Supervision 3 - Kyra

Q45. Using the Java API documentation or otherwise, compare the Java classes Vector, LinkedList, ArrayList and TreeSet.

1. Inheritance:
    1. `Vector`, `ArrayList` and `LinkedList` implemented the `List` interface.
    2. `LinkedList` also implements the `Queue` interface.
    3. `TreeSet` implemented the `SortedSet` interface which inherited the `Set` interface.
    4. Both the `List` interface and the `Set` interface inherited the `Iterable`, `Collection` interface which means all the subclasses in the collections framwork (i.e., `Vector`, `LinkedList`, `ArrayList` and `TreeSet`) are iterable.
2. Duplicate elements:
    1. `TreeSet` only stores unique elements.
    2. `Vector`, `ArrayList` and `LinkedList` can store duplicated elements.
3. The order of elements:
    1. In `TreeSet` - a natural order because it uses a Red-black tree(i.e., self-balancing binary tree) to sort the elements, we can pass a comparator to implement a custom sorting order.
    2. In `ArrayList`, `Vector` and `LinkedList`, all of them follows an insertion order.
4. Thread-safe:
    1. Only `Vector` is thread-safe because objects in `Vector` class is immutable and can be executed parallely.
    2. `TreeSet`, `LinkedList` and `ArrayList` are all dynamic in nature, so they are not thread-safe. However, we can have a synchronised wrapper to make them thread-safe.
5. Element access:
    1. `ArrayList` uses indexes to access elements, so manipulating `ArrayList` might be slower than `LinkedList` because it requires to shift elements around.
    2. `LinkedList` is a recursive data structure uses two fields, one is value and one is a pointer to the next element, so accessing elements requires to traverse the linkedlist from the head element. It stores elements in a heap data structure.
    3. `Vector` is similar to `ArrayList`, which uses indexes but it is immutable and thread-safe.
    4. `TreeSet` is a self-balancing binary tree so it access elements by traversing through the tree from the root node element.

```
// Comments:
```

Q46. Java provides the List interface and an abstract class that implements much of it called AbstractList. The intention is that you can extend AbstractList and just fill in a few implementation details to have a Collections-compatible structure. Write a new class CollectionArrayList that implements a mutable Collections-compatible Generics array-based list using this technique. Comment on any difficulties you encounter.

Question 46 on Github

- Difficulties I have encountered:

    1. Resize and reallocate process:
        - The internal array has a fixed size, if users want to add extra elements into the array, but this operation will cause the array to excess its initial capacity, then I have to initialise a new array with more capacities and then reallocate all the elements from the previous array into the new one.
    2. Casting from `Object[]` to `E[]`, I know from the point of type erasure that all unbound paramaterised type will be cast into `Object`. When I cast `E[]` to `Object[]`, IDE reported a warning `Type safety: Unchecked cast from Object[] to E[]`.
        - Is it because if `E[]` is a bounded paramaterised type (i.e., E extends classA) then only classA and its subtypes are allowed, thus `Object[]` cannot be casted into `E[]` (i.e., `Object[]` is a superclass of classA)?

```
// Comments:
```

Q47 Write a Java program that calculates the average (mean) for a list of integers. Provide three implmentations: 1) using a regular for-loop; 2) using a for-each loop; 3) using an iterator. What are the pros and cons of each?

Question 47 on Github

1. Using a regular for-loop:

    - Pros:

        1. Basic and relatively easy to understand. We can access to the index of the element.

2. We can iterative the collection in any order we want, we can also iterate part of the collection using bounds and steps.

- Cons:

1. We can not delete elements in the collection while iterating through it.
2. We use index to iterate the elements inside a collection, the `for-loop` can not be directly used if the collection is not index based, we have to apply some logic.

2. Using a for-each loop:

- Pros:

1. It normally has the shortest syntax, improve readability and become more clear.
2. Level up the abstraction for programmers as they do not need to work on the implementation details of iteration (either using an index or an iterator).
3. We do not need to use indexes, so some data structures which are not index-based can be iterated through.

- Cons:

1. We can not delete elements in the collection while iterating through it.
2. We can not access to the index of the elements.

3. Using an iterator:

- Pros:

1. We can delete elements in the collection while iterating through it.
2. It is an abstraction, it allows user to process every element of a container while isolating the user from the internal structure of the container.
3. Do not need to use index to access elements, so data structures which are not index-based can be iterated through.

- Cons:

1. Only collections which implemented the iterable interface will have a built-in iterator, programmers need to write an iterator for their customised data structures.
2. We can not access to the index of the element.

```
// Comments:
```

Q48. Explain why the following code excerpts behave differently when compiled and run (may need some research):

```
// Question 48
String s1 = new String("Hi");
String s2 = new String("Hi");
System.out.println( (s1==s2) );
String s3 = "Hi";
String s4 = "Hi";
System.out.println( (s3==s4) );

// results are as follows:
// false
// true
```

1. `s1` and `s2` are references which points to a chunk of memory that store the actural data. Hence, `s1==s2` is a reference equality test, it tests whether the memory addresses of the data each reference points to are the same. Since we have constructed the two strings independently using `new`, the twp strings are two different instances of the `String` class, thus they are stored in different memory locations.
2. Strings are immutable and we have a `String Constant Pool` in Java which is a separate place in heap memory where the values of all the strings which are defined in the program are stored. It aims at reducing memory usage and encourage reuse of the existing instances in memory. So a String declared using `""(double quotes)` which has the same value as one of the existing one will points to the same memory address that stores the data.
3. Hence `s3` and `s4` points to the same immutable string value on the heap memory, they have the same memory address. A reference equality test shows they are indeed the same (i.e, true).

```
// Comments:
```

Q49. Complete part 6 of the ‘Classic collections’ task on Chime

Question 49 on Chime

1. Change the generic parameter of `LinkList` class to add the constraint that the object in the list must be Comparable.

```
public class LinkList<T extends Comparable<T>> implements OopList{
    ...
}
```

2. Why is it safe not to include this constraint in `OopList`
    - Because if we have included this constraint in `OopList`, then all the classes which implements the `OopList` interface needs to give implementation details for the `compareTo(Object obj)` method.
    - Some classes do not need to compare two instances and sometimes it does not make sense to compare two instance for a class.

3. `reorderHighLow` function:

```
public void reorderLowHigh(){
    if (this.length()==0) {
      return;
    }
    Node<T> current = this.head;
    Node<T> previous = null;
    int flag = 1;
    while (current!=null && current.next!=null){
      if (current.value.compareTo(current.next.value)*flag > 0){
        Node<T> temp = current.next;
        current.next = current.next.next;
        temp.next = current;
        if (previous==null) {
          previous = temp;
          previous.next = current;
          this.head = previous;
        }
        else {
          previous.next = temp;
          previous = temp;
        }
      }
      else {
        previous = current;
        current = current.next;
      }
      flag = -flag;
    }
  }
```

```
// Comments:
```

Q50. Write an immutable class that represents a 3D point (x,y,z). Give it a natural order such that values are sorted in ascending order by z, then y, then x

Question 50 on Github

```
// Comments:
```

Q51. Write a Java class that can store a series of student names and their corresponding marks (percentages) for the year. Your class should use at least one Map and should be able to output a List of all students (sorted alphabetically); a List containing the names of the top P% of the year as well; and the median mark.

Question 51 on Github

- I implemented the Student class with Comparator interface and implemented the method to compare two doubles.
- I also construct a shared comparator to sort the array in a decending order according to the percentage.

```
Comparator<String> SortByPercentUp = (s1, s2) -> compare(map.get(s1),
map.get(s2));

...
@Override
public int compare(Double double1, Double double2) {
    return double2.compareTo(double1);
}
```

```
// Comments:
```

Q52. The user of the class Car below wishes to maintain a collection of Car objects such that they can be iterated over in some specific order.

Question 52 on Github

```
public class Car {
private String manufacturer;
private int age;
}
```

- (a) Show how to keep the collection sorted alphabetically by the manufacturer without writing a Comparator.
    - If we are not allowed to write a comparator, we can implement the comparable interface for the collection framework. This requires us to implement the `compareTo` function. Since the variable `manufacturer` is a string type, we can invoke the `compareTo` method on `String` class to compare the two string values

```
@Override
public int compareTo(Car o) {
    return this.manufacturer.compareTo(o.manufacturer);
}

Car[] arr = { car1, car2, car3, car4 };
ArrayList<Car> carList = new ArrayList<>(Arrays.asList(arr));

// Main.java
import java.util.Collections;
// Test 1:
Collections.sort(carList);
System.out.println(carList);
// [(a, 3), (b, 1), (e, 7), (e, 0)]
```

- (b) Using a Comparator, show how to keep the collection sorted by {manufacturer, age}. i.e. sort first by manufacturer, and sub-sort by age.
    - The `sort` method requires an customised comparator. We chaining comparators to compare multiple fields, first by manufacturer, then by age.

```
// Main.java
import java.util.Comparator;
...
// Test 2:
carList.sort(Comparator.comparing(Car::getManufacturer).thenComparing(
Car::getAge));
System.out.println(carList);
// [(a, 3), (b, 1), (e, 0), (e, 7)]
```

```
// Comments:
```

Q53. Write a Java program that reads in a text file that contains two integers on each line, separated by a comma (i.e. two columns in a comma-separated file). Your program should print out the same set of numbers, but sorted by the first column and subsorted by the second.

Code on Github

```
// Comments:
```

Q54. The following code captures errors using return values. Rewrite it to use exceptions.

```
public class RetValTest {
public static String sEmail = "";
public static int extractCamEmail(String sentence) {
if (sentence==null || sentence.length()==0)
return -1; // Error - sentence empty
String tokens[] = sentence.split(" "); // split into tokens
for (int i=0; i< tokens.length; i++) {
if (tokens[i].endsWith("@cam.ac.uk")) {
sEmail=tokens[i];
return 0; // success
}
}
return -2; // Error - no cam email found
}
public static void main(String[] args) {
int ret=RetValTest.extractCamEmail("My email is rkh23@cam.ac.uk");
if (ret==0) System.out.println("Success: "+RetValTest.sEmail);
else if (ret==-1) System.out.println("Supplied string empty");
else System.out.println("No @cam address in supplied string");
}
```

Question 54 on Github

Since the code is also relatively short, I also included it here for reference.

```
package Question54;

import java.util.NoSuchElementException;

public class RetValTest {
    public static String extractCamEmail(String sentence) {
        String sEmail = "";
        try {
            if (sentence == null || sentence.length() == 0)
                throw new Exception("Supplied string empty"); // Error -
sentence empty
            String tokens[] = sentence.split(" "); // split into tokens
            for (int i = 0; i < tokens.length; i++) {
                if (tokens[i].endsWith("@cam.ac.uk")) {
                    sEmail = tokens[i];// success
                }
            }
            if (sEmail == "") {
                throw new NoSuchElementException("Error - no cam email
found"); // Error - no cam email found
            }
        } catch (Exception e) {
            System.out.println(e);
        }
```

```
            return sEmail;
        }

        public static void main(String[] args) {
            String res = RetValTest.extractCamEmail("My email is
    rkh23@cam.ac.uk");
            System.out.println(res); // "rkh23@cam.ac.uk"
            res = RetValTest.extractCamEmail(""); // throw
    "java.lang.Exception: Supplied string empty"
            System.out.println(res); // ""
            res = RetValTest.extractCamEmail("aaa"); //
    java.util.NoSuchElementException: Error - no cam email found
            System.out.println(res); // ""
        }
    }
```

```
// Comments:
```

Q55. Write a Java function that computes the square root of a double number using the Newton-Raphson method. Your function should make appropriate use of exceptions

Question 55 on Github

```
// Comments:
```

Q56. Comment on the following implementation of pow, which computes the power of a number:

```
public class Answer extends Exception {
private int mAns;
public Answer(int a) { mAns=a; }
public int getAns() {return mAns;}
}
public class ExceptionTest {
private void powaux(int x, int v, int n) throws Answer {
if (n==0) throw new Answer(v);
else powaux(x,v*x,n-1);
}
public int pow(int x, int n) {
try { powaux(x,1,n); }
catch(Answer a) { return a.getAns(); }
return 0;
}
}
```

1. Not an approapriate use of Exceptions because Exceptions should be used for exceptional circumstances and Exception handler should only be used to handle errors, not like a `Go-to` statement.
2. This piece of code is hard to read and unclear because of the exceptions it used to get the correct answer.
3. We can simplify this piece of code by removing the exceptions and property handle cases. (i.e., Code on Github)

```
// Comments:
```

Q57. Give an example of how covariant arrays in Java can create runtime errors.

Question 57 on Github

```
public class Covariant {
    public static void main(String[] args) {
        Integer[] arrI = { 1, 2, 4, 9 };
        Number[] arrN = arrI;
        arrN[0] = 2.5f; // cause runtime error
    }
}
```

1. Arrays are covariant which means a function accept type A can also accept its subtype, according to the substitution principle, anywhere we want a type A can be replaced with its subtype.
2. In this case `Integer[]` is a subtype of `Number[]`, we can use a reference of type `Number[]` (i.e., `arrN`) to point to an integer array `arrI`.
3. We manipulate the number array `arrN` by change one of its element to a float, which is compile-safe as float is a subtype of `Number`.
4. At runtime, since `arrI` points to the same integer array and we change one of its element to a float, that cased a runtime error.

```
// Comments:
```

Q58. Compare Inner-classes, method-local classes, anonymous inner classes and lambda functions. What general advice would you give someone who is trying to choose which one to use?

1. Inner-classes and method-local classes can be used to create multiple instances; The anonymous inner class can only create one instance; Lambda functions are functions, they are not used for instantiation.
2. Static inner class can not access instance variables in the outer class, but can access static variables in the outer class; Instance inner class can access both instance or static variables of the outer class; Method-local class is never a static class because it can access the local variables of the method and static and instance variables in the class which contains the method.
3. The scope of the method-local classes is the function that includes it. The scope of the inner class is the outer class.
4. Lambda expression is used to implement functional interfaces which contains only one function. All other three classes can implement interfaces as well as inherits from other classes.

- General advice:
    - Inner class:
        - Need to create more than one instance of a class
        - Need to access its constructor or introduce additional methods.
        - Use static inner class if not required to access instance variables from the outer class.
        - Use instance inner class if required to access instance variables from the outer class.
    - Method local class:
        - A class that only need to be used in some particular method and nowhere else.
    - Annoymous class:
        - Need to create only one instance but with additional methods or fields.
    - Lambda expression:
        - Need a certain instance of a functional interface.

```
// Comments:
```

Q59. Complete the Alice in Wonderland task on Chime

Question 59(Task 1) on Chime

Question 59(Task 2) on Chime

```
// Comments:
```

Q60. Explain how Java uses the Decorator pattern with Reader.

1. The Decorator pattern aims at wrapping concrete objects with different concrete decorator and execute different behaviour at run-time.
2. `BufferedReader` is a concrete decorator; `InputStreamReader` is an adaptor between `BufferedReader` and `GZIPInputStream`; `GZIPInputStream` is a concrete decorator; ``FileInputStream` is the is the concrete component.
3. `BufferedReader` class has a field of type `Reader`, so it takes a `Reader` type into its constructor.
4. `BufferedReader` class and `InputStreamReader` class both inherits from `Reader` class, so an `inputStreamReader` object can be passed into the `BufferedReader` constructor.
5. `InputStreamReader` class follows an adaptor pattern where it acts as a bridge between byte streams and character streams so that the character streams can be processed later by the `readLine()` method in the `BufferedReader` class.
6. `InputStreamReader` class takes an `InputStream` type Object in its constructor. While both `GZIPInputStream` class and `FileInputStream` class inherits from `InputStream` class, so by substitution principle, an `GZIPInputStream` object can be passed to the constructor of `InputStreamReader` class.
7. `GZIPInputStream` class takes `inputStream` as its parameter in the constructor, so an object with type `FileInputStream` can be passed into it.
8. The `File` book is first implemented concretly using the `FileInputStream`, then wrapped with `GZIPInputStream`, adapted with `InputStreamReader`, then wrapped with `BufferedReader`.

9. At run-time, while the concrete object called `readLine()` method, the adaptor called the `read()` method in the `GZIPInputStream` class before executing its self-implementation, `GZIPInputStream` class called the `read()` method in the `FileInputStream` class before executing its self-implementation. The concrete component will be treated as different objects at each stage.
10. So the file will be read as bytes, then compressed, then convert into characters, then read line by line.
11. I do have a question about how the adapator `InputStreamReader` class can invoke `read()` method, because `BufferedReader` object calls the `readLine()` method and there is no `readLine()` method in the `InputStreamReader` class...

```
public static void main(String[] args) throws IOException {
    File book = downloadBook();

    // Demonstrate decorator and adaptor pattern
    try (BufferedReader is =
        new BufferedReader(  // decorator
            new InputStreamReader( // adapter
                new GZIPInputStream( // decorator
                        new FileInputStream(book) // concrete
implementation
                ), StandardCharsets.UTF_8))) {
      String line;
      while ((line = is.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
```

```
// Comments:
```

Q61. In lectures the examples for the State pattern looked at implementing a Fan with different speeds. Compare the different approaches taken. What were the advantages and disadvantages with them? In what sense is the final solution demonstrating the open-closed principle?

1. `FanSpeedWithInts`:
    1. Advantage:
        1. Only one class is included and we can get an overview of the number of states involve as well as the relationship between states.

    2. Disadvantage:
        1. If we want to add more fan speeds, we have to modify each methods linked with states of fan speed (i.e., update & click)
        2. We have to assign each state both a name and an integer, but we never used the integer at all in all the methods.
2. `FanSpeedWithEnum`:
    1. Advantage:
        1. Getting rid of the integers so only represent each state with a enumeration type.
        2. Similarly, we can get an overview of the states involved in the finite state machine and how they are linked with each other.
    2. Disadvantage:
        1. Bulky conditional statements if we have multiple states included
        2. Adding new states involves changing each related methods, which is quite cumbersome and may cause inconsistency in the codes if we forget to modify one of the methods.
3. `FanSpeedWithStatePattern`:
    1. Advantage:
        1. Abbey the open-closed principle. The open-closed principle states that class should open for extension but closed for modification.
            1. If we want to add more states, there is no need to modify the original source code
            2. we can simply add another class which implements the shared interface and override the common methods to provide a different method implementation.
        2. The method in the source code will be very light because the actuaal object will be determined at run time.
        3. Single responsibility principle: since each state will be stored in a separate file, each is responsible for a specific task.
        4. It gets rid of the bulky state machine conditions in the main method.
    2. Disadvantage:
        1. Each new state requires a separate class, so more files are included.
        2. Apply the pattern can be overkill if a state machine has only a few states or rarely changes.

```
// Comments:
```

Q62. Complete the Game of Life task on Chime

Task 1 Code on Chime

Task 2 Code on Chime

Task 3 Code on Chime

Task 4 Code on Chime

Task 5 Code on Chime

```
// Comments:
```

Q63. Explain the difference between the State pattern and the Strategy pattern

1. State pattern is an extension of the strategy pattern as both patterns are based on composition. They change the behaviour of the context by delegating some work to helper objects.
2. In the state pattern, the particular states may be aware of each other and initiate transitions from one state to another, whereas in the strategy pattern, the strategies do not know each other.

```
// Comments:
```

Q64. A drawing program has an abstract Shape class. Each Shape object supports a draw() method that draws the relevant shape on the screen (as per the example in lectures). There are a series of concrete subclasses of Shape, including Circle and Rectangle. The drawing program keeps a list of all shapes in a List object.

(a) Should draw() be an abstract method?

- Yes. Although the abstract class `Shape` can have both abstract and concrete methods, the `Shape` class can not construct an instance. There is no point to draw a general shape, we always want to draw some particular shape type (i.e., circles, points), so it is better to leave the method abstract and let the subclasses of `Shape` to implement it.

(b) Write Java code for the function in the main application that draws all the shapes on each screen refresh.

Question (b) on Github

```
// Question (b)
public static void main(String[] args) throws InterruptedException {
        Shape circle1 = new Circle();
        Shape rect1 = new Rectangle();
        Shape[] arr = { circle1, rect1 };
        ArrayList<Shape> shapeList = new ArrayList<>(Arrays.asList(arr));

        System.out.println("For question B:");

        int cnt = 0;
        while (cnt < 5) {
            System.out.println("Time:" + cnt);
            for (Shape item : shapeList) {
                item.draw();
            }
            TimeUnit.SECONDS.sleep(1);
            cnt++;
        }
}
```

(c) Show how to use the Composite pattern to allow sets of shapes to be grouped together and treated as a single entity.

Question (c) on Github

```
// Main.java

public static void main(String[] args) throws InterruptedException {
    ...
// Question (c)
        Composite set1 = new Composite(shapeList);

        Shape circle2 = new Circle();
        Shape circle3 = new Rectangle();
        Composite set2 = new Composite(circle2, circle3);

        Composite set3 = new Composite();
        set3.addComponent(set1);
        set3.addComponent(set2);

        System.out.println("For question C:");
        set3.draw();
}

// Composite.java
public class Composite extends Shape {
    private ArrayList<Shape> component;

    Composite(ArrayList<? extends Shape> component) {
        this.component = (ArrayList<Shape>) component;
```

```
    }

    Composite(Shape... component) {
        this.component = new ArrayList<Shape>(Arrays.asList(component));
    }

    Composite() {
        this.component = new ArrayList<Shape>();
    }

    public void addComponent(Shape component) {
        this.component.add(component);
    }

    public void removeComponent(Shape component) {
        this.component.remove(component);
    }

    @Override
    public void draw() {
        for (Shape item : component) {
            item.draw();
        }
    }

}
```

(d) Which design pattern would you use if you wanted to extend the program to draw frames around some of the shapes? Show how this would work.

Question (d) on Github

1. I would use a Decorator patter to extend the behaviours because we want to add some extra behaviour on top of the existing ones at run-time without breaking the code.
2. First create a `Decorator` class which inherits the `Shape` abstract class.

```
    public class Decorator extends Shape {

    protected Shape wrapee;

    Decorator(Shape wrapee) {
        this.wrapee = wrapee;
    }

    @Override
    public void draw() {
        wrapee.draw();
    }

}
```

3. Then implement the various decorators, i.e., `Frame` class and `RoundFrame` class.

```
public class Frame extends Decorator {

    Frame(Shape wrapee) {
        super(wrapee);
    }

    @Override
    public void draw() {
        wrapee.draw();
        System.out.println("A frame");
    }

}

public class RoundFrame extends Decorator {

    RoundFrame(Shape wrapee) {
        super(wrapee);
    }

    @Override
    public void draw() {
        wrapee.draw();
        System.out.println("Draw a round frame.");
    }

}
```

4. Wrapped the concrete objects into the concrete decorators.

```
// Question (d)
    public static void main(String[] args){
        System.out.println("For question D:");
        Shape shape1 = new Frame(new Circle());
        Shape shape2 = new RoundFrame(new Rectangle());
        shape1.draw();
        shape2.draw();
    }
```

5. Thus the objects can be treated as concrete decorators at runtime and execute the approapriate behaviour.

```
For question D:
Draw a circle.
```

```
A frame
Draw a rectangle.
Draw a round frame.
```

```
// Comments:
```

65. What would you say are the three most important concepts from the content for this supervision?

1. The difference between various collections in the collection frame. Because we will utilise them a lot and it is crucial to understand the inheriteance hierachy and their properties.
2. The new features introduced in Java 8, particularly Lambda expression, streams and method reference.
3. The structure of different design patterns and what problems do each of them solve.

66. Give one question that you would like to discuss in the supervision.

1. Problem related with question 60:

   > I do have a question about how the adapator `InputStreamReader` class can invoke `read()` method, because `BufferedReader` object calls the `readLine()` method and there is no `readLine()` method in the `InputStreamReader` class...

2. The difference between default method in interfaces and concrete methods in abstract class. Is it because default method in interface can not have any interactions with variables since all variables in interface are final?