Homework 9: Abstract Data Types & RPN Calculator

In this homework, you will create an abstract data type for a generic stack. You will then use this stack to build an RPN calculator. Now that you have experience with Java, you'll create these files yourself without starter files. However, feel free to refer to previous homework for examples. Continue to follow good practices of testing and commenting.

Problem 1: Implementing a Stack

The goal of this problem is to create a generic stack class Stack and associated tests StackTest. Java has a built-in Stack class in java.util.stack, but in this problem you will be creating your own class instead. Remember that a stack is a first-in, first-out queue. Only the element on the top of the stack can be directly accessed. Your stack class should be defined as:

```
class Stack<E extends Comparable<E>>
```

Stack is an abstract data type, meaning that the internal representation of the stack should be invisible to the user. For this homework, you will use your List class from Homework 6 as a private instance variable within the Stack class to represent the elements of the stack. Copy your List.java file from Homework 6 into your Homework 9 project. If you didn't get List working in Homework 6, you will need to complete it now. Because the class is abstract and the internal representation is private, it should be possible to replace the List with another representation such as ArrayList without causing any changes visible to a user of the Stack class.

Your stack should be generic, supporting any object E that extends Comparable. For example, E could be String, Integer, etc. Your stack should support the following public methods, with the maximum time complexity specified below. N indicates the size of the stack.

Method	**Complexity**	**Summary**
Stack()	O(1)	Construct empty stack
int getSize()	O(N) or O(1)	Number of elements
boolean isEmpty()	O(1)	Is stack empty?
E peek() throws EmptyStackException	O(1)	Return top element without removing it
E pop() throws EmptyStackException	O(1)	Remove and return top element

Method	Complexity	Summary
void push(E e)	O(1)	Push e onto top of stack
void clear()	O(1)	Remove all elements in constant time
boolean equals(Stack<E> s)	O(N)	Compare two stacks
String toString()	O(N)	Convert elements of stack to space-delimited string

Note that peek and pop throw an EmptyStackException if the stack is empty. You will need the following import to define this exception:

```
import java.util.EmptyStackException;
```

Problem 2: RPN Calculator

The goal of this problem is to create a four-function RPN integer calculator that can accept math statements from the command line or a file and print the result. The calculator should use your Stack<Integer> class to hold operands and results as they are pushed on the stack. All operands must be Integers, and division truncates the result.

- In Reverse Polish Notation (RPN) expressions, operators always *follow* their operands (like Racket in reverse!):
  - 3 - 1 becomes 3 1 -
  - 3 * (4 + 5) - 6 becomes 3 4 5 + * 6 -
  - How would you evaluate the following Reverse Polish Notation expression by hand? (The answer should be **3**.): **8 23 + 4 - 9 /**
- Note that since each operator always has exactly two operands, Reverse Polish Notation does not require parentheses to enforce order of operations!

Write a StackCalculator class implementing the following methods, and associated StackClaculatorTest.

Method	Complexity	Summary
StackCalculator()	O(1)	Constructor
int evalRPN(String[] instructions)	O(N)	Perform calculation
int calculateStream(InputStream stream)		Read a line from the input stream, break it into tokens,

Method	Complexity	Summary
		and evalRPN the tokens
int cacluateUser()		calculate from System.in
int calculateFile(String filename)		calculate from specified file

evalRPN() takes an array of strings called the *instructions* representing numbers or the four operations +, -, *, /. It does not need to accept malformed instructions, such as undefined operators or division by 0. It evaluates the instructions by pushing numbers onto the stack. When it finds an operation, it pops the top two numbers off the stack, performs the operation on those numbers, and pushes the result back on the stack. The final result is the value on the top of the stack after executing all of the instructions. For example, if instructions are [“3” “4” “5” “+” “*” 6 “-”], evalRPN’s stack looks like:

Instruction	Stack	Notes
3	3	Push 3 on stack
4	4 3	Push 4 on stack
5	5 4 3	Push 5 on stack
+	9 3	Pop 5 and 4, add, push sum of 9
*	27	Pop 9 and 3, multiply, push product of 27
6	6 27	Push 6 on stack
-	21	Pop 6 and 27, subtract 27-6, push result of 21

Note that Integer.parseInt(s) converts a string s to an integer.

You will likely find the following code useful for reading from the keyboard and a file:

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.InputStream;
import java.util.Scanner;

 public int calculateStream(InputStream stream) {
     try (Scanner scan = new Scanner(stream)) {
         String in = scan.nextLine();
         String[] instructions = in.split(" ");
         int result = evalRPN(instructions);
```

```
            System.out.println(result);
            return result;
        }
    }

    public int calculateUser() {
        return calculateStream(System.in);
    }

    public int calculateFile(String filename) {
        Path p = Path.of(filename).toAbsolutePath();
        try (InputStream is = Files.newInputStream(p)) {
            return calculateStream(is);
        } catch (Exception e) {
            System.out.println(e);
            return 0;
        }
    }

    public static void main(String[] args) {
        StackCalculator calc = new StackCalculator();
        calc.calculateUser();
        calc.calculateFile("hw9/eqn.txt");
    }
```

calculateStream takes a Java InputStream, such as what you are typing at the keyboard (System.in), or what you read from a file. InputStreams only read one character at a time, so calculateStream then creates a Scanner object to read an entire line. The string .split(" ") method splits the line into an array of strings based on spaces. For example, `"dog cat mouse froggy".split(" ")` would produce an array (`"dog", "cat", "mouse", "froggy"`).

calculateUser() invokes calculateStream with System.in, the default input stream.

calculateFile() invokes calculateStream with a file. It first converts the filename to an absolute path (for example, hw9/eqn2.txt converts to /Users/harris/Documents/Classes/CS60/Assignments/hw9/hw9/eqn.txt). When running Java in VSCode, the base path defaults to the directory containing your project (e.g. /Users/harris/Documents/Classes/CS60/Assignments/hw9), so if your project is also called hw9 and a file named eqn.txt is in the top level directory of your project, you would want the absolute path given above. calculateFile then opens a newInputStream for that file using the Files class. Observe the *try-with-resources* pattern, which automatically closes the InputStream after the try block. If there is a problem such as a FileNotFoundException, the catch block prints the error message.

The main() tests a user input and a file input. When you run StackCalculator (click the triangle icon in the upper right), the terminal window will wait for you to type a line, then performs the calculation you entered. For example, if you type 40 2 +, it should calculate 42. Next, main() reads another calculation from a file. If eqn.txt is in the top level of your VSCode project and contains 6 9 *, the terminal will print 54.

What to Turn In

Turn in your Stack.java, StackTest.java, StackCalculator.java, and StackCalculatorTest.java files in Gradescope. Each should pass the autograder, conform to the CS60 style guidelines, and be written in a way to meet the time complexity specifications.

*The autograder also expects the classes to be **public**. The autograder will **NOT** run if the Stack and Stack Calculator classes are not public* (i.e. `public class StackCalculator`, `public class Stack`). So make those changes before submitting ONLY for the autograder!

Some other common reasons the autograder might fail:

- The package name is incorrect (not `com.gradescope.hw9`)
- Importing an unnecessary package that the autograder does not have (e.g. `import com.gradescope.hw6.List`).
- The .java file name does not exactly match the class name (e.g. "`Stack Calculator.java`" with a space in the name).
- Using `==` rather than `.equals()` to compare objects.
- The autograder uses a reference implementation of List.java from HW6, so it may fail if your List implementation does not conform to that spec.

Rubric

#	Name	Autograder	Style	Total
1	Stack	25	5	30
2	StackTest	8	2	10
3	StackCalculator	25	5	30
4	StackCalculatorTest	8	2	10
		82.5%	17.5%	80