**Software Developer Course Assessment**

**Quantitative Assessment Practice**

**Course Name: Advanced Programming (Java)**

**Current Week: 4th July 2024**

**Submission date: 12th July 2024**

**Introduction:**

The purpose of this assessment is to help us understand how the class is doing in terms of the review material that we have covered during the previous couple of weeks. The **only** purpose of this assessment is for us to improve our approach to review and ensure that what we’re currently doing is an effective strategy. Completion of this assessment is **mandatory - if you don’t submit a solution, it will be marked as incomplete. You must complete a minimum of 80% of your assigned QAPs per course – otherwise you will be marked as incomplete for that course no matter how good your other grades are.** If you do submit a solution, it will be marked as complete, as you will receive full marks no matter what your actual performance was – again this is a participation grade.  
  
Again, the goal here is to help you all in the best way that we can, so please do be honest when answering the questions related to how long it took, which resources you used, etc. And please ensure that you do your **own** work – don't just copy off a friend to get it done, earnestly do your best with it. If you can’t get it completely working, give us what you have. While it will be graded, the grade will not count against you, it’s just a way for us to see where everybody is, and to know which concepts, if any, we, as a class, may be struggling with.

**Deadline:** You will have until the end of the day on 12th July (11:59pm) to submit your assessment solutions. Please ensure you answer all the questions outlined in the instructions portion of this document as well in your submission.

**Marking:** In this program core evaluation is marked with one of three possible marks: *Incomplete, Pass, Pass Outstanding.* For QAPs, though, where incomplete marks are more important for our own information as well as for the information of the student, we wanted to increase the resolution of our grading system. Therefore, QAPs are marked on a scale of 1-5. The details of this marking system are summarized in the table below.

|  |  |
| --- | --- |
| **Grade** | **Meaning** |
| 1 | *Incomplete.* Student shows severe lack of understanding of the material – solution is heavily incomplete, non-functional, or completely off base of what the assignment was asking for. |
| 2 | *Partially Complete.* Students show some understanding of the material. Solution may be non-functional or partially functional, but the approach is correct, albeit with some major bugs or missing features. |
| 3 | *Mostly Complete.* Student demonstrates understanding of the major ideas of the assignment. Solution is mostly working, albeit with a few small bugs or significant edge cases which were not considered. Shows a good understanding of the correct approach, and is either nearly a feature-complete solution, or is a feature-complete solution with some bugs. |
| 4 | *Complete (Equivalent to: Pass.)* Student shows complete understanding of assigned work and implemented all necessary features. Any bugs that are present are insignificant (for example aesthetic bugs when testing the functionality of code) and do not impact the core functionality in a significant way. All necessary objectives for the assignment are completed, and the student has delivered something roughly equivalent to the canonical solution in terms of features and approach. |
| 5 | *Complete with Distinction (Equivalent to: Pass Outstanding)* The student demonstrates a clear mastery of the subject matter tested by the QAP. The solution goes above and beyond in some way, makes improvements on the canonical solution, or otherwise demonstrates the student’s mastery of the subject matter in some way. A solution in this category would consider all reasonable edge cases and implement more than the necessary functionality required by the assignment. |

**Java QAP 4.4**

**Joseph Flores**

**July 14, 2024**

**Instructions:**

**You are allowed to complete the assessment problems below in whatever way you can but please answer the following questions/points as part of your submission:**

1. How many hours did it take you to complete this assessment? (Please keep try to keep track of how many hours you have spent working on each individual part of this assessment as best you can - an estimation is fine; we just want a rough idea.)

5 hours. Sorry for the late submission.

1. What online resources you have used? (My lectures, YouTube, Stack overflow etc.)

Notes, lecture videos, chatgpt for code verification

1. Did you need to ask any of your friends in solving the problems. (If yes, please mention name of the friend. They must be amongst your class fellows.)

No

1. Did you need to ask questions to any of your instructors? If so, how many questions did you ask (or how many help sessions did you require)?

No

1. Rate (subjectively) the difficulty of each question from your own perspective, and whether you feel confident that you can solve a similar but different problem requiring some of the same techniques in the future now that you’ve completed this one.

Not difficult. But like what I’ve mentioned in the private message I’ve sent, I was sick for the past 2 weeks and it was only Friday noon that I was able to actually feel a little bit better. I started to finally focus on this last night. It was difficult if you are not even close to 20% well to focus and understand anything.

# Problem#1: Abstract Classes

Consider the following shapes; Ellipse, Circle, Triangle, EquilateralTriangle. Each shape should have a name, a method to compute its perimeter, and another method to compute its area. The name should be an instance variable of type String. Design your inheritance hierarchy with the common features in the **Abstract** superclass Shape. Notice that the area and perimeter are common to all Shapes, but we don’t know how to compute the area or perimeter for a general shape.

The ellipse class has a major and minor axes a and b, respectively. The constructor should assign the largest value to a and smallest to b. The area and perimeters of an ellipse are:

Perimeter = P = π [Note that if *a* = *b* = *r*, then P = 2π*r*]

Area = A = π*ab*

The Triangle class has three instance variables side1, side2, and side3. The formula for the area and perimeter of a general Triangle with sides A, B, and C is given by.





The condition for any three positive values to make sides of a Triangle is:

side1+side2>side3 and side2+side3>side1 and side3+side1>side2

You need to check this condition inside the constructor. If it is not satisfied, print an error message and terminate the program, otherwise make your Triangle object.

The three sides of the equilateral triangle are equal.

Make a Test class where you make objects from the different classes and store them in an array of type Shape. Then, make a loop and print the objects name, area, and perimeter through toString i.e. you need to override toString in the Shape class only.

**Deliverables:**

**Complete and working-class files with proper comments.**

* **Shape.java**
* **Circle.java**
* **Ellipse.java**
* **Triangle.java**
* **EquilateralTriangle.java**
* **Demo.java**
* **Screenshot of the running code’s output**

# Problem#2: Interfaces

Some OOP languages such as C++ allow a sub-class to inherit from more than one super class (multiple-inheritance). While this has some advantages, it makes such languages complex. To avoid such complexities, Java does not allow for multiple-inheritance. However, a lot of the advantages of multiple-inheritance can be achieved using **Interfaces**.

 An interface is similar to a class but with the following restrictions:

* All methods are implicitly **abstract** and **public**
* An interface cannot have instance variables. However, an Interface may have constants (final variables) and these are implicitly public and static. Also they are inherited by any class that implements the interface.
* An Interface can extend another interface and it is implemented by a class using the ***implements*** keyword. In fact, a class may implement any number of interfaces.

Consider an interface Scalable with a method scale of type void. It takes the scaling factor as a parameter. Make the shape class defined above implement the Scalable interface. Note that since Shape is abstract, it does not have to implement scale method.

Make the appropriate subclasses override scale method by multiplying their instance variables by the scale factor.

Modify the above Test class so that you add a static method that receives an array of Type Scalable, and a scale factor. This method should visit all the elements of the Scalable array and call the scale method with the scale factor passed to the static method. You should print your objects before and after scaling.

**Deliverables:**

**Complete and working-class files with proper comments.**

1. **Shape.java**
2. **Circle.java**
3. **Ellipse.java**
4. **Triangle.java**
5. **EquilateralTriangle.java**
6. **Scalable.java**
7. **Demo.java**
8. **Screenshot of the running code’s output**

**Problem#3: Exception Handling**

File *CountLetters.java* contains a program that reads a word from the user and prints the number of occurrences of each letter in the word. Save it to your directory and study it, then compile and run it to see how it works. In reading the code, note that the word is converted to all upper case first, then each letter is translated to a number in the range 0..25 (by subtracting 'A') for use as an index. No test is done to ensure that the characters are in fact letters.

1. Run CountLetters and enter a phrase, that is, more than one word with spaces or other punctuation in between. It should throw an ArrayIndexOutOfBoundsException, because a non-letter will generate an index that is not between 0 and 25. It might be desirable to allow non-letter characters, but not count them. Of course, you could explicitly test the value of the character to see if it is between 'A' and 'Z'. However, an alternative is to go ahead and use the translated character as an index, and catch an **ArrayIndexOutOfBoundsException** if it occurs. Since you want don't want to do anything when a non-letter occurs, the handler will be empty. **Modify this method to do this as follows:**

* **Put the body of the first for loop in a try.**
* **Add a catch that catches the exception, but don't do anything with it.**

**Compile and run your program.**

2. Now modify the body of the catch so that **it prints a useful message** (e.g., "Not a letter") followed by the exception. Compile and run the program. Although it's useful to print the exception for debugging, when you're trying to smoothly handle a condition that you don't consider erroneous you often don't want to. In your print statement, replace the exception with the character that created the out of bounds index. Run the program again; much nicer!

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// CountLetters.java

//

// Reads a words from the standard input and prints the number of

// occurrences of each letter in that word.

//

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import java.util.Scanner;

public class CountLetters

{

public static void main(String[ ] args)

{

int[ ] counts = new int[26];

Scanner scan = new Scanner(System.in);

//get word from user

System.out.print("Enter a single word (letters only, please): ");

String word = scan.nextLine( );

//convert to all upper case

word = word.toUpperCase( );

//count frequency of each letter in string

for (int i=0; i < word.length( ); i++)

counts[word.charAt(i)-'A']++;

//print frequencies

System.out.println( );

for (int i=0; i < counts.length; i++)

if (counts [i] != 0)

System.out.println((char)(i +'A') + ": " + counts[i]);

}

}

**Deliverables:**

**Complete and working-class files with proper comments.**

* **Modified CountLetters.java File**

**Problem#4: Exception Handling**

File *ParseInts.java* contains a program that does the following:

* Prompts for and reads in a line of input
* Uses a second Scanner to take the input line one token at a time and parses an integer from each token as it is extracted.
* Sums the integers.
* Prints the sum.

Save ParseInts to your directory and compile and run it. If you give it the input

10 20 30 40

it should print

The sum of the integers on the line is 100.

Try some other inputs as well. Now try a line that contains both integers and other values, e.g.,

We have 2 dogs and 1 cat.

You should get a **NumberFormatException** when it tries to call *Integer.parseInt* on "We", which is not an integer. One way around this is to put the loop that reads inside a *try* and catch the **NumberFormatException** but not do anything with it. This way if it's not an integer it doesn't cause an error; it goes to the exception handler, which does nothing. Do this as follows:

* **Modify the program** to add a try statement that encompasses the entire *while* loop. The *try* and opening { should go before the *while*, and the *catch* after the loop body. Catch a **NumberFormatException** and have an empty body for the catch.
* Compile and run the program and enter a line with mixed integers and other values. You should find that it stops summing at the first non-integer, so the line above will produce a sum of 0, and the line "1 fish 2 fish" will produce a sum of 1. This is because the entire loop is inside the *try*, so when an exception is thrown the loop is terminated. To make it continue, move the *try* and *catch* inside the loop. Now when an exception is thrown, the next statement is the next iteration of the loop, so the entire line is processed. The dogs-and-cats input should now give a sum of 3, as should the fish input.

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// ParseInts.java

//

// Reads a line of text and prints the integers in the line.

//

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import java.util.Scanner;

public class ParseInts

{

public static void main(String[] args)

{

int val, sum=0;

Scanner scan = new Scanner(System.in);

String line;

System.out.println("Enter a line of text");

Scanner scanLine = new Scanner(scan.nextLine());

while (scanLine.hasNext())

{

val = Integer.parseInt(scanLine.next());

sum += val;

}

System.out.println("The sum of the integers on this line is " + sum);

}

}

**Deliverables:**

**Complete and working-class files with proper comments.**

* **Modified ParseInts.java**