CSE 331 18su Final Exam

Name _____

Welcome to the 331 Final!

Please wait to turn the page until everybody is told to begin.

Friendly reminders:

1. **Attempt every problem.** You can receive partial credit.
2. **Write clearly!** We can't give you credit for answers that we can't read.
3. **If you make a mistake** and need to correct it, be sure that your final answer is very clearly indicated.

This is an opportunity to show off what you have learned. Good luck and have fun!

Part 1: Postfix Calculator

Please turn to the last page of the exam and rip off the page containing the source code for PostfixCalculator. Refer to the PostfixCalculator code to answer the following questions.

1.1. Your friend proposes that instead of having a stack as a private instance field, PostfixCalculator should extend the Stack class. Would this be proper use of subclassing? Explain why or why not.

PostfixCalculator should not extend the stack class since PostfixCalculator's stack is part of its private state. If PostfixCalculator extended Stack, and a client called its pop method, then the calculator's internal state would get messed up (rep exposure). Also, PostfixCalculator is not substitutable for a stack.

Alternative answer: PostfixCalculator is not substitutable for Stack becasue it does not have the same public methods. Therefore it is not a proper subtype of Stack and should not extend the Stack class.

1.2. Your company will implement an amazing and advanced new technology, five-function calculator. Your manager gives you part of the specification, including the class overview and the method signatures.:

```
/**
 * Makes integer arithmetic computations, given a sequence of tokens
 *   in postfix notation
 * Accepts two kinds of tokens, integer and operator. Here are their formats:
 *     Integer is a sequence of decimal characters, e.g. 5367
 *     Operator is one of the following five characters
 *       +     (add)
 *       -     (subtract)
 *       *     (multiply)
 *       /     (divide)
 *       ^     (power)
 */
public class FiveFunctionCalculator {
    public FiveFunctionCalculator();
    public void accept(String token);
    public boolean hasResult();
    public int getResult() throws IllegalStateException;
    public void clear();
}
```

Your manager tells you to write the implementation and Javadocs. You notice that the given spec is similar to the spec for PostfixCalculator. Can FiveFunctionCalculator be a subclass of PostfixCalculator? Why or why not? If not, explain how you would (or would not) use PostfixCalculator in your implementation. (You do not have to write the implementation and Javadocs in your response, but you may write part of it if you wish to illustrate a point.)

FiveFunctionCalculator can be a subclass of PostfixCalculator because it is substitutable for PostfixCalculator.

More detail: A true subclass has a stronger specification than its superclass, meaning that each method's specification must have a stronger (or equivalent) precondition and a weaker (or equivalent) postcondition.

FFC's accept method has a stronger spec than PC's. PC's accept method requires that tokens passed in to it are legal. The language for FFC is a superset of the language for PC, it accepts everything tha PC would accept, and more. Therefore, FFC has a weaker precondition for its accept method. PC does not specify its behavior in the case that illegal tokens are passed in. Therefore, FFC is within spec when it executes computations using tokens that would be illegal for PC.

Possible gotcha: You might think that a FFC violates the principle of least surprise. A client could be surprised if they had an object they thought was a PC, but secretly the object was an FFC, because the client could pass in the "illegal" token "^" and not get an exception. However, the client should not be surprised in this case because PC does not promise to throw an exception for illegal tokens. Its behavior for illegal tokens is unspecified.

1.3. According to the specification of java.util.Stack, the pop method throws an EmptyStackException if there is nothing to pop. The current implementation of accept in PostfixCalculator propagates any EmptyStackException up to the client. Explain why propagating the EmptyStackException is bad design. Rewrite the accept method to correct this issue. Include the implementation of any helper methods or internal classes that you may need.

Propagating the EmptyStackException is bad design because stack is an implementation detail of PostfixCalculator, and the client doesn't know about it, so would be confused if they caught an exception about a stack. Instead, PostfixCalculator should catch the EmptyStackException and throw an exception that is at the client's level of abstraction.

Possible solution:
The code below is an example of *exception translation* because it replaces the EmptyStackException with a different exception at the client's level of abstraction. It calls the IllegalArgumentException's constructor with a message understandable to the client. It passes in the EmptyStackException as the cause (*exception chaining*) so that the client can retrieve it by calling the IllegalArgumentException's getCause method.

```
    public void accept(String token) {
        try {
            compute(token);
        } catch(EmptyStackException ese) {
            throw new IllegalArgumentException("not enough operands
provided for  " + token " operator.", ese);
```

```
        }
    }
```

Alternative solutions: Use IllegalStateException or create a new exception class rather than using IllegalArgumentException.

Note: A common misconception is that accept should not throw *any* exception to the client. In fact, it is important for accept to throw some exception to the client because the situation resulting in an empty stack is due to client error. If there was an exception caused by an internal problem with PostfixCalculator, unrelated to client input, then it would be appropriate for PostfixCalculator to prevent that exception from reaching the client.

Note: It is not appropriate to replace the exception with a println statement, because we cannot assume that the client is reading stdout or stderr. The client might be a program, not a person.

1.4. Your teammate heard that lazy evaluation is all the rage, and proposes to use lazy evaluation in the PostfixCalculator. In the proposed implementation, the accept method would simply accept operands and operators and unconditionally push them all onto the stack. It would not do any computation until the client calls getResult or hasResult. Then it would compute the result all at once. Explain why this design might make it difficult for a client to debug their code that uses PostfixCalculator.

This might cause the client to have trouble debugging since the client might call getResult in their code at a time much later than when they pass tokens into the calculator. Then the client might observe a failure that is very far away from the defect. Good design will cause the failure to be close to the defect.

Note: Failure is the human-observable problem with execution. Defect is the part of the code that is a bug.

Note: This question is about debugging from the perspective of a *client* who is using PostfixCalculator. Some answers pointed out issues with the implementation of PostfixCalculator itself. However, the question does not provide enough information to conclude that there is any bug in the proposed implementation of lazy evaluation, and we specifically asked for an answer describing a client issue.

1.5. Adapter Pattern. Consider the Adder interface.

```
interface Adder {
    public int add(int a, int b);
}
```

Suppose that a client relies on the Adder interface, but decides that they want to use PostfixCalculator because they could not find a better implementation. The problem is that

PostfixCalculator does not have the interface that the client needs. Write an adapter that enables the client to use PostfixCalculator without changing their code that relies on Adder. You do not need to document your code.

Possible answer:

```
class CalculatorAdapter implements Adder {
    PostfixCalculator calc;

    public CalculatorAdapter() {
        calc = new PostfixCalculator();
    }

    public int add(int a, int b) {
        calc.accept(String.valueOf(a));
        calc.accept(String.valueOf(b));
        calc.accept("+");
        return calc.getResult();
    }
}
```

1.6. Generics. Find and correct improper use of raw types in the code for PostfixCalculator.
a) Write the line number(s) in which improper use of raw types occurs.
b) Rewrite each of those lines to be correct.
c) Explain why raw types in Java are not type safe.

a) Line 21 shows improper use of raw types.
b) stack = new Stack<>(); OR stack = new Stack<Integer>();
c) The following code illustrates why raw types are not type safe:

```
Stack rawStack = new Stack<Integer>();
Stack<Integer> maybeNotInts = rawStack;
rawStack.push(new Object());
Integer i = maybeNotInts.pop(); // ClassCastException
```

That is, any type of object could be pushed into a raw Stack, but code that was popping something out of a Stack<Integer> would expect an Integer and it might not be an Integer, if the object looked like a raw Stack to the code that was pushing the object.

More detail: Compiler would not throw an error since type erasure means that raw Stack and Stack<Integer> have the same runtime type so it is legal to assign an object with type raw Stack to a variable with type Stack<Integer>. However, this is for backwards compatibility and should not be done. Compiler would throw warning since it can no longer ensure type safety.

Note: Raw types were addressed in depth in the reading (EJ2: 23 / EJ3: 26), and only briefly in the slides, so students who did not read that chapter might have had to guess at what a "raw type" is.

1.7. Time Management. Your teammate is working on the implementation of the intrepid TrigonometricCalculator. Your teammate sets the following goal: "Implement and test 90% of the code for TrigonometricCalculator by Friday, September 13 at 5pm." Describe why this might not be a good goal, and write a better goal.

This is a bad goal because it's impossible to know what 90% of the code is until you've written 100% of the code. Also, the amount of code written is not always proportional to the amount of progress made. It's possible that 90% of the code will be finished for 75% of the duration of the project.

Possible better goal: The following goal is better because it's more meaurable.

Implement and test the add, subtract, multiply, divide, power, sin, cos, tan, cot, sec, and csc capabilities for TrigonometricCalculator by Friday, September 13 at 5pm.

Part 2: MapReduce

Let us define MapReduce as an operation that takes in a collection of numeric values and performs two operations: map and reduce, which we will define as follows:

- A map operation takes in a collection, performs the same operation on every element, and returns the resulting collection.
- A reduce operation Δ is a binary operation with the following properties:
    - commutative: $a \Delta b = b \Delta a$
    - associative: $(a \Delta b) \Delta c = a \Delta (b \Delta c)$

  Any operation that is commutative and associative can be used to reduce a collection to a single scalar value. The arithmetic operations of addition and multiplication are two examples of reduce operations.

The following code performs a MapReduce operation, where the map operation is squaring and the reduce operation is addition.

```
public static int squareSum(Collection<Integer> input) {
    Collection<Integer> squares = new ArrayList<>();

    // first square...
    for(int element : input) {
        squares.add(element * element);
    }

    // then sum
    int sum = 0;
    for(int element : squares) {
        sum += element;
    }

    return sum;
}
```

2.1. Write a static method called mapReduce that can perform a MapReduce operation for **any** map and reduce operations provided by the client, assuming that the provided operations operate over java Integers. (Hint: It may be helpful to think of the map and reduce operations as callbacks that are passed in to the mapReduce method. You may specify the way in which the client must provide the operations.)

Possible answer:

```
public interface Mapper {
    public int map(int a);
}

public interface Reducer {
    public int reduce(int a, int b);
}


public static int mapReduce(Collection<Integer> input,
                    Mapper m, Reducer r, int initialValue) {
    Collection<Integer> mapped = new ArrayList<>();
    for(int element : input) {
        mapped.add(m.map(element));
    }
    int aggregate = initialValue;
    for(int element : mapped) {
        aggregate = r.reduce(aggregate, element);
    }
    return aggregate;
}
```

Part 3

You are helping the UW Waterfront Activities Center to inventory their equipment. Consider the following class hierarchy:

```
Sailboat extends Watercraft;
Sunfish extends Sailboat;
Motorboat extends Watercraft;
```

Consider the following variable declarations

```
Object o;
Watercraft w;
Sailboat s;
Sunfish f;
Motorboat m;
List<? extends Watercraft> lew;
```

<? extends Watercraft> is satisfied by Watercraft or any of its subclasses

```
List<? super Sailboat> lss;
```

<? super Sailboat> is satisfied by Sailboat or any of its superclasses

```
List<? extends Sailboat> les;
```

<? extends Sailboat> is satisfied by Sailboat or any of its subclasses

For each of the following statements, write "ok" if it compiles with no type errors and "error" if there is a type error.

Added some explanations to help clarify these answers (explanation not required in response)

1. `lew = new ArrayList<Watercraft>();` **ok -- see above**
2. `lew = new ArrayList<Sunfish>();` **ok -- see above**
3. `lss = new ArrayList<Sunfish>();` **error -- see above**
4. `les = new ArrayList<Sunfish>();` **ok -- see above**
5. `les = new ArrayList<Motorboat>();` **error -- see above**
6. `les.add(o);` **error -- e.g. if les is List<Sailboat>**
7. `lss.add(f);` **ok -- Sunfish is a subtype of all supertypes of Sailboat**
8. `les.add(f);` **error -- e.g. if les is List<Sunfish1973>, where Sunfish1973 is a subtype of Sunfish (most frequently missed question in this section)**
9. `lss.add(m);` **error -- e.g. if lss is List<Sailboat>**
10. `les.add(m);` **error -- e.g. if les is List<Sailboat>**
11. `lew.add(null);` **ok -- null can be considered as any type**
12. `w = lss.remove(0);` **error -- e.g. if lss is List<Object>**
13. `s = les.remove(0);` **ok -- Sailboat is a supertype of all subtypes of Sailboat**
14. `f = les.remove(0);` **error -- e.g. if les is List<Sailboat>**
15. `m = lss.remove(0);` **error -- e.g. if lss is List<Sailboat>**
16. `m = les.remove(0);` **error -- e.g. if les is List<Sailboat>**

Part 4 (6 points)
Please choose one of the following prompts and write a brief paragraph on that topic below:

- How could applying the ideas you learned in 331 help to make the world a better place?
- What 331 topic do you wish you had learned earlier? When would it have helped?
- What 331 topic do you think will be most useful in the future? Why?

Full credit given for any thoughtful and complete response.

PostfixCalculator Code - Do not write answers on this page.

```
1  /**
2   * Makes integer arithmetic computations, given a sequence of tokens
3   *   in postfix notation
6   * Accepts two kinds of tokens, integer and operator. Here are the formats:
7   *     Integer is a sequence of decimal characters, e.g. 5367
8   *     Operator is one of the following four characters
9   *       +     (add)
10  *       -     (subtract)
11  *       *     (multiply)
12  *       /     (divide)
13  */
14  public class PostfixCalculator {
15      private Stack<Integer> stack;
16
17      /**
18       * Construct a new postfix calculator with clean state
18       * @effects constructs a new PostfixCalculator
19       */
20      public PostfixCalculator() {
21          stack = new Stack();
22      }
23
24      /**
25       * Take in an operator or operand.
26       * @requires the given token must be a legal token
27       * @requires the sequence of tokens passed in so far must form a legal
28       *    (so far) postfix notation sequence representing a legal
29       *    arithmetic operation
30       * @param token - the token to accept
31       * @modifies this
32       * @effects adds the token to this calculator's internal state
33       */
34      public void accept(String token) {
35          compute(token);
36      }
37
38      // Take in an operator or operand
39      // If operand, push it onto stack
40      // If operator, pop its operands and push the result onto the stack
41      private void compute(String token) {
42              if (token.matches("\\d+")) { // non-empty sequence of digits
43              stack.push(Integer.parseInt(token));
44          } else if ("+".equals(token)) {
45              int addend2 = stack.pop();
46              int addend1 = stack.pop();
47              stack.push(addend2 + addend1);
48          } else if ("-".equals(token)) {
49              int subtrahend = stack.pop();
50              int minuend = stack.pop();
```

```
51              stack.push(minuend - subtrahend);
52          } else if ("*".equals(token)) {
53              int factor2 = stack.pop();
54              int factor1 = stack.pop();
55              stack.push(factor1 * factor2);
56          } else if ("/".equals(token)) {
57              int divisor = stack.pop();
58              int dividend = stack.pop();
59              stack.push(dividend / divisor);
60          } else {
61              throw new IllegalArgumentException(
62                                  "Unrecognized token " + token);
63          }
64      }
65
66      /**
67       * Say whether the calculator has a result.
68       * @return true if the calculator has consumed all accepted tokens and
69       *     has a result available, false otherwise.
70       */
71      public boolean hasResult() {
72          return stack.size() == 1;
73      }
74
75      /**
76       * Return the calculator's result. If there are unconsumed tokens
77       *     other than the result, throw an exception.
78       * Client should call hasResult to make sure it is safe
79       *     to call this method.
80       * @return the calculator's result
81       * @throws IllegalStateException if there are unconsumed tokens
82       *     other than the result
83       */
84      public int getResult() throws IllegalStateException {
85          if (stack.size() != 1) {
86              throw new IllegalStateException("No result at this time");
87          }
88          return stack.pop();
89      }
90
91      /**
92       * Reset the calculator to its initial state
93       * @modifies this
94       * @effects clears the internal state of this
95       */
96      public void clear() {
97          while(!(stack.empty())) {
98              stack.pop();
99          }
100     }
101 }
```