Q1 Ground Rules

0 Points

Grading comment:

During this exam, you may consult:

- Your personal notes from this term
- Your projects and labs (and my comments on them)

You **MAY NOT**:

- Run any python or java code (so, don't use IDLE, PyCharm, IntelliJ, etc...)
- Type any code into any editor or development environment that supports syntax highlighting (i.e. only write code in the boxes provided here on the exam)
- Use any other resource in any medium (book, paper, web, ...)
- Consult with any person other than the instructor

You have 120 minutes (**2 hours**) for the exam, and there are 100 total points available. Once you open and start the exam, you'll have 120 minutes to complete it on gradescope.

☑ Choice 1 of 2:I understand and will abide by the ground rules.

☐ Choice 2 of 2:I do not agree to the ground rules.

Grading comment:

Note: the code samples I provide below are images that you can open in separate tabs if you want, thus reducing the need to scroll back and forth.

Q2 Tracing

10 Points

Grading comment:

Consider the following Java code:

`Main.java`:

```java
import java.util.ArrayList;

public class Main {
    public static void main(String [] args) {
        ArrayList<Ball> balls = new ArrayList<Ball>();
        balls.add(new Basketball("Orange ball"));
        balls.add(new Baseball("White ball"));

        for (Ball b : balls) {
            bounce(b);
        }
    }

    private static void bounce(Ball b) {
        int oldY = b.getHeight();
        b.bounce(2);
        int newY = b.getHeight();
        System.out.println(b
                           + " bounced with force 2, going from height "
                           + oldY + " to new height "
                           + newY);

    }
}
```

Ball.java:

```java
public interface Ball
{
    public void bounce(int force);
    public int getHeight();
}
```

Baseball.java:

```java
public class Baseball implements Ball {
    private int height;
    private String name;

    public Baseball(String name) {
        height = 1;
        this.name = name;
    }

    public String toString() {
        return "Baseball " + name + " with height " + getHeight();
    }

    public int getHeight() {
        return height;
    }

    public void bounce(int force) {
        System.out.println("Bouncing baseball");
        height += (int) (force / 2.0);
    }
}
```

`Basketball.java`:

```java
public class Basketball implements Ball {
    private int height;
    private String name;

    public Basketball(String name) {
        height = 2;
        this.name = name;
    }

    public String toString() {
        return "Basketball " + name + " with height " + getHeight();
    }

    public int getHeight() {
        return height;
    }

    public void bounce(int force) {
        System.out.println("Bouncing basketball");
        height += (int) (force * 2.0);
    }
}
```

What, ***precisely***, will be printed when the `Main` class is run?
Bouncing Basketball Basketball Orange ball with height 6 bounced with force 2, going from height 2 to new height 6 Bouncing baseball Baseball White ball with height 2 bounced with force 2, going from height 1 to new height 2

Q3 Test Coverage

20 Points

Q3.1
10 Points

Grading comment:

Explain why test coverage metrics (such as line or statement coverage) can be helpful in creating thorough test suites.

The test coverage metrics can be helpful in ensuring that the we have not make any errors that the python might not like. It ensures that every line of the code is following the logic of python (not the logic that the user is looking for i.e. the desired output) and makes sure that there are no syntax errors in our code which may otherwise have led to the error in the compilation of code. In short it just makes sure that the code is working perfectly.

Q3.2
10 Points

Grading comment:

Explain how a test suite that achieves 100% statement or line coverage might still not be thorough enough.

Even though, if each of the line of the code is tested perfectly, we may not know if we made the logical errors that is the output that we desired from the code. for example we wanted to test a function that adds two integers, but due to some reason if we mistakenly entered a negative sign instead of positive sign, we may not achieve the right result though the test will be covering each line of the code perfectly without any error from the compiler.

Q4 Refactoring
40 Points

Grading comment:

One thing that is useful for making sure one implements a class's desired behaviors correctly is to decide and write down exactly how the instance variables are supposed to be used in what is called a "class invariant". The invariant is a description of what must be true after the constructor finishes as well as before and after each public method finishes.

Consider the following code for modeling a Deck of cards (I've omitted the `Card` class, which is like the `Card` class you've used in the projects):

```
 1: import java.util.ArrayList;
 2: import java.util.Collections;
 3:
 4: public class Deck {
 5:     private ArrayList<Card> contents;
 6:     private int topIndex;
 7:
 8:     public static final String [] SUITS = {"S", "H", "C", "D"};
 9:
10:     public Deck () {
11:         initializeContents();
12:         topIndex = 0;
13:     }
14:
15:     private void initializeContents() {
16:         contents = new ArrayList<Card>();
17:         for (String suit: SUITS) {
18:             for (int rank = 2; rank <= 14; rank++) {
19:                 contents.add(new Card(rank, suit));
20:             }
21:         }
22:     }
23:
24:     public Card deal() {
25:         if (size() == 0) {
26:             return null;
27:         } else {
28:             Card toReturn = contents.get(topIndex);
29:             topIndex++;
30:             return toReturn;
31:         }
32:     }
33:
34:     public void shuffle() {
35:         Collections.shuffle(contents.subList(topIndex, contents.size()));
36:     }
37:
38:     public int size() {
39:         return contents.size() - topIndex;
40:     }
41: }
```

This implementation was developed according to the following invariant:

- if `topIndex > 0`, `contents` from index `0` to `topIndex - 1` contains the cards that have been dealt, in the order in which they were dealt. In other words:
  - The first card dealt is at index `0`
  - The most recently-dealt card will be at `topIndex - 1`
- if `topIndex <= 51`, `contents` at index `topIndex` and higher contains cards yet to be dealt, in the order they will be dealt (i.e. `shuffle` rearranges that portion of `contents`).

In this question, imagine you are **refactoring this `Deck` class** to make shuffling more efficient, according to the following implementation idea:

Dealing a card from the deck happens in one of two ways:

- If the `Deck` has not been shuffled (i.e. if `shuffle` has not yet been called), deal the card that is at `topIndex` in `contents`.
- If the `Deck` has been shuffled, choose a card at a random index in `contents`.

In this way, **shuffling doesn't need to rearrange** a portion of `contents`. To do this, we will introduce a new instance variable `private boolean shuffled` and then enforce the following class invariant:

- if `topIndex > 0`, `contents` from index `0` to `topIndex - 1` contains the cards that have been dealt, in the order they were dealt. In other words:
  - The first card dealt will be at index `0` after it is dealt.
  - The most recently-dealt card will be at index `topIndex - 1`.
- if `topIndex <= 51`, `contents` at indexes `topIndex` or higher have yet to be dealt.
- `shuffled` is true when the deck has been shuffled and false otherwise.

In the following, you may use these helper methods:

```
/**
 * @return a random number between low (inclusive) and high (exclusive)
 */
 private int getRandomIndex(int low, int high)

/**
 * Swap the cards at indexes i and j
 */
 private void swap(int i, int j)
```

Q4.1 Constructor

5 Points

Grading comment:

Give the code for the constructor so that it follows this new implementation idea.

public Deck (){ initializeContents(); topIndex = 0; shuffled = false; }

Q4.2 deal

15 Points

Grading comment:

Give the code for `deal` so that it follows this new implementation idea.
public Card deal(){ if (size () == 0) { return null; } else { if (shuffled== true){ int cardRandPos=getRandomIndex(topIndex, contents,size()); swap(cardRandPos, topIndex); } else toReturn = contents.get(topIndex); topIndex++; return toReturn;} } }

Q4.3 shuffle

10 Points

Grading comment:

Give the code for `shuffle` so that it follows this new implementation idea.
public void shuffle(){ shuffled = true; }

Q4.4 size

10 Points

Grading comment:

Give the code for `size` so that it follows this new implementation idea.
public int size() { return contents.size() - topIndex; }

Q5 Recursion

30 Points

Grading comment:

Consider a very minimal `SimpleList` Java class, which has the following constructors and methods.

Note that there are no methods for determining the length of a `SimpleList` and no method for testing two `SimpleList`s for equality. The only way to determine the length is to use repeated calls to `get` to find the first index where it returns `null`. For example, if `get(4)` returns a value but `get(5)` returns `null`, then the length of the list is 5.

```
 1: public class SimpleList<E>
 2: {
 3:     /**
 4:      * Constructs an empty SimpleList
 5:      */
 6:     public SimpleList()
 7:
 8:     /**
 9:      * Appends an element to the list.
10:      *
11:      * @param toAdd the element to append.
12:      */
13:     public void add(E toAdd)
14:
15:     /**
16:      * Gets an element from a given position in the list.
17:      *
18:      * @param index the index of the element to get.
19:      *
20:      * @return the element at the given index. If the index is for a
21:      * position beyond the end of the list, returns null.
22:      */
23:     public E get(int index)
24:
25:     /**
26:      * Gets a sublist of the list.
27:      *
28:      * @param startingIndex the position in the list to start from
29:      *
30:      * @return a sub-list of the list, starting at startingIndex and
31:      * going until the end of the list. If the startingIndex is beyond
32:      * the end of the list, returns an empty list.
33:      */
34:     public SimpleList<E> subList(int startingIndex)
35:
36:     /**
37:      * Swaps two elements in the list.  If either of the given indexes
38:      * is beyond the end of the list, has no effect.
39:      *
40:      * @param i the index of one of the elements to swap.
41:      * @param j the index of one of the elements to swap.
42:      */
43:     public void swap(int i, int j)
44: }
```

Q5.1

15 Points

Grading comment:

Now, consider the following method we would like to add to a new `SimpleListProcessor` class:

```
public boolean same(SimpleList<String> a, SimpleList<String> b)
```

This method should return true if the two lists have the same strings, in the same order, and should return false otherwise.

In `same`, there are three cases to consider:

- Both `a` and `b` are empty.

- Exactly one of `a` or `b` is empty.
- Neither `a` nor `b` is empty.

In two of these cases, the answer is computed easily. In the other case, we can determine the answer recursively.

Write `same` **recursively**.

```
public boolean same(SimpleList<String> a, SimpleList<String> b)
```

public boolean same(SimpleList <String> a, SimpleList <String> B){ if(a.get(0)== null && b.get(0)==null){ return true; } if (a.get(0) == null || b.get(0) == null){ return false; } String firstElemA =a.get(0); String firstElemB =b.get(0); if (firstElemA.equals(firstElemB)){ return same(a.sublist(1),b.sublist(1)); } return false; }

Q5.2

15 Points

Grading comment:

In the previous part, you probably used `SimpleList`'s `subList` method for the recursive cases. However, that can be inefficient and we'd like to avoid that. So, let's make a new version, called `same2`, that doesn't use `subList`. `same2` calls a private `same2` that takes an additional parameter to indicate which part of the list to consider:

```
public boolean same2(SimpleList<String> a, SimpleList<String> b)
{
    return same2(a, b, 0);
}

/**
 * @return true iff a[index:] and b[index:] have the same contents.
 */
private boolean same2(SimpleList<String> a, SimpleList<String> b,
                      int index)
```

Write the private `same2` recursively, **without using `SimpleList`'s `subList` method**.

```
private boolean same2(SimpleList<String> a, SimpleList<String> b,
                      int index)
```

public boolean same2(SimpleList <String> a, SimpleList <String> B, int index){ if(a.get(index)== null && b.get(index)==null){ return true; }if (a.get(index) == null || b.get(index) == null){ return false; } String firstElemA =a.get(index); String firstElemB =b.get(index); if (firstElemA.equals(firstElemB)){ return same(a,b ,index+1); } return false; }