IY4113 Milestone 3

Assessment Details	Please Complete All Details
Group	B
Module Title	Object Orientated programming
Assessment Type	Coursework
Module Tutor Name	Jonathon Shore
Student ID Number	P508377
Date of Submission	08/02/26
Word Count	1087

- ✓ *I confirm that this assignment is my own work. Where I have referred to academic sources, I have provided in-text citations and included the sources in the final reference list.*
- ✓ *Where I have used AI, I have cited and referenced appropriately.

Research (minimum of 2, at least 3)

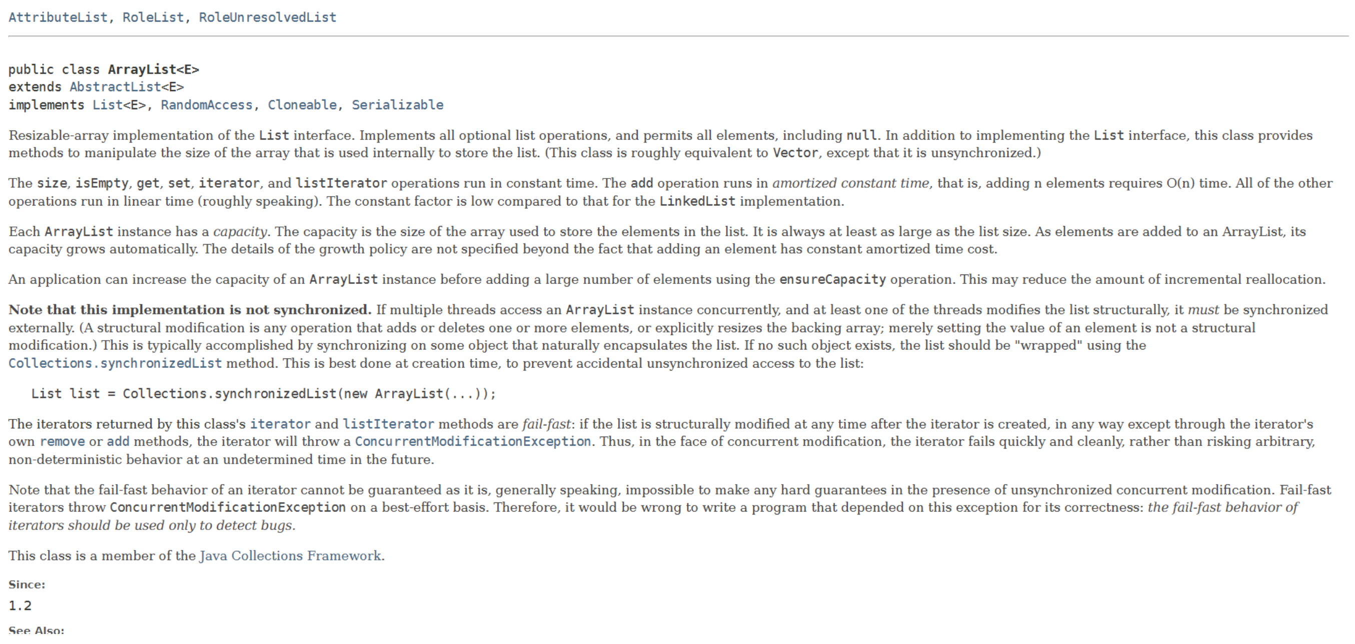
Conduct research to support your coding process, including use of code examples, tutortials, documentation and AI tools (if used). Use the structure below to capture your evidence:

Title of research: Using Arraylist to Store Objects Reference (link): https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html How does the research help with coding practise?: This research helped me understand how to store multiple journey records in memory during the program session. As, the given scenario for the assignment CityRide lite requires the user to add many journeys and later list or remove them, using ArrayList is the most suitable way to handle journeys. Key coding ideas you could reuse in your program:

ArrayList

.add()

.remove() Screenshot of research:



AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
   List list = Collections.synchronizedList(new ArrayList(...));
```

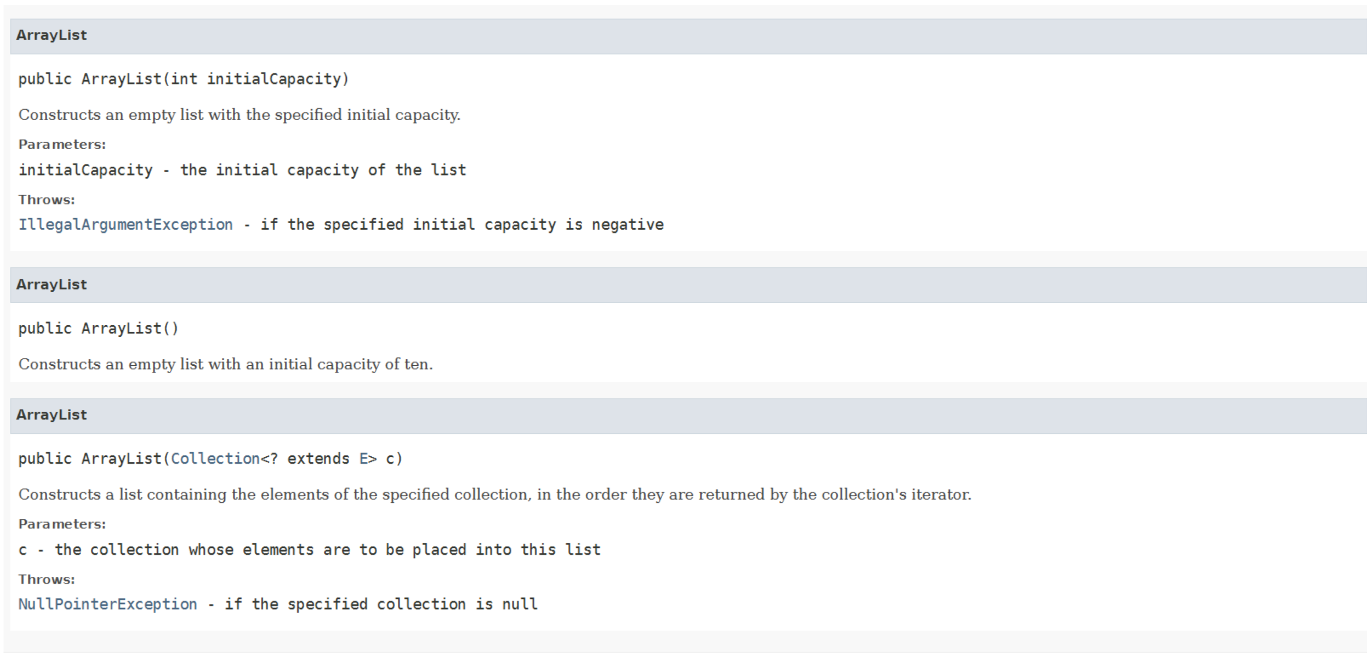
The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:
1.2

See Also:



ArrayList

```
public ArrayList(int initialCapacity)
```

Constructs an empty list with the specified initial capacity.

Parameters:
`initialCapacity - the initial capacity of the list`

Throws:
`IllegalArgumentException - if the specified initial capacity is negative`

ArrayList

```
public ArrayList()
```

Constructs an empty list with an initial capacity of ten.

ArrayList

```
public ArrayList(Collection<? extends E> c)
```

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Parameters:
`c - the collection whose elements are to be placed into this list`

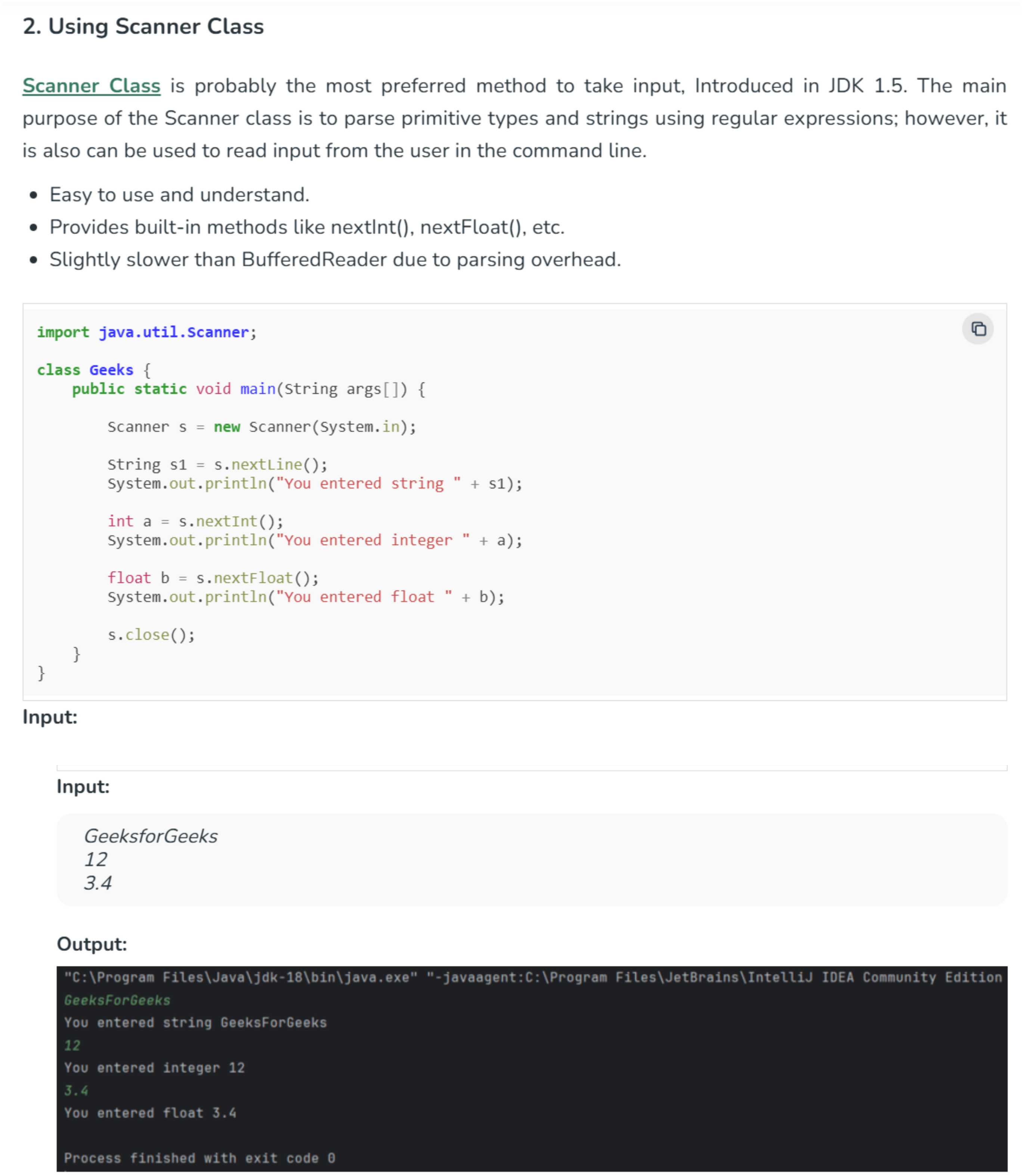
Throws:
`NullPointerException - if the specified collection is null`

Title of research: Java Scanner Input Validation and Error handling Reference (link): https://www.geeksforgeeks.org/ways-to-read-input-from-console-in-java/ How does the research help with coding practise?: This research helped me understand how to correctly take user input from the console and validate it. As, the program rejects the invalid zones, passenger types or blank inputs. So, this will help me to keep running the program with wrong inputs. Key coding ideas you could reuse in your program:

-using scanner to take user input

-using loops for safe validation

-Reprompting users for valid input Screenshot of research:



2. Using Scanner Class

Scanner Class is probably the most preferred method to take input, Introduced in JDK 1.5. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions; however, it is also can be used to read input from the user in the command line.

- Easy to use and understand.
- Provides built-in methods like nextInt(), nextFloat(), etc.
- Slightly slower than BufferedReader due to parsing overhead.

```java
import java.util.Scanner;

class Geeks {
    public static void main(String args[]) {

        Scanner s = new Scanner(System.in);

        String s1 = s.nextLine();
        System.out.println("You entered string " + s1);

        int a = s.nextInt();
        System.out.println("You entered integer " + a);

        float b = s.nextFloat();
        System.out.println("You entered float " + b);

        s.close();
    }
}
```

Input:

Input:

GeeksforGeeks
12
3.4

Output:

```
"C:\Program Files\Java\jdk-18\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
GeeksForGeeks
You entered string GeeksForGeeks
12
You entered integer 12
3.4
You entered float 3.4

Process finished with exit code 0
```

Program Code

Paste the current program code created so far. It does not have to be runnable code (document though if it does not work!)

Program code goes here:

```
import java.util.ArrayList;
import java.util.Scanner;

public class CityRideLite {
```

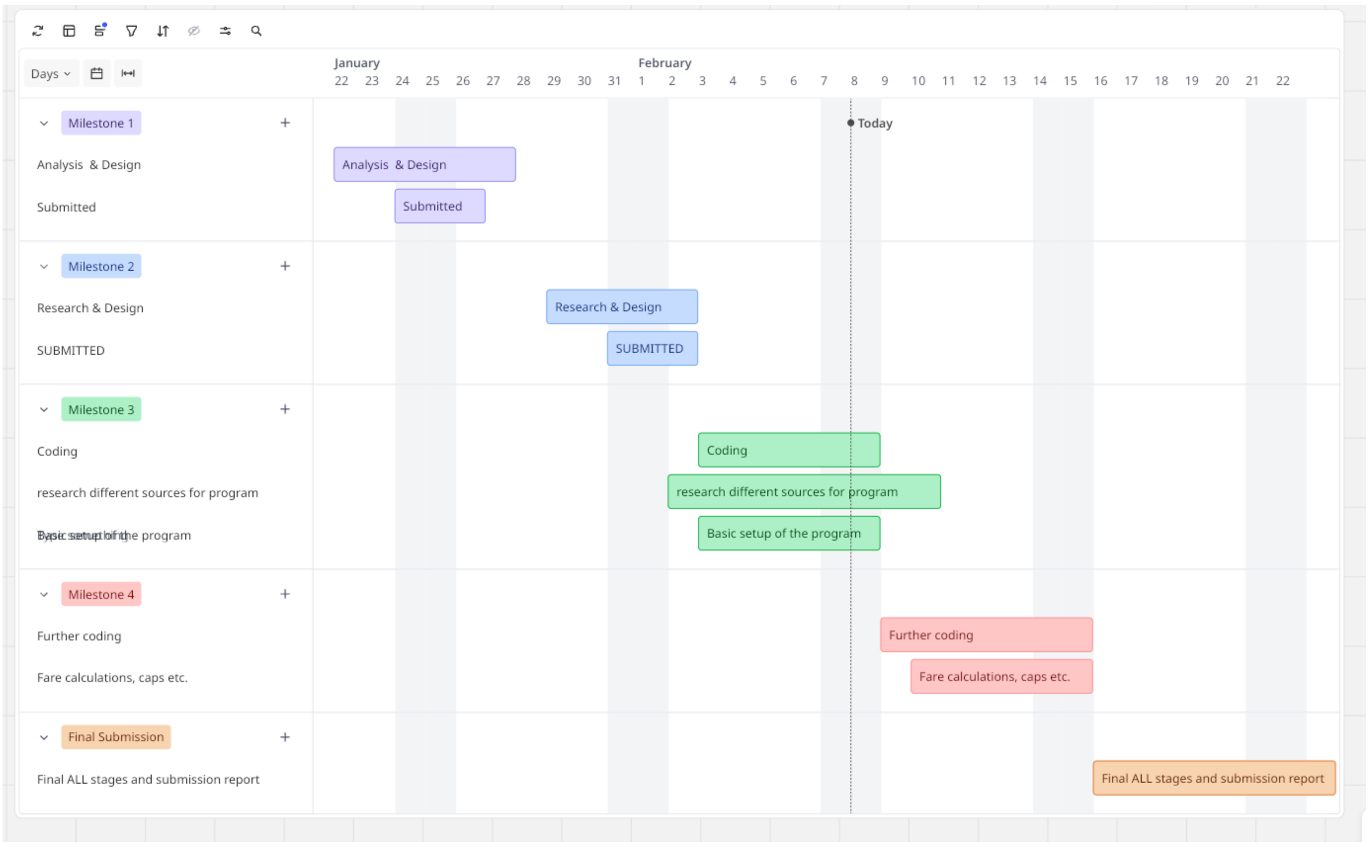
```java
private final ArrayList<Journey> journeyList;
private int nextJourneyId;

public CityRideLite() {
    journeyList = new ArrayList<>();
    nextJourneyId = 1;
}

public static void main(String[] args) {

    CityRideLite app = new CityRideLite();
    app.runProgram();
}

private void runProgram() {

    Scanner inputScanner = new Scanner(System.in);
    boolean isRunning = true;

    System.out.println("---------------------------------");
    System.out.println("     Welcome to CityRide Lite    ");
    System.out.println("---------------------------------");

    while (isRunning) {

        displayMainMenu();
        String userChoice = inputScanner.nextLine();

        if (userChoice.equals("1")) {
            addJourney(inputScanner);
        } else if (userChoice.equals("2")) {
            listAllJourneys();
        } else if (userChoice.equals("3")) {
            isRunning = false;
            System.out.println("Goodbye!");
        } else {
            System.out.println("Error: Please select a valid option.");
        }
    }

    inputScanner.close();
}


private void displayMainMenu() {

    System.out.println("\n--- Main Menu ---");
    System.out.println("1. Add Journey");
    System.out.println("2. List Journeys");
    System.out.println("3. Exit");
    System.out.print("Enter choice: ");
}


private void addJourney(Scanner inputScanner) {

    System.out.println("\n--- Add Journey ---");

    System.out.print("Enter date (DD/MM/YYYY): ");
    String journeyDate = inputScanner.nextLine();

    int fromZone = getValidZone(inputScanner, "Enter starting zone (1-5): ");
    int toZone = getValidZone(inputScanner, "Enter destination zone (1-5): ");

    System.out.print("Enter passenger type (Adult/Student/Child/Senior): ");
    String passengerType = inputScanner.nextLine();

    System.out.print("Enter time band (Peak/Off-Peak): ");
    String timeBand = inputScanner.nextLine();

    Journey newJourney = new Journey(
            nextJourneyId,
            journeyDate,
            fromZone,
            toZone,
            passengerType,
            timeBand
    );

    journeyList.add(newJourney);
    nextJourneyId++;

    System.out.println("Journey added successfully.");
    System.out.println("Zones crossed: " + newJourney.getZonesCrossed());
}


private void listAllJourneys() {

    System.out.println("\n--- Journey Records ---");

    if (journeyList.isEmpty()) {
        System.out.println("No journeys have been recorded yet.");
        return;
    }

    for (Journey journey : journeyList) {
        System.out.println(journey);
    }
}
```

```java
private int getValidZone(Scanner inputScanner, String message) {

    int zoneNumber = 0;
    boolean isValidZone = false;

    while (!isValidZone) {

        System.out.print(message);

        try {
            zoneNumber = Integer.parseInt(inputScanner.nextLine());

            if (zoneNumber >= 1 && zoneNumber <= 5) {
                isValidZone = true;
            } else {
                System.out.println("Error: Zones must be between 1 and 5.");
            }

        } catch (NumberFormatException exception) {
            System.out.println("Error: Please enter a valid numeric zone.");
        }
    }

    return zoneNumber;
}


private static class Journey {

    private final int journeyId;
    private final String date;
    private final int fromZone;
    private final int toZone;
    private final String passengerType;
    private final String timeBand;
    private final int zonesCrossed;

    public Journey(int journeyId,
                   String date,
                   int fromZone,
                   int toZone,
                   String passengerType,
                   String timeBand) {

        this.journeyId = journeyId;
        this.date = date;
        this.fromZone = fromZone;
        this.toZone = toZone;
        this.passengerType = passengerType;
        this.timeBand = timeBand;
        this.zonesCrossed = Math.abs(toZone - fromZone) + 1;
    }

    public int getZonesCrossed() {
        return zonesCrossed;
    }


    @Override
    public String toString() {

        return "ID: " + journeyId +
                " | Date: " + date +
                " | From Zone: " + fromZone +
                " | To Zone: " + toZone +
                " | Passenger: " + passengerType +
                " | Time Band: " + timeBand +
                " | Zones Crossed: " + zonesCrossed;
    }
}
```

}

The program currently works but it is not completed !

Updated Gantt Chart



Diary Entries:-

04/02/26- Today, I started coding the cityRideLite program by creating the main menu and program loop. At first, I wasnt sure how to keep the program readable, then i seperated tasks into smaller methods like addJourney() and listAllJourneys().

05/02/26- I researched how to store journeys properly in memory and understood that Arraylist is the best structure for this program. I went through all the sources and they helped me implement an ArrayList so that journeys can be added easily during the program.

06/02/25 - While going through my inputs, i figured out that zones values entered by users are read as strings, so i had to convert them into integers. I statrted my research again, https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#parseInt-java.lang.String. After going through this I used Integer.parseInt(), but entering letters were causing errors. So to fix this, I added try/catch validation so the programs re run instead of crashing.

07/02/26 - I implemented the zones crossed formula given in the scenario which was abs(toZone - fromZone) + 1. How to implement this correctly i went through this https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html , so that the answer is positive.

08/02/26 - Towards the submission, I worked on making the journey output clearer when listing journeys. I discovered that Java prints object references by default, which is not helpful for the user.To solve this, I researched the @Override annotation and rewrote the toString() method inside the Journey class. This allowed journey details like date, zones, passenger type, and zones crossed to display in a readable format.Overall, the program can now successfully add journeys, calculate zones crossed using the correct formula, and list journeys back to the user.