that size and assign the address of this new object to the instance variable.

(d) **Implement a method called public int processFormula(final String formula)**:

i. This method must throw an IllegalArgumentException if formula is equal to null or if it is a string of length zero.

ii. Otherwise, instantiate a Scanner object, passing the formula to its constructor. We will parse the formula using an instance of the Scanner class.

iii. Here's the algorithm I'd like you to use. While the Scanner object's hasNext( ) method returns true, check if the hasNextInt( ) method returns true, too. If it does, we know the next token in the formula String is an operand that we can push onto the Stack. If this is the case, go ahead and push that int onto the stack with a helper method called void push(final int operand).

iv. **The method void push(final int operand)** must check that unused( ) does not return zero. If it does, we must throw a StackOverflowException with an appropriate message. Otherwise, push the operand onto the Stack.

v. Otherwise, it must be an operation. If it's an operation, we must scan the operation and use it to instantiate the correct Operation descendant. Do this in a **helper method called private Operation getOperation(final char symbol)**.

vi. Inside private Operation getOperation(final char symbol), use a switch statement to evaluate symbol. If it's '+', return a new AdditionOperation object, if it's '-', return a new SubtractionOperation, etc.

vii. The return value must be assigned to a variable inside the processFormula method whose data type is Operation. This makes our code flexible. Now we can use polymorphism! We can create and use any kind of Operation we like. We can define new and novel operations. All we have to do is add a line inside the switch statement in the getOperation method.

viii. In the switch statement, the default case must throw a checked exception called **InvalidOperationTypeException** (you will need to create this). Pass the errant operation to the InvalidOperationTypeException constructor.

ix. Otherwise, processFormula must pass the instance of Operation created in the helper method to a method called perform( ), and then invoke a public method called getResult( ).

(e) **public int getResult()** must use the peek( ) method in the Stack to retrieve the current value in the Stack, i.e., the result. If the size of the Stack is zero throw a new StackUnderflowException and pass the message, "There are no operands!".

(f) Finally. The moment you've been waiting for. The one method that rules them all. **private void perform(final Operation operation)** will accept the Operation object instantiated by processFormula and its helpers. Check that operation is not null. If it is, throw an IllegalArgumentException using the message, "Operation cannot be null!". Otherwise, pop the top two operands and pass them to the Operation's perform method. Use the push( ) method to push the result onto the Stack.

10. **In order to give you a little start, I've included the main method which you must insert into your RPNCalculator class**. You may not change this. Not a single thing:

```
/**
 * Drives the program by evaluating the RPN calculation provided as
 * a command line argument.
 *
 * Example usage: RPNCalculator 10 "1 2 +"
 *
 * Note that the formula MUST be placed inside of double quotes.
 *
 * @param argv - the command line arguments are the size of the Stack
 *               to be created followed by the expression to evaluate.
 */
public static void main(final String[] argv) {

    // Checks for correct number of command line arguments.
```

```
        if (argv.length != 2) {
            System.err.println("Usage: Main <stack size> <formula>");
            System.exit(1);
        }

        // Initializes stack and RPNCalculator.
        final int stackSize = Integer.parseInt(argv[0]);
        final RPNCalculator calculator = new RPNCalculator(stackSize);

        try {
            System.out.println("[" + argv[1] + "] = "
                    + calculator.processFormula(argv[1]));
        } catch (final InvalidOperationTypeException ex) {
            System.err.println("formula can only contain integers, +, -, *, and /");
        } catch (final StackOverflowException ex) {
            System.err.println("too many operands in the formula, increase the stack size");
        } catch (final StackUnderflowException ex) {
            System.err.println("too few operands in the formula");
        }
    }
```

That's it! I'll furnish some unit tests later that include some formulas your RPNCalculator must be able to evaluate, and which ensure your methods are throwing the right checked Exceptions, etc. Good luck, and have fun!