COMP 2522
Object Oriented Programming
Lab 4

Yue Wang
yue_wangn@bcit.ca

Due at noon on Sunday, February 12, 2023

1 Introduction

Welcome to your fourth COMP 2522 lab. This is the final lab before the midterm. This lab relies heavily on the concepts of polymorphism and abstraction, which rest comfortably upon the concepts we have explored this term: encapsulation, information hiding, coupling and cohesion. We will use interfaces and abstract classes to generate layers of the "onion" that are modular and easy to grow.

You will implement a novel calculator using abstraction and polymorphism. I would like you to work on this for an hour or three each day until it is due. Fluency comes from using a language, and I'd like you to use Java every day. Take your time. Help each other. Ask me lots of questions. Much of the information required to complete this lab will be shared during lecture on Monday. Pay close attention!

Good luck!

2 About Reverse Polish Notation

For this lab, you will examine abstraction and inheritance through the lens of mathematics.

In 1924, a Polish mathematician named Jan Łukasiewicz (woo-kah-SHEV-eetch) 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 this lab. For this lab, in order to gain some experience with abstraction, interfaces, and polymorphism, you will do something extraordinary.

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

3 Submission requirements

This lab must be completed and submitted to D2L at or before 11:59:59 AM on Sunday, February 12, 2023.

Don't wait until the last minute as you never know what might happen (computer issues, no Internet, power outage, ...). Submit often, and please make sure your comments are short and clear and specific.

You must not ignore Checkstyle warnings or the code style warnings produced by IntelliJ. **You must style your code so that it does not generate any warnings.**

When code is underlined in red, IntelliJ is telling us there is an error. Mouse over the error to (hopefully!) learn more.

When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more. If the warning seems silly, tell me about it, and I will investigate and possibly ask the class to modify some settings to eliminate it once and for all!

4 Grading scheme

Figure 1: This lab, like all labs, will be graded out of 5

Your lab will be marked out of 5. Consult the rubric available on D2L. On a high level, expect that marks will be assigned for:

1. (3 point) Meeting the functional requirements in this lab, i.e., does the code do what it is supposed to do?

2. (1 point) Meeting non-functional Java OOP design requirements such as encapsulation, information hiding, minimizing mutability, maximizing cohesion and minimizing coupling, making good choices about associations and thinking about Demeter, etc.

3. (1 point) Writing code that is self-documenting and well-styled that generates no warnings from IntelliJ or Checkstyle, with correct and complete Javadocs, using short and atomic methods correctly encapsulated in classes that are well defined.

5 Style Requirements

Continuing with this lab, there are code style requirements that you must observe:

1. Your code must compile.

2. Your code must not generate any IntelliJ code warnings or problems.

3. Your code must execute without crashing.

4. Your code must not generate any Checkstyle complaints (unless they are complains I have specifically said you can ignore).

5. Don't squeeze your code together. Put a blank space on either side of your operands, for example. I will be assessing the readability and clarity of your code.

6. All of your program classes must be in package ca.bcit.comp2522.xxx (replace xxx as required by the assignment, lab, quiz, etc.). For example, today's work should go in the ca.bcit.comp2522.labs.lab0X package.

7. All classes require Javadoc comments including the @author tag AND the @version tag. Class comments go after the package and import statements. There should be no blank lines between a class comment and the class it describes.

8. Public constants require Javadoc comments, private constants do not. Constants should be the first thing in your class.

9. Constants should be static and final, are often public, and should be followed by instance variables.

10. Instance variables have private or protected visibility, never public or default visibility.

11. Public and protected methods require Javadoc comments including @param tag(s)the @return tag, and the @throws tag where needed (we won't worry about throws until we talk about exceptions in depth).

12. A method's comment must begin with verbs describing what the method does, i.e., Calculates, Returns, Sets, Prints, etc. Note that we use present tense in Java – Returns, not Return. Prints, not Print.

13. The @return and @param tags go AFTER the description.

14. Private methods require non-Javadoc comments (the comments that start with a slash and a single asterisk).

15. Do comment complicated logical blocks of code inside methods with sparse, clear inline comments.

16. Do not use magic numbers (you must use constants instead). Remember that a magic number is any numeric literal. A constant can be local in a method (use the final keyword with it) or class-level (make it static and ALL_CAPS).

17. All method parameters that are object references must be made final (so we don't forget parameters are passed by value):

(a) Nice to prevent erroneous assignments, and necessary if parameter is referenced by inner class, but that is perhaps a little advanced for now.

(b) References made final mean that the reference, once pointing to an object, cannot be changed to point at a different object.

18. Consider making your methods final:

    (a) Making a method final prevents subclasses (those that inherit the method) from changing its meaning.

    (b) Final methods are more efficient (the methods become inline, thus avoiding the stack and generating overhead).

19. Data and methods that work together must be encapsulated in the same class.

20. Code duplication must be minimized.

21. The values of local variables that are primitives are set when they are declared, and local variables are not declared until they are needed.

22. Every class that stores state needs an equals method, a hashCode method, and a toString method.

23. In general, we enforce a fairly hard maximum method length of 20 lines of code. Aim for 10 (excluding braces).

6 Coding Warmup

The following must be added to ca.bcit.comp2522.lab04:

1. This is an open-ended warmup exercise. I would like you to demonstrate what you understand about inheritance. You will solve a famous OOP problem – how to organize the rectangle and the square. Are they related? Which one is the superclass? Which one is the subclass? Or are they independent siblings that share a common parent?

2. You must implement an inheritance hierarchy for the following classes: Quadrilateral, Parallelogram, Rectangle, and Square. Use Quadrilateral as the superclass of the hierarchy.

3. Implement a Point class to represent the points in each shape. A Point has an x and a y (doubles, of course).

4. Make the hierarchy as deep, i.e., as many levels, as necessary. Be thoughtful about the instance variables that need to go in each class. To get you started, I recommend that the private instance variables inside Quadrilateral be the x-y coordinate pairs of the four endpoints.

5. You must organize the hierarchy so that the relationships between the classes make sense. Consider qualities like length, parallelism, angles, etc.

6. Include a method called describe that prints a short String description of the Shape.

7. Write a Driver that instantiates one object from each of the classes and outputs each object's description. Your Driver may only use variables of type Quadrilateral (hint: use polymorphism!)

7 Coding Exercise

The following must be added to ca.bcit.comp2522.lab04:

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 ca.bcit.comp2522.lab04.Operation**. An Operation is a function that has a symbol. It represents an operation that accepts two operands, i.e., it represents a binary operator. 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 ca.bcit.comp2522.lab04.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**. Ensure the classes are in the ca.bcit.comp2522.lab04 package. Each of these classes must contain a static constant char called ADDITION_SYMBOL or SUBTRACTION_SYMBOL or MULTIPLICATION_SYMBOL or DIVISION_SYMBOL, each of which is assigned the value '+', '-', '*', or '/' (I will let you decide which goes with which!). The constructor must pass the class constant to the superclass constructor, where it will be stored in the operationType instance variable. 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 old 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 or equal to one, throw an IllegalArgumentException. If it is 2 or greater, initialize the stackValues array to the provided size.

   (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 **BufferOverflowException**. 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 **BufferUnderflowException**.

(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 BufferUnderflowException.

9. The pieces are in place. We just need to create the main class.

10. **Implement a class called ca.bcit.comp2522.lab04.RPNCalculator**. It's time for the fun stuff!

(a) RPNCalculator contains a single integer constant called MIN_SIZE which is equal to 2. The smallest RPN calculation is two operands followed by a single operation.

(b) RPNCalculator contains a single instance variable of type ca.bcit.comp2522.lab04.Stack called (wait for it) stack.

(c) The constructor must accept an integer called stackSize. If this integer is less than MIN_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.

iv. Otherwise, if the next token is not an integer, 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)**.

v. 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.

vi. 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.

vii. In the switch statement, the default case must throw an IllegalArgumentException. Pass the errant operation to the IllegalArgumentException constructor.

viii. 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) This is 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.

(f) **public int getResult()** must use the peek( ) method in the Stack to retrieve the current value in the Stack, i.e., the result.

11. **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 IllegalArgumentException ex) {
        System.err.println("formula can only contain integers, +, -, *, and /");
    } catch (final BufferOverflowException ex) {
        System.err.println("too many operands in the formula, increase the stack size");
    } catch (final BufferUnderflowException ex) {
        System.err.println("too few operands in the formula");
    }
}
```

Good luck, and have fun!