Student Name: Sean Welch

Student number: x23285508

Course Code: HDSDEV_JAN24

// Question 1:

```
# Question 1:



### A. Explain in detail, with examples, the concept of Multithreading. (5 Marks)

* Multithreading is a concept in software engineering that refers to the execution of multiple threads in parallel to each other; whereby a thread represents a lightweight subprocess in execution [1].

* Multithreading is distinct from multiprocessing because threads have a shared memory area and context-switching takes less time than changing between processes [1].

* Multithreading allows for non-blocking operations to occur thus enhancing performance for the user as multiple operations can occur at the same time [1].

* Threads are often independent meaning that if an exception is thrown in one thread, it won't affect the execution of other threads [1].

* In Java, we can aggregate threads into a thread pool, which is a collection of workers that wait to process jobs in the queue which can simplify the process of managing thread execution [1].



--------------------------------------------------------------



### B. Discuss the type of problems, with examples, where multithreading fails to improve performance. (5 Marks)

* On a single processor server, it does not make sense to use multithreading as we do not have multiple processors/cores to make use of, therefore context switching will incur reduced performance due to necessity of sequential operations. [2].

* When threads need access to a shared resource, such as a database or shared data structure, often time multithreading does not make sense as it can lead to locking situations to avoid data corruption [2].
```

* If we are dealing with a small amount of data to process, it does not make sense to use multithreading as the overhead of thread management will likely not outweigh the performance benefits of multithreading [2].

* Any task that requires sequential execution, such as a sorting algorithm whereby the next operation is dependent on the result of its predecessor, will not benefit from multithreading [2].

* Any process that involves a lot of I/O such as a large amount of reading/writing to a file will not benefit from multithreading as the bottleneck is the I/O, not the computation itself [2].

### C. Discuss the difference between threads that extend the Thread class and threads that implement the Runnable interface. (5 Marks)

* When extending the thread class, we are not overriding any of its methods, instead we override the method of Runnable, which happens to be implemented by Thread, which is a violation of IS-A principle [3].

* Implementing the Runnable interface and passing it to the thread class is an example of composition of inheritance and is therefore more flexible [3].

* We can't extend any other classes if using Thread due to Single inheritance in Java [3].

* Runnable is often considered better design because it separates thread behavior from thread execution [3].

* Access to more methods if implementing the Runnable interface as we can still implement other interfaces [3].

# References

[1] “Multithreading in java,” www.javatpoint.com. [Online]. Available: https://www.javatpoint.com/multithreading-in-java. [Accessed: 21-Aug-2024].

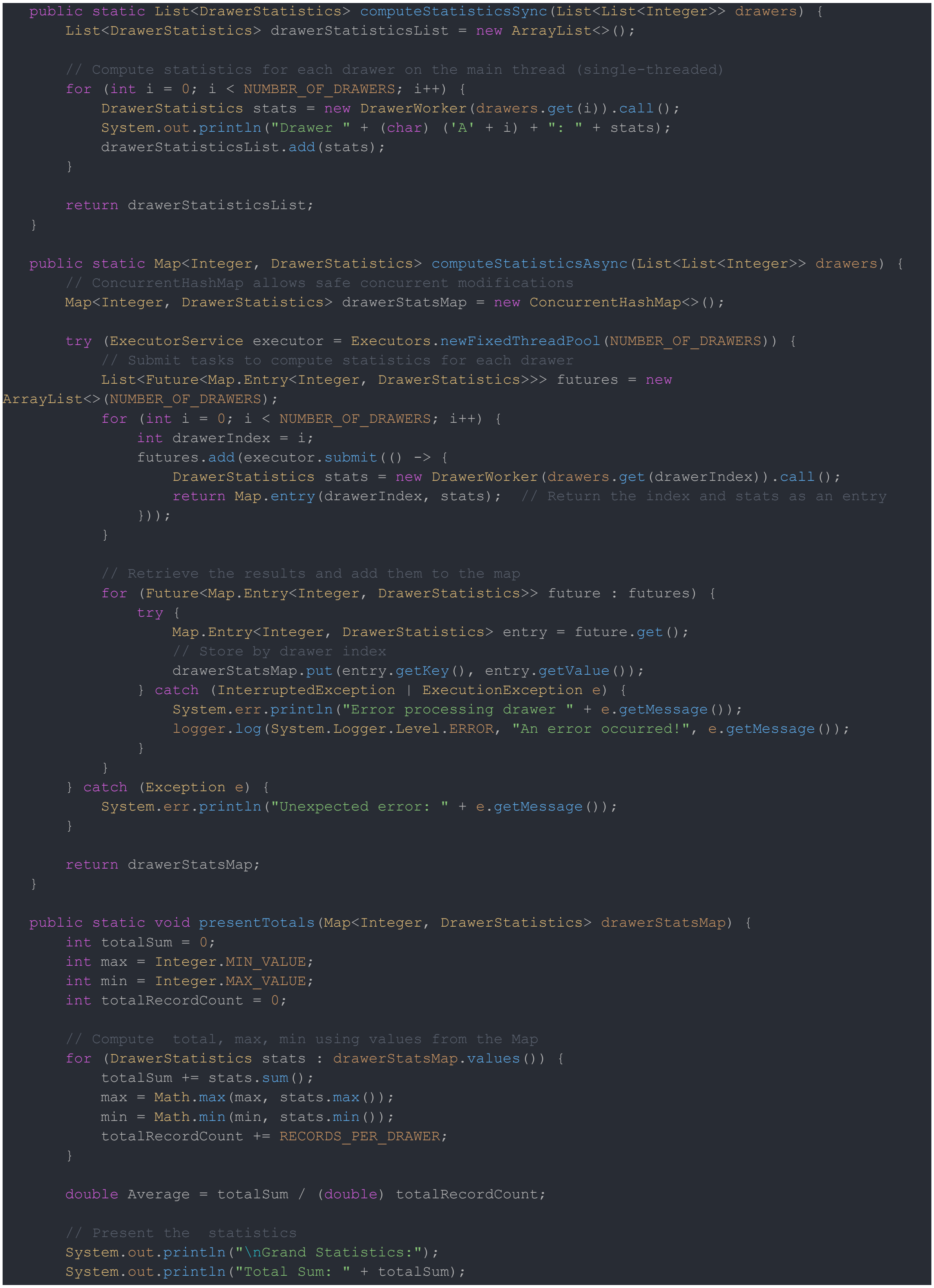
[2] “When is multi-threading not a good idea?,” Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/93834/when-is-multi-threading-not-a-good-idea. [Accessed: 22-Aug-2024].

[3] “Implementing a Runnable vs Extending a Thread,” Baeldung.com. [Online]. Available: https://www.baeldung.com/java-runnable-vs-extending-thread. [Accessed: 22-Aug-2024].

// Question 2

```
package taba.question2;

import java.util.List;
import java.util.concurrent.Callable;

public record DrawerStatistics(int sum, double average, int max, int min) {
   @Override
   public String toString() {
       return "Sum: " + sum + ", Average: " + average + ", Max: " + max + ", Min: " + min;
   }
}

class DrawerWorker implements Callable<DrawerStatistics> {
   private final List<Integer> drawerData;

   public DrawerWorker(List<Integer> drawerData) {
       this.drawerData = drawerData;
   }

   @Override
   public DrawerStatistics call() {
       int sum = 0;
       int max = Integer.MIN_VALUE;
       int min = Integer.MAX_VALUE;

       for (int num : drawerData) {
           sum += num;
           if (num > max) max = num;
           if (num < min) min = num;
       }

       double average = sum / (double) drawerData.size();
       return new DrawerStatistics(sum, average, max, min);
   }
}
```

```
package taba.question2;

import java.util.*;
import java.util.concurrent.*;

public class Question2 {
   private static final int NUMBER_OF_DRAWERS = 10; // Constant for the number of drawers
   private static final int RECORDS_PER_DRAWER = 10000; // Constant for records per drawer
   private static final System.Logger logger = System.getLogger(Question2.class.getName());


   public static List<List<Integer>> generateData() {
       Random randomObject = new Random();
       List<List<Integer>> drawers = Collections.synchronizedList(new ArrayList<>(NUMBER_OF_DRAWERS));

       for (int iCount = 0; iCount < NUMBER_OF_DRAWERS; iCount++) {
           // Generates 10,000 random integer numbers between -10,000 and 10,000
           List<Integer> drawer = new ArrayList<>();
           randomObject.ints(RECORDS_PER_DRAWER, -10000, 10000).forEach(drawer::add);
           drawers.add(drawer);
       }
       return drawers;
   }

```



```
    public static List<DrawerStatistics> computeStatisticsSync(List<List<Integer>> drawers) {
        List<DrawerStatistics> drawerStatisticsList = new ArrayList<>();

        // Compute statistics for each drawer on the main thread (single-threaded)
        for (int i = 0; i < NUMBER_OF_DRAWERS; i++) {
            DrawerStatistics stats = new DrawerWorker(drawers.get(i)).call();
            System.out.println("Drawer " + (char) ('A' + i) + ": " + stats);
            drawerStatisticsList.add(stats);
        }

        return drawerStatisticsList;
    }

    public static Map<Integer, DrawerStatistics> computeStatisticsAsync(List<List<Integer>> drawers) {
        // ConcurrentHashMap allows safe concurrent modifications
        Map<Integer, DrawerStatistics> drawerStatsMap = new ConcurrentHashMap<>();

        try (ExecutorService executor = Executors.newFixedThreadPool(NUMBER_OF_DRAWERS)) {
            // Submit tasks to compute statistics for each drawer
            List<Future<Map.Entry<Integer, DrawerStatistics>>> futures = new
ArrayList<>(NUMBER_OF_DRAWERS);
            for (int i = 0; i < NUMBER_OF_DRAWERS; i++) {
                int drawerIndex = i;
                futures.add(executor.submit(() -> {
                    DrawerStatistics stats = new DrawerWorker(drawers.get(drawerIndex)).call();
                    return Map.entry(drawerIndex, stats);  // Return the index and stats as an entry
                }));
            }

            // Retrieve the results and add them to the map
            for (Future<Map.Entry<Integer, DrawerStatistics>> future : futures) {
                try {
                    Map.Entry<Integer, DrawerStatistics> entry = future.get();
                    // Store by drawer index
                    drawerStatsMap.put(entry.getKey(), entry.getValue());
                } catch (InterruptedException | ExecutionException e) {
                    System.err.println("Error processing drawer " + e.getMessage());
                    logger.log(System.Logger.Level.ERROR, "An error occurred!", e.getMessage());
                }
            }
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
        }

        return drawerStatsMap;
    }

    public static void presentTotals(Map<Integer, DrawerStatistics> drawerStatsMap) {
        int totalSum = 0;
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        int totalRecordCount = 0;

        // Compute  total, max, min using values from the Map
        for (DrawerStatistics stats : drawerStatsMap.values()) {
            totalSum += stats.sum();
            max = Math.max(max, stats.max());
            min = Math.min(min, stats.min());
            totalRecordCount += RECORDS_PER_DRAWER;
        }

        double Average = totalSum / (double) totalRecordCount;

        // Present the  statistics
        System.out.println("\nGrand Statistics:");
        System.out.println("Total Sum: " + totalSum);
        System.out.println("Average: " + Average);
        System.out.println("Max: " + max);
        System.out.println("Min: " + min);
    }

    public static void main(String[] args) {
        // Generate data for 10 drawers
        List<List<Integer>> drawers = generateData();

        // Compute statistics for each drawer
        Map<Integer, DrawerStatistics> drawerStats = computeStatisticsAsync(drawers);

        // Compute and present total statistics
        presentTotals(drawerStats);
    }
}
```

```
package taba.question2;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

import java.util.List;
import java.util.Map;

class Question2Test {

    @Test
    void compareSingleThreadedAndMultiThreadedPerformance() {
        // Generate data for 10 drawers
        List<List<Integer>> drawers = Question2.generateData();

        // Measure time for async execution
        long asyncStartTime = System.nanoTime();
        Map<Integer, DrawerStatistics> asyncResults =
Question2.computeStatisticsAsync(drawers);
        long asyncEndTime = System.nanoTime();
        long asyncDuration = asyncEndTime - asyncStartTime;

        // Measure time for synchronous execution
        long synchronousStartTime = System.nanoTime();
        List<DrawerStatistics> synchronousResults = Question2.computeStatisticsSync(drawers);
        long synchronousEndTime = System.nanoTime();
        long synchronousDuration = synchronousEndTime - synchronousStartTime;

        // Ensure both methods return the same results
        assertEquals(synchronousResults.size(), asyncResults.size(), "Both methods should
produce same number of results");

        // Print results to console
        System.out.println("\nTime taken by async version: " + asyncDuration + " ns");
        System.out.println("Time taken by synchronous version: " + synchronousDuration + "
ns");
        System.out.println("\nEfficiency comparison:");
        System.out.println("Async execution is " + (synchronousDuration / (double)
asyncDuration) + " times faster");

        assertTrue(asyncDuration < synchronousDuration, "Async execution should be faster");
```

~~~~
    }
}
~~~~

~~~~
# C. Explain in writing the efficiency of your program by comparing it with a single-threaded program.
Use appropriate references where necessary. (5 Marks)

## Explanation

####
* The code for part a uses multithreading in order to achieve a more efficient processing speed.
* Multithreading refers to the process of using multiple cores on a CPU in order to run tasks in
parallel to each other.
* To do this, I have implemented the use of an Executor Service to manage the threads which automates
the submitting of tasks to the queue.
* We Then pool together these threads in the thread pool to avoid creating a new thread every time a
task is to be executed in the queue.
* The future tasks are implemented as a callable object which are added to a map along with an index to
represent each statistic or their future asynchronous result.
* A concurrent hashmap is then used to store the computed statistics in a thread safe manner. This was
crucial to increasing performance as I'm working on an 8 Core M2 Macbook Air, with a programme running
10 threads of execution

* In the single threaded version of the method, we are simply adding each future result to the
statistics array in a loop.
* The code below and test results show the outcome of both operations, with the results showing that
the async approach is on average 1.5x faster.
####

---------------------------------------------------------------


## Test Code

---------------------------------------------------------------


```java
package taba.question2;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

import java.util.List;
import java.util.Map;

class Question2Test {

   @Test
   void compareSingleThreadedAndMultiThreadedPerformance() {
       // Generate data for 10 drawers
       List<List<Integer>> drawers = Question2.generateData();

       // Measure time for async execution
       long asyncStartTime = System.nanoTime();
       Map<Integer, DrawerStatistics> asyncResults = Question2.computeStatisticsAsync(drawers);
       long asyncEndTime = System.nanoTime();
       long asyncDuration = asyncEndTime - asyncStartTime;

       // Measure time for synchronous execution
       long synchronousStartTime = System.nanoTime();
       List<DrawerStatistics> synchronousResults = Question2.computeStatisticsSync(drawers);
       long synchronousEndTime = System.nanoTime();
       long synchronousDuration = synchronousEndTime - synchronousStartTime;
~~~~

~~~
        // Ensure both methods return the same results
        assertEquals(synchronousResults.size(), asyncResults.size(), "Both methods should produce same
number of results");

        // Print results to console
        System.out.println("\nTime taken by async version: " + asyncDuration + " ns");
        System.out.println("Time taken by synchronous version: " + synchronousDuration + " ns");
        System.out.println("\nEfficiency comparison:");
        System.out.println("Async execution is " + (synchronousDuration / (double) asyncDuration) + "
times faster");

        assertTrue(asyncDuration < synchronousDuration, "Async execution should be faster");
    }
}
```

# Test Code Results

### Result 1

--------------------------------------------------------------
* Drawer A: Sum: -1403, Average: -0.1403, Max: 9999, Min: -9995
* Drawer B: Sum: 100811, Average: 10.0811, Max: 9998, Min: -9999
* Drawer C: Sum: -527075, Average: -52.7075, Max: 9998, Min: -9998
* Drawer D: Sum: 131911, Average: 13.1911, Max: 9997, Min: -10000
* Drawer E: Sum: -749405, Average: -74.9405, Max: 9996, Min: -10000
* Drawer F: Sum: -43741, Average: -4.3741, Max: 9991, Min: -10000
* Drawer G: Sum: -579892, Average: -57.9892, Max: 9998, Min: -10000
* Drawer H: Sum: -649010, Average: -64.901, Max: 9999, Min: -9999
* Drawer I: Sum: -277948, Average: -27.7948, Max: 9998, Min: -10000
* Drawer J: Sum: -407252, Average: -40.7252, Max: 9995, Min: -10000

Time taken by async version: 5135959 ns
Time taken by synchronous version: 7517542 ns

Efficiency comparison:
Async execution is 1.4637075568554967 times faster

### Result 2

--------------------------------------------------------------
* Drawer A: Sum: -412660, Average: -41.266, Max: 9997, Min: -10000
* Drawer B: Sum: 160125, Average: 16.0125, Max: 9998, Min: -9999
* Drawer C: Sum: -793994, Average: -79.3994, Max: 9999, Min: -9999
* Drawer D: Sum: 894926, Average: 89.4926, Max: 9998, Min: -10000
* Drawer E: Sum: -517631, Average: -51.7631, Max: 9992, Min: -10000
* Drawer F: Sum: 407670, Average: 40.767, Max: 9999, Min: -9999
* Drawer G: Sum: -907340, Average: -90.734, Max: 9998, Min: -9998
* Drawer H: Sum: 21972, Average: 2.1972, Max: 9998, Min: -10000
* Drawer I: Sum: 646527, Average: 64.6527, Max: 9998, Min: -10000
* Drawer J: Sum: -229551, Average: -22.9551, Max: 9989, Min: -10000

Time taken by async version: 4928791 ns
Time taken by synchronous version: 7401625 ns

Efficiency comparison:
Async execution is 1.5017120831457451 times faster

### Result 3

--------------------------------------------------------------
* Drawer A: Sum: -755120, Average: -75.512, Max: 9999, Min: -10000
* Drawer B: Sum: -351901, Average: -35.1901, Max: 9996, Min: -9998
* Drawer C: Sum: -120262, Average: -12.0262, Max: 9993, Min: -9995
* Drawer D: Sum: -199275, Average: -19.9275, Max: 9996, Min: -9999
* Drawer E: Sum: 250066, Average: 25.0066, Max: 9998, Min: -9996
~~~

```
* Drawer F: Sum: -278982, Average: -27.8982, Max: 9999, Min: -9999
* Drawer G: Sum: -24974, Average: -2.4974, Max: 9999, Min: -9999
* Drawer H: Sum: 465794, Average: 46.5794, Max: 9994, Min: -9999
* Drawer I: Sum: -1159984, Average: -115.9984, Max: 9999, Min: -9999
* Drawer J: Sum: 548317, Average: 54.8317, Max: 9997, Min: -10000

Time taken by async version: 4991042 ns
Time taken by synchronous version: 10109292 ns

Efficiency comparison:
Async execution is 2.0254872629803558 times faster
--------------------------------------------------------------
```

// Question 3:

```
package taba.question3;

import java.io.*;

public class ManageProducts {
   private static final System.Logger logger =
System.getLogger(ManageProducts.class.getName());

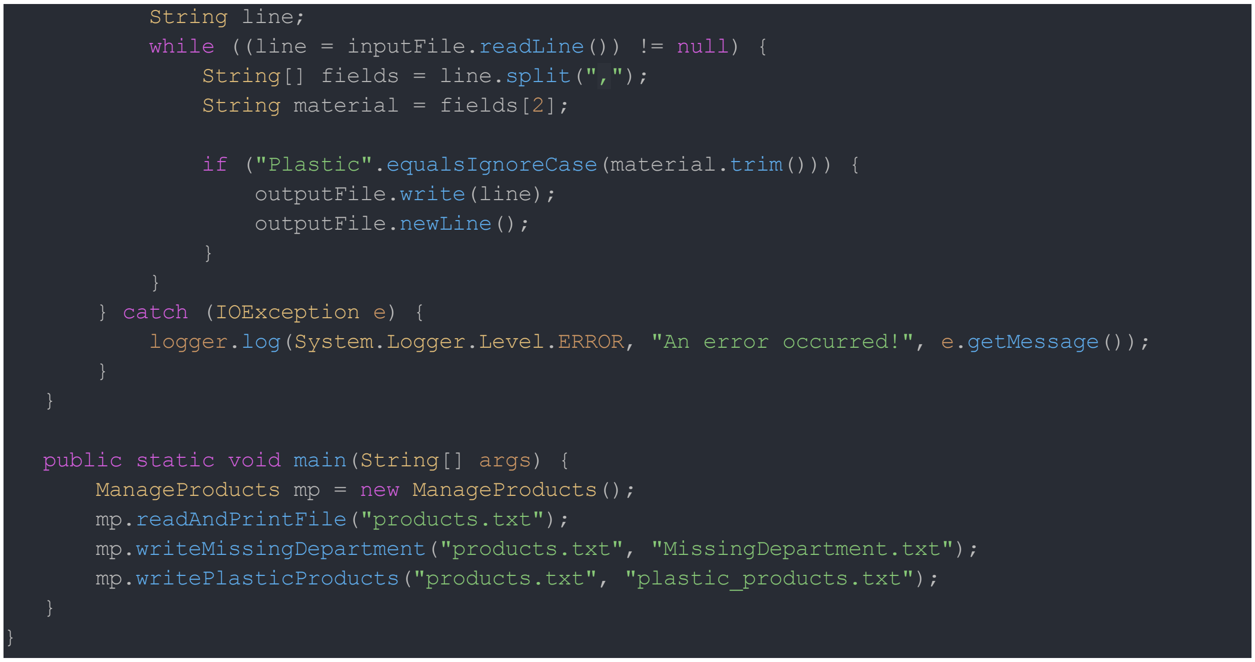
   // Part A
   public void readAndPrintFile(String filePath) {
       try (BufferedReader inputFile = new BufferedReader(new FileReader(filePath))) {
           String line;
           while ((line = inputFile.readLine()) != null) {
               System.out.println(line);
           }
       } catch (IOException e) {
           logger.log(System.Logger.Level.ERROR, "An error occurred!", e.getMessage());
       }
   }

   // Part B
   public void writeMissingDepartment(String inputFilePath, String outputFilePath) {
       try (BufferedReader inputFile = new BufferedReader(new FileReader(inputFilePath));
            BufferedWriter outputFile = new BufferedWriter(new FileWriter(outputFilePath))) {

           String line;
           while ((line = inputFile.readLine()) != null) {
               String[] fields = line.split(",");
               String department = fields[3];

               if (department == null || department.trim().isEmpty()) {
                   outputFile.write(line);
                   outputFile.newLine();
               }
           }
       } catch (IOException e) {
           logger.log(System.Logger.Level.ERROR, "An error occurred!", e.getMessage());
       }
   }

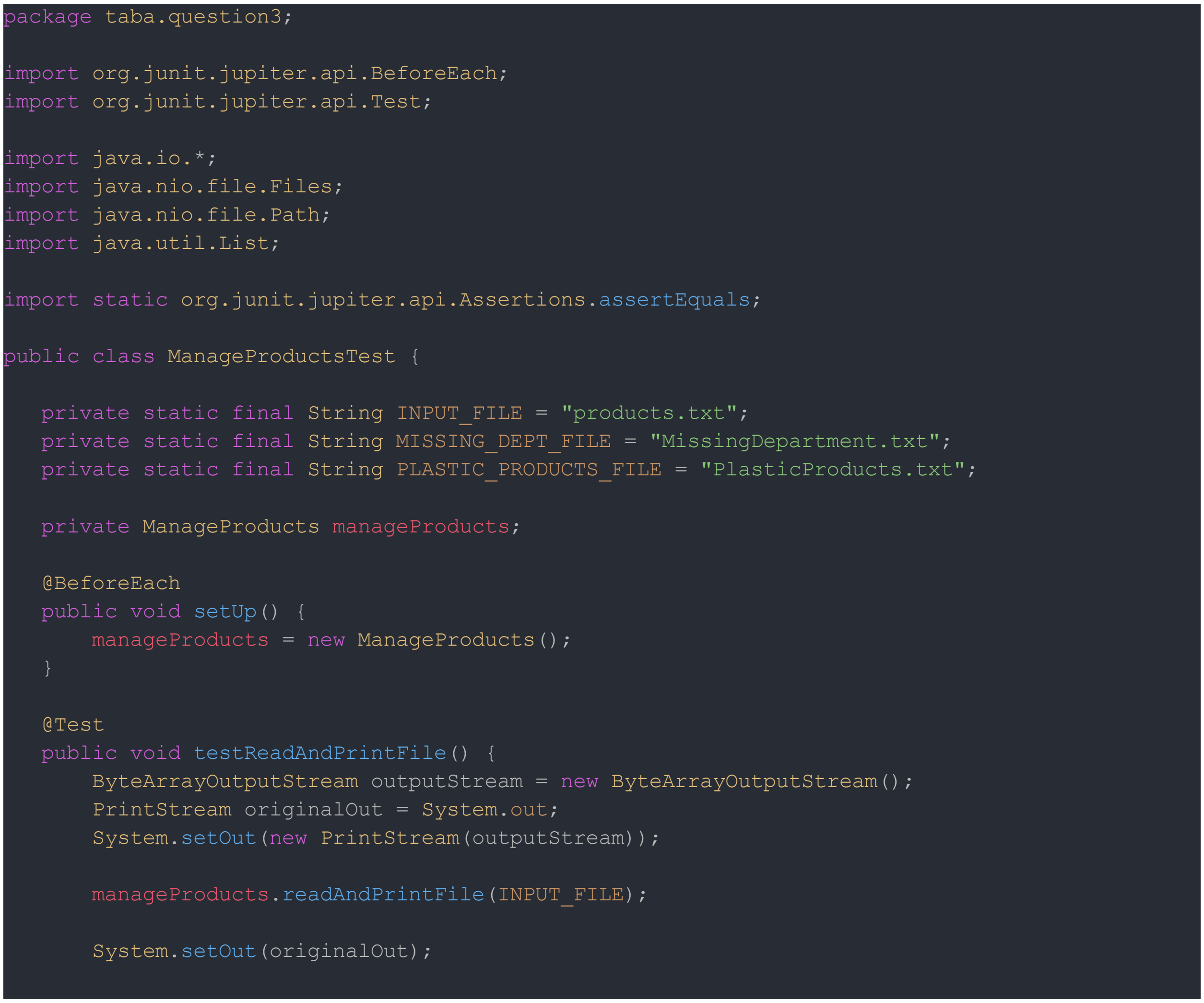
   // Part D
   public void writePlasticProducts(String inputFilePath, String outputFilePath) {
       try (BufferedReader inputFile = new BufferedReader(new FileReader(inputFilePath));
            BufferedWriter outputFile = new BufferedWriter(new FileWriter(outputFilePath))) {
```



```java
            String line;
            while ((line = inputFile.readLine()) != null) {
                String[] fields = line.split(",");
                String material = fields[2];

                if ("Plastic".equalsIgnoreCase(material.trim())) {
                    outputFile.write(line);
                    outputFile.newLine();
                }
            }
        } catch (IOException e) {
            logger.log(System.Logger.Level.ERROR, "An error occurred!", e.getMessage());
        }
    }

    public static void main(String[] args) {
        ManageProducts mp = new ManageProducts();
        mp.readAndPrintFile("products.txt");
        mp.writeMissingDepartment("products.txt", "MissingDepartment.txt");
        mp.writePlasticProducts("products.txt", "plastic_products.txt");
    }
}
```



```java
package taba.question3;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class ManageProductsTest {

    private static final String INPUT_FILE = "products.txt";
    private static final String MISSING_DEPT_FILE = "MissingDepartment.txt";
    private static final String PLASTIC_PRODUCTS_FILE = "PlasticProducts.txt";

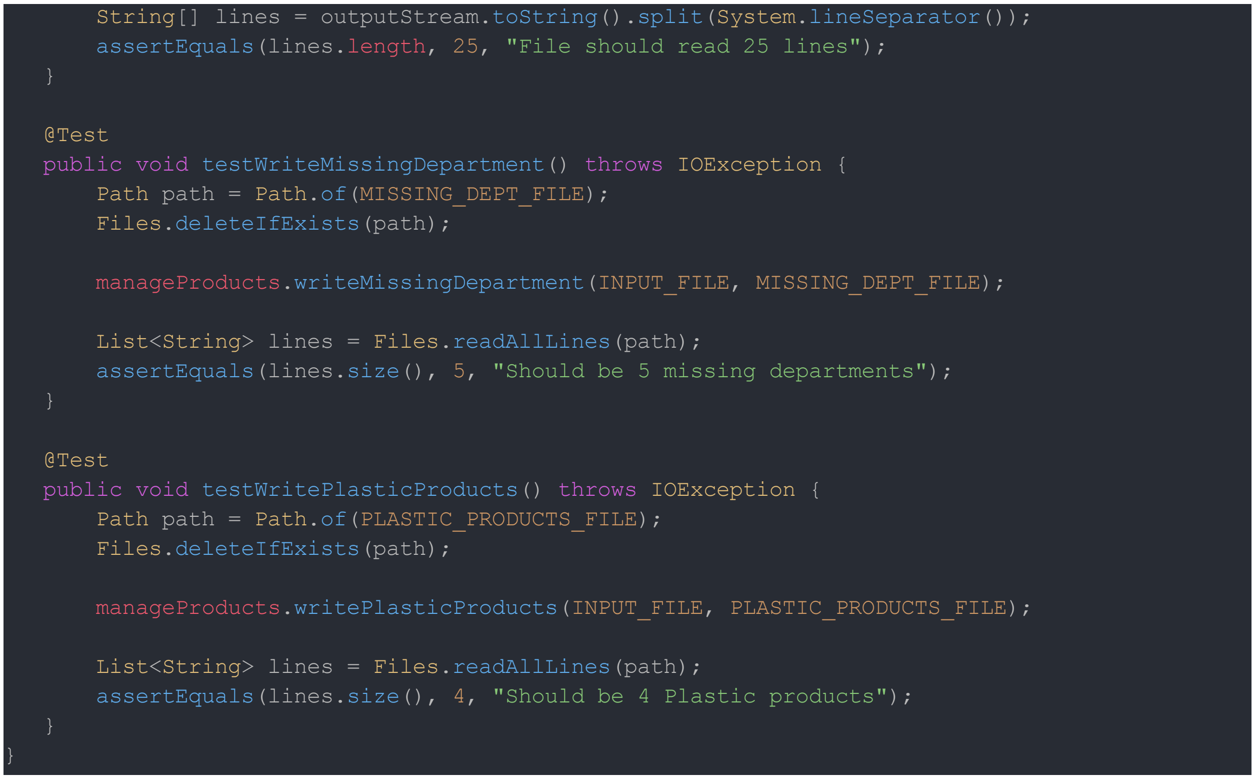
    private ManageProducts manageProducts;

    @BeforeEach
    public void setUp() {
        manageProducts = new ManageProducts();
    }

    @Test
    public void testReadAndPrintFile() {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        PrintStream originalOut = System.out;
        System.setOut(new PrintStream(outputStream));

        manageProducts.readAndPrintFile(INPUT_FILE);

        System.setOut(originalOut);
```



```
        String[] lines = outputStream.toString().split(System.lineSeparator());
        assertEquals(lines.length, 25, "File should read 25 lines");
    }

    @Test
    public void testWriteMissingDepartment() throws IOException {
        Path path = Path.of(MISSING_DEPT_FILE);
        Files.deleteIfExists(path);

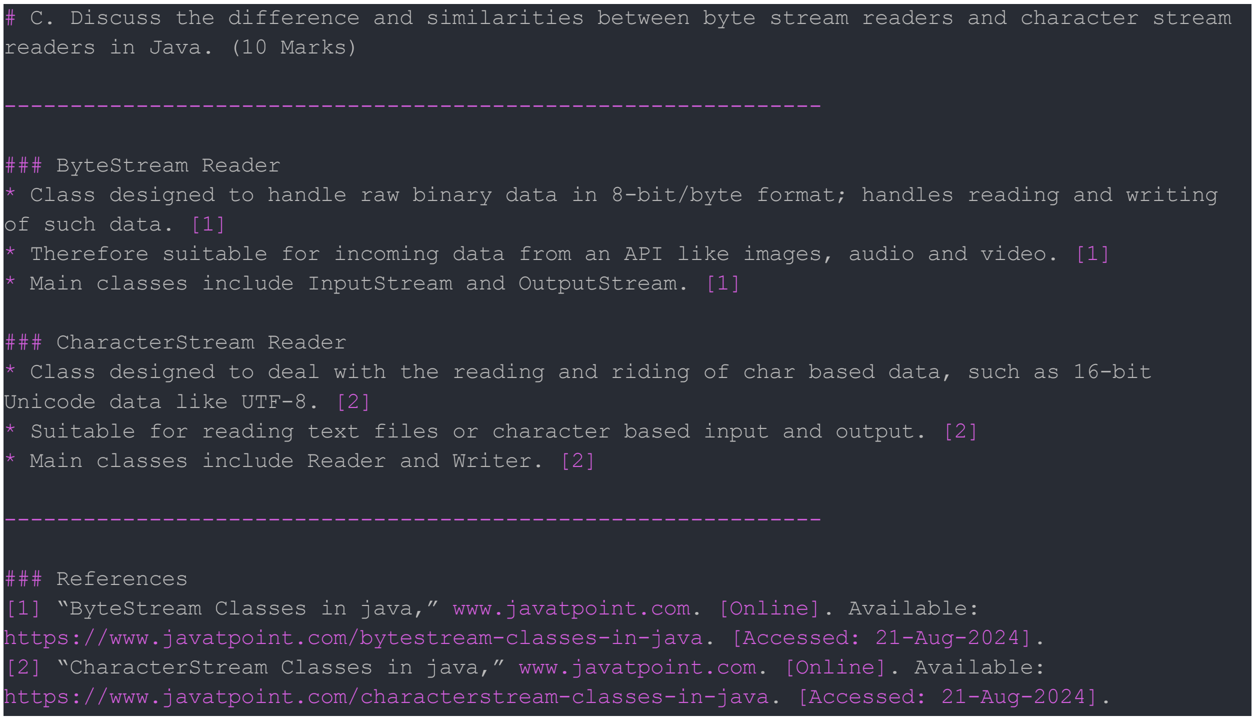
        manageProducts.writeMissingDepartment(INPUT_FILE, MISSING_DEPT_FILE);

        List<String> lines = Files.readAllLines(path);
        assertEquals(lines.size(), 5, "Should be 5 missing departments");
    }

    @Test
    public void testWritePlasticProducts() throws IOException {
        Path path = Path.of(PLASTIC_PRODUCTS_FILE);
        Files.deleteIfExists(path);

        manageProducts.writePlasticProducts(INPUT_FILE, PLASTIC_PRODUCTS_FILE);

        List<String> lines = Files.readAllLines(path);
        assertEquals(lines.size(), 4, "Should be 4 Plastic products");
    }
}
```



```
# C. Discuss the difference and similarities between byte stream readers and character stream
readers in Java. (10 Marks)

--------------------------------------------------------------

### ByteStream Reader
* Class designed to handle raw binary data in 8-bit/byte format; handles reading and writing
of such data. [1]
* Therefore suitable for incoming data from an API like images, audio and video. [1]
* Main classes include InputStream and OutputStream. [1]

### CharacterStream Reader
* Class designed to deal with the reading and riding of char based data, such as 16-bit
Unicode data like UTF-8. [2]
* Suitable for reading text files or character based input and output. [2]
* Main classes include Reader and Writer. [2]

--------------------------------------------------------------

### References
[1] “ByteStream Classes in java,” www.javatpoint.com. [Online]. Available:
https://www.javatpoint.com/bytestream-classes-in-java. [Accessed: 21-Aug-2024].
[2] “CharacterStream Classes in java,” www.javatpoint.com. [Online]. Available:
https://www.javatpoint.com/characterstream-classes-in-java. [Accessed: 21-Aug-2024].
```