IY4113 Milestone 3

Assessment Details	Please Complete All Details
Group	A
Module Title	IY4113 Applied Software Engineering using Object-Orientated Programming
Assessment Type	Java Fundamentals
Module Tutor Name	Jonathan Shore
Student ID Number	P488018
Date of Submission	8/02/2026
Word Count	1935

- [x] *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.*

- [x] *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 in Java

Reference (link): [Java ArrayList](https://www.w3schools.com/java/java_arraylist.asp)

How does the research help with coding practise?

This research was on understanding how the ArrayList class can be used to store multiple objects during program execution. Since the CityRide Lite application requires all journeys to be stored in memory for a single session, a data structure was required.

The W3Schools tutorial provided explanations and examples of how to create an ArrayList, add objects using the add() method, retrieve elements and iterate through them using a for each loop. This helped the implementation of the journeyList within the CityRideManager class where each Journey object is stored after being created.

Understanding how collections work helped my ability to manage dynamic data without using arrays of fixed size. It also ensured that journeys could be displayed, counted, and processed when generating summaries.

Key coding ideas you could reuse in your program:

Declaring ArrayList<Journey>

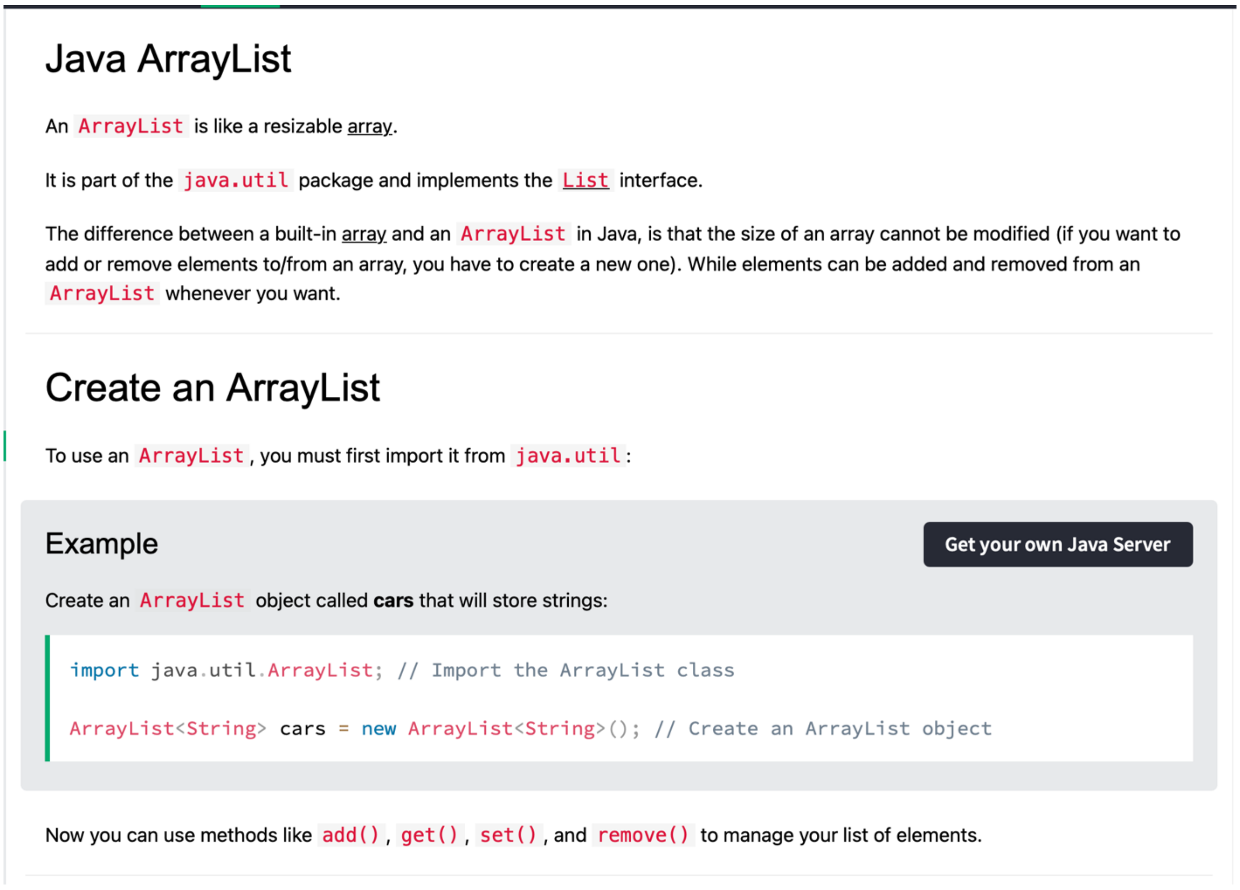
Using add() to insert objects

Using size() to count stored items

Using for-loops to display stored journeys

Checking if a list is empty before printing results

Screenshot of research:



Java ArrayList

An `ArrayList` is like a resizable array.

It is part of the `java.util` package and implements the `List` interface.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want.

Create an ArrayList

To use an `ArrayList`, you must first import it from `java.util`:

Example

Create an `ArrayList` object called **cars** that will store strings:

```java
import java.util.ArrayList; // Import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Now you can use methods like `add()`, `get()`, `set()`, and `remove()` to manage your list of elements.

Add Elements

To add elements to an `ArrayList`, use the `add()` method:

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Try it Yourself »

You can also add an element at a specified position by referring to the index number:

Example

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");

    cars.add(0, "Mazda"); // Insert element at the beginning of the list (0)

    System.out.println(cars);
  }
}
```

Try it Yourself »

An `ArrayList` keeps elements in the same order you add them, so the first item you add will be at index 0, the next at index 1, and so on.

Access an Element

To access an element in the `ArrayList`, use the `get()` method and refer to the index number:

Example

```
cars.get(0);  // Get the first element
```

Try it Yourself »

Loop Through an ArrayList

Loop through the elements of an `ArrayList` with a `for` loop, and use the `size()` method to specify how many times the loop should run:

Example

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

Try it Yourself »

You can also loop through an `ArrayList` with the **for-each** loop:

Title of research:

Using ChatGPT to check code structure and program design

Reference (link):

https://chat.openai.com/

How does the research help with coding practise?:

ChatGPT was used as a research and review tool during the process of the CityRide Lite application. Rather than making a solution, it was used to review program structure and suggest improvements in clarity, organisation, and logical flow.

For example, at early implementation, i was not sure whether to place all logic in the main class or separate it into multiple classes. ChatGPT suggested maintaining a similar design pattern to the classroom example (Car and Garage), where one class represents the data model (Journey) and another manages operations (CityRidemManager). This helped the separation of responsibilities and improved readability.

ChatGPT was also used to confirm the logic for daily cap enforcement and discount calculations. It helped ensure that the charged fare would not exceed the passenger type, daily limit.

It helping improve my confidence in decision-making and professional programming practices.

Key coding ideas you could reuse in your program:

Structuring the program into separate classes (Journey, Manager, Main)

Implementing daily cap logic using conditional statements

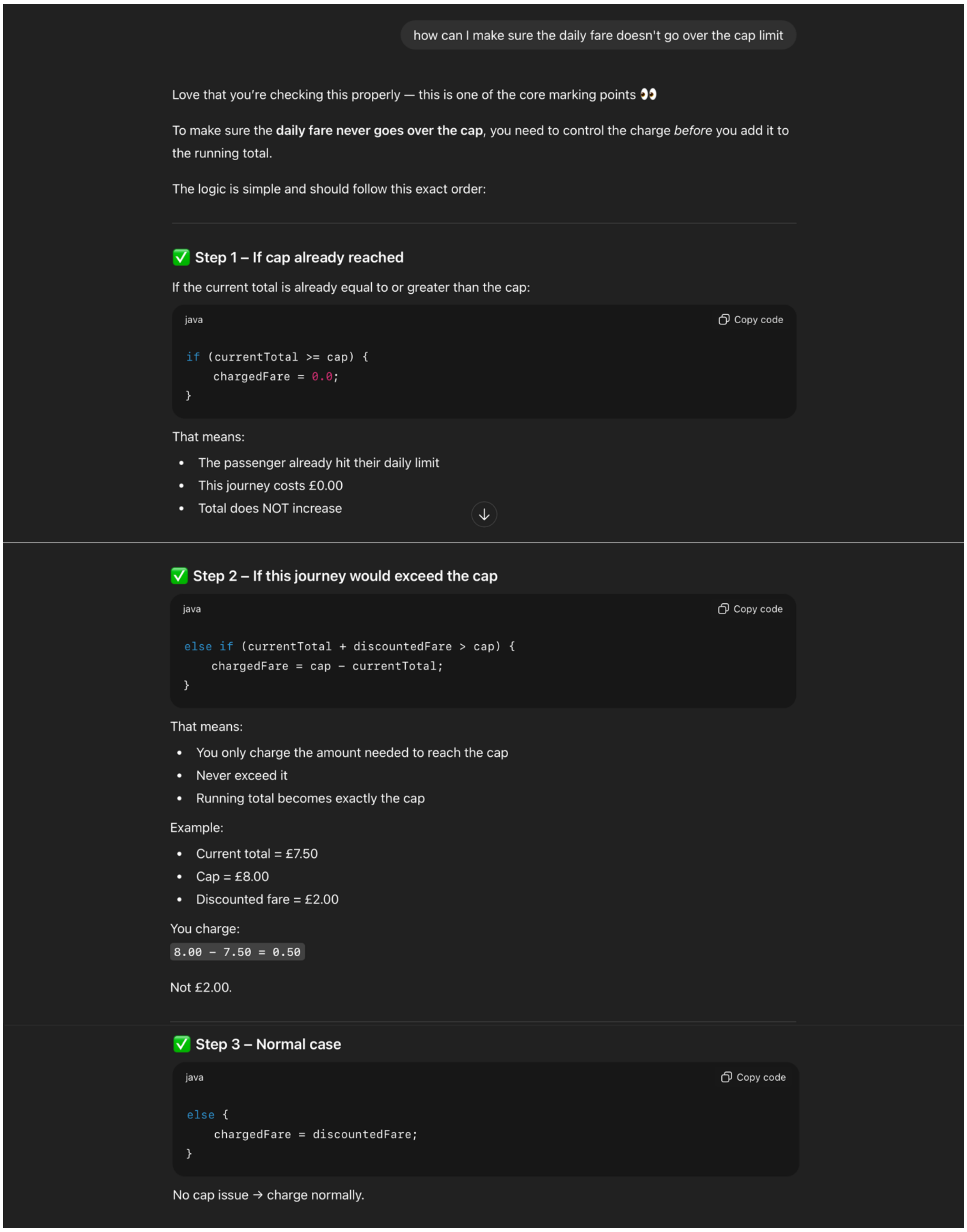
Ensuring consistent naming

Improving menu flow clarity

Refining conditional structures for discounts and totals

Checking for logical errors in fare calculation

Screenshot of research:



how can I make sure the daily fare doesn't go over the cap limit

Love that you're checking this properly — this is one of the core marking points 👀

To make sure the **daily fare never goes over the cap**, you need to control the charge *before* you add it to the running total.

The logic is simple and should follow this exact order:

✅ Step 1 – If cap already reached

If the current total is already equal to or greater than the cap:

```java
if (currentTotal >= cap) {
    chargedFare = 0.0;
}
```

That means:

- The passenger already hit their daily limit
- This journey costs £0.00
- Total does NOT increase

✅ Step 2 – If this journey would exceed the cap

```java
else if (currentTotal + discountedFare > cap) {
    chargedFare = cap - currentTotal;
}
```

That means:

- You only charge the amount needed to reach the cap
- Never exceed it
- Running total becomes exactly the cap

Example:

- Current total = £7.50
- Cap = £8.00
- Discounted fare = £2.00

You charge:
`8.00 - 7.50 = 0.50`

Not £2.00.

✅ Step 3 – Normal case

```java
else {
    chargedFare = discountedFare;
}
```

No cap issue → charge normally.

Title of research: Creating Menu-Driven Programs in Java Using Loops and Switch Statements

Reference (link):[Java Switch](https://www.w3schools.com/java/java_switch.asp)

How does the research help with coding practise?

This research is based on how to build a repeating console menu using loops and conditional structures. The CityRide Lite program needs the main menu to display until the user selects the exit option. The W3Schools tutorials on while loops and switch statements provided practical examples of how user input can control program flow.

This supported the implementation of the do-while loop in the main class which ensures that the menu remains active throughout the session. The switch structure also allows different program features (add journey, view journeys, summary) to be executed based on user selection.

Key coding ideas you could reuse in your program:

Using do-while to repeat the menu

Using switch for user choices

Capturing input using Scanner

Preventing program termination on invalid menu options

Structuring clear user prompts

Screenshot of research:

Java Switch Statements

Instead of writing **many** `if..else` statements, you can use the `switch` statement.

Think of it like ordering food in a restaurant: *If you choose number 1, you get Pizza. If you choose 2, you get a Burger. If you choose 3, you get Pasta. Otherwise, you get nothing*.

The `switch` statement selects one of many code blocks to be executed:

Syntax

Get your own Java Server

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once.
- The result is compared with each `case` value.
- If there is a match, the matching block of code runs.
- The `break` statement stops the switch after the matching case has run.
- The `default` statement runs if there is no match.

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
```

The break Keyword

When Java reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The `default` keyword specifies some code to run if there is no case match:

Example

```
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Try it Yourself »

Title of research:

Understanding Object-Oriented Programming and Encapsulation in Java

Reference (link):

[Java OOP (Object-Oriented Programming)](https://www.w3schools.com/java/java_oop.asp)
[Java Encapsulation and Getters and Setters](https://www.w3schools.com/java/java_encapsulation.asp)

How does the research help with coding practise?:

This focused on object-oriented programming principles, particularly encapsulation and class structure. The CityRide Lite application follows a similar design pattern to the Car and Garage example which we did in class where one class represents the data model (Journey) and another class manages the operations (CityRide Manager)

The W3Schools OOP tutorials shows how private variables protect object data and how getters allow controlled access. This helped the design of the Journey class ensuring that attributes such as id, fromZone and chargedFare remain private and are accessed only through methods.

Separating responsibilities between classes improved program organisation and readability. The main class handles user interaction while the manager class handles business logic like adding journeys and calculating fares. This shows following of the professional programming standards and clear structural design.

Key coding ideas you could reuse in your program:

Creating classes with private attributes

Using constructors to initialise objects

Implementing getter methods

Separating data and management logic

Following structured OOP design

Screenshot of research:

Java - What is OOP?

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Java - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

class	objects
Fruit	Apple

Another example:

class	objects
Car	Volvo
	Audi
	Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

You will learn much more about classes and objects in the next chapter.

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:

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

class Journey {

    private int id;
    private int fromZone;
    private int toZone;
    private String passengerType;
    private String timeBand;
    private double chargedFare;

    public Journey(int id, int fromZone, int toZone,
                   String passengerType, String timeBand,
                   double chargedFare) {

        this.id = id;
        this.fromZone = fromZone;
        this.toZone = toZone;
        this.passengerType = passengerType;
        this.timeBand = timeBand;
        this.chargedFare = chargedFare;
    }

    public int getId() {
        return id;
    }

    public String getPassengerType() {
        return passengerType;
    }

    public double getChargedFare() {
        return chargedFare;
    }

    public void displayDetails() {
        System.out.println("ID: " + id +
                " | From: " + fromZone +
                " | To: " + toZone +
                " | Passenger: " + passengerType +
                " | Time: " + timeBand +
                " | Charged: ¬£" + chargedFare);
    }

}

class CityRideManager {

    private List<Journey> journeyList = new ArrayList<>();
    private int nextId = 1;

    private double adultTotal = 0;
    private double studentTotal = 0;
    private double childTotal = 0;
    private double seniorTotal = 0;

    public void addJourney(int fromZone, int toZone,
                       String passengerType, String timeBand) {

        int zonesCrossed = Math.abs(toZone - fromZone) + 1;

        double baseFare = zonesCrossed * 2.0;

        double discount = 0;
        double cap = 0;
        double currentTotal = 0;

        if (passengerType.equalsIgnoreCase("Adult")) {
            discount = 0;
            cap = 8.0;
            currentTotal = adultTotal;
        }
        else if (passengerType.equalsIgnoreCase("Student")) {
            discount = 0.25;
            cap = 6.0;
            currentTotal = studentTotal;
        }
        else if (passengerType.equalsIgnoreCase("Child")) {
            discount = 0.5;
            cap = 4.0;
            currentTotal = childTotal;
        }
        else if (passengerType.equalsIgnoreCase("Senior")) {
            discount = 0.3;
            cap = 7.0;
            currentTotal = seniorTotal;
        }

        double discountedFare = baseFare - (baseFare * discount);

        double chargedFare;

        if (currentTotal >= cap) {
            chargedFare = 0;
        }
        else if (currentTotal + discountedFare > cap) {
            chargedFare = cap - currentTotal;
        }
        else {
            chargedFare = discountedFare;
        }

        if (passengerType.equalsIgnoreCase("Adult")) {
            adultTotal += chargedFare;
        }
        else if (passengerType.equalsIgnoreCase("Student")) {
            studentTotal += chargedFare;
        }
        else if (passengerType.equalsIgnoreCase("Child")) {
            childTotal += chargedFare;
        }
        else if (passengerType.equalsIgnoreCase("Senior")) {
            seniorTotal += chargedFare;
        }

        Journey j = new Journey(nextId++, fromZone, toZone,
                passengerType, timeBand, chargedFare);

        journeyList.add(j);

        System.out.println("Journey added. Charged ¬£" + chargedFare);
    }

    public void showAllJourneys() {
        if (journeyList.isEmpty()) {
            System.out.println("No journeys recorded.");
        } else {
            for (Journey j : journeyList) {
                j.displayDetails();
            }
        }
    }

    public void showSummary() {

        int totalJourneys = journeyList.size();
        double totalCost = adultTotal + studentTotal + childTotal + seniorTotal;

        double average = 0;
        if (totalJourneys > 0) {
            average = totalCost / totalJourneys;
        }

        System.out.println("Total Journeys: " + totalJourneys);
        System.out.println("Total Cost: ¬£" + totalCost);
        System.out.println("Average Cost: ¬£" + average);
    }

}

public class CityRideLiteApp {
```

```
public static void main(String[] args) {

    Scanner scanner = new Scanner(System.in);
    CityRideManager manager = new CityRideManager();

    int choice;

    do {
        System.out.println("\nCityRide Lite");
        System.out.println("1. Add Journey");
        System.out.println("2. View All Journeys");
        System.out.println("3. Daily Summary");
        System.out.println("4. Exit");
        System.out.print("Choice: ");

        choice = scanner.nextInt();
        scanner.nextLine();

        switch (choice) {

            case 1:
                System.out.print("Enter From Zone (1-5): ");
                int from = scanner.nextInt();

                System.out.print("Enter To Zone (1-5): ");
                int to = scanner.nextInt();
                scanner.nextLine();

                System.out.print("Enter Passenger Type (Adult/Student/Child/Senior): ");
                String passenger = scanner.nextLine();

                System.out.print("Enter Time Band (Peak/OffPeak): ");
                String time = scanner.nextLine();

                manager.addJourney(from, to, passenger, time);
                break;

            case 2:
                manager.showAllJourneys();
                break;

            case 3:
                manager.showSummary();
                break;

            case 4:
                System.out.println("Exiting...");
                break;

            default:
                System.out.println("Invalid choice.");
        }

    } while (choice != 4);
```

```
        scanner.close();
    }

}
```

Updated Gantt Chart

Add your updated Gantt chart here!

Diary Entries

Add diary entries here detailing what you have done, wny you have done it, and any problems encountered.

Thursday

Today I looked at the full assessment brief for CityRide Lite to ensure I clearly understood all functional requirements. I looked at fare rules, daily caps, passenger types and validation requirements. I updated my Gantt chart to show the progress.

I began reviewing my existing class structure from previous practical sessions (CarApp example) to make sure I would follow a similar structure for consistency. I found out that I would need a Journey class and a main application class to manage the menu and logic.

One difficulty i encountered today was understanding how the daily cap should behave when the total is close to the limit. I noted this as something needing further research.

Friday

Today I focused on research to support my implementation. I reviewed W3Schools documentation on ArrayList and try-catch exception handling to ensure I could correctly store journeys and validate user input. This helped me confirm that using an ArrayList would allow me to add, list and iterate through journeys.

I also used ChatGPT to review my cap logic structure. This helped me understand how to structure conditional statements so that the total never exceeds the daily cap and becomes £0.00 once the cap is reached.

I then implemented the basic program structure including:

Main menu loop

Add journey function

Basic fare calculation

Discount application

A challenge I faced was ensuring the running totals were updated correctly for each passenger type without mixing them.

Saturday

Today I completed the core functionality of the program, I implemented:

Daily cap enforcement

Running totals per passenger type

Summary calculations (total journeys, total cost, average cost)

Input validation for zones and passenger types

I tested multiple scenarios to confirm that:

The cap is never exceeded

Once the cap is reached, further journeys are charged £0.00

Invalid inputs do not crash the program

During testing, I found that incorrect input types (such as letters instead of numbers) caused errors. I resolved this using try-catch blocks based on my research.

I updated my Gantt chart to show the progress

Sunday

Today I finalised the program and tested all that i had included. I verified that the menu loops correctly until exit, summaries calculate accurately and totals reset when required.

I checked my research section and ensured it explained how external sources influenced my coding decisions particularly in relation to ArrayList usage, exception handling and cap logic implementation.

I looked at the code formatting and ensured comments were clear and brief. I confirmed that the program aligns with the functional requirements.

The project was completed today and all required things like research Gantt chart, diary entries and program code are now prepared for submission.
