FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)

Department of Computer Science

COMP 468(A) ASSIGNMENT # 1

SMART RIDE SHARING SYSTEM

JOSHUA SADAQAT [240-545460]

Instructor: Mr. Adeem Akhtar

Session 2020-2024

OBJECTIVE

The goal of this project is to create a **Smart Ride-Sharing System** using Kotlin. The system allows for the management of **drivers**, **riders**, and **ride requests**. The application automatically matches riders with available drivers based on **proximity**, simulates traffic conditions, calculates estimated ride times, and allows riders to rate drivers after completing rides.

PROJECT STRUCTURE

The system is divided into several classes, each responsible for specific parts of the functionality. Below are the key components:

1. **Main.kt**: Contains the user interface (console-based) for interaction.
2. **RideSharingSystem.kt**: Handles the core logic of assigning rides, managing drivers and riders, and updating ride status.
3. **Driver.kt**: Represents a driver with attributes such as location, availability, and rating.
4. **Rider.kt**: Represents a rider with attributes such as current location, destination, and request status.
5. **Ride.kt**: Represents the ride between a rider and an assigned driver.
6. **Utils.kt**: Contains utility functions like the Haversine formula (to calculate distances between coordinates) and traffic simulation.

EXPLANATION OF CLASSES AND FUNCTIONS

Main.kt – User Interface

This file handles user interaction with a **console-based menu**. Users can:

- Add new drivers and riders to the system.
- Request rides.
- Display drivers and riders.
- Rate drivers after a ride.
- Exit the system.

Menu-driven interface

```
while (true) {
    println("\nMenu:")
    println("1. Add Driver")
    println("2. Add Rider")
    println("3. Request Ride")
    println("4. Display Drivers")
    println("5. Display Riders")
    println("6. Rate Driver")
    println("7. Exit")
    print("Choose an option: ")
```

The **menu** allows users to repeatedly choose different actions until they decide to exit. This is handled using a while loop that continues until the user selects the exit option.

RideSharingSystem.kt

This class handles the **core logic** of the system, including adding drivers and riders, assigning rides, and updating driver ratings.

Key Methods:

1. **addDriver (driver: Driver)**: Adds a new driver to the system.
2. **addRider (rider: Rider)**: Adds a new rider to the system.

```kotlin
class RideSharingSystem {
    val drivers = mutableListOf<Driver>()
    val riders = mutableListOf<Rider>()

    fun addDriver(driver: Driver) {
        drivers.add(driver)
    }

    fun addRider(rider: Rider) {
        riders.add(rider)
    }
```

3. **assignRide(rider: Rider)**: This method matches a rider with the **closest available driver**. The **Haversine formula** is used to calculate the distance between the rider's and driver's locations. If two drivers are equally close, the system selects the one with the **highest rating**. The ride is assigned, and the rider's status is updated to "Accepted."

```kotlin
class RideSharingSystem {
    // Assign a ride to the rider
    fun assignRide(rider: Rider): Ride? {
        val availableDrivers = drivers.filter { it.isAvailable }
        if (availableDrivers.isEmpty()) {
            println("No available drivers for ${rider.name}.")
            return null
        }

        val closestDriver = availableDrivers.minWithOrNull {...}

        closestDriver?.let { driver ->
            val rideDistance = Utils.haversine(rider.currentLocation.first, rider.currentLocation.second,
                rider.destinationLocation.first, rider.destinationLocation.second)
            val trafficFactor = Utils.simulateTraffic()

            val ride = Ride(rider, driver, rideDistance, estimatedTime: 0.0)
            ride.calculateEstimatedTime(trafficFactor)

            driver.setAvailability(false)
            rider.requestStatus = "Accepted" // Update ride request status
            println("Ride Assigned: ${rider.name} -> ${driver.name}")
            println("Distance to Destination: ${ride.distanceToDestination} km")
            println("Estimated Time: ${ride.estimatedTime} hours")
            return ride
        }
        return null
    }
```

4. **rateDriver(driver: Driver, rating: Double)**: This method allows the rider to rate the driver after the ride is completed. The driver's rating is updated as an **average** of previous and new ratings.

```kotlin
fun rateDriver(driver: Driver, rating: Double) {
    driver.updateRating(rating)
    println("${driver.name}'s new rating is ${driver.rating}")
}
```

5. **displayDrivers()** and **displayRiders()**: These methods display all the drivers and riders in the system in a formatted table, including their current **availability** and **ride request status**.

```kotlin
fun displayDrivers() {
    println("------------------------------------------------")
    println("| Driver Name | Availability | Rating | Location |")
    println("------------------------------------------------")
    drivers.forEach {
        val availability = if (it.isAvailable) "Available" else "Unavailable"
        println("| ${it.name.padEnd(11)} | ${availability.padEnd(12)} | ${it.rating}   | ${it.currentLocation} |")
    }
    println("------------------------------------------------")
}


fun displayRiders() {
    println("------------------------------------------------------")
    println("| Rider Name | Request Status | Current Location | Destination |")
    println("------------------------------------------------------")
    riders.forEach {
        println("| ${it.name.padEnd(10)} | ${it.requestStatus.padEnd(14)} | ${it.currentLocation} | ${it.destinationLocation} |")
    }
    println("------------------------------------------------------")
}
```

Driver.kt

The Driver class represents a driver in the system. It has attributes like the driver's name, current location, availability, and rating.

```kotlin
data class Driver(
    val name: String,
    var currentLocation: Pair<Double, Double>,
    var isAvailable: Boolean = true,
    var rating: Double = 5.0
) {
    fun setAvailability(available: Boolean) {
        isAvailable = available
    }

    fun updateRating(newRating: Double) {
        rating = (rating + newRating) / 2.0
    }
}
```

Key Attributes

- **isAvailable**: Tracks whether the driver is available for a new ride.
- **rating**: Tracks the driver's current rating, which gets updated after each ride.

Rider.kt

The Rider class represents a rider in the system. The rider has attributes for the current location, destination, and ride request status.

```kotlin
data class Rider(
    val name: String,
    var currentLocation: Pair<Double, Double>,
    var destinationLocation: Pair<Double, Double>,
    var requestStatus: String = "In Queue" // Default status
) {
    fun requestRide(system: RideSharingSystem) {
        system.assignRide( rider: this)
    }
}
```

- **requestRide ()**: Initiates the ride request process by calling the assignRide() method of the RideSharingSystem.
- **requestStatus**: Tracks whether the rider's ride is "In Queue" or "Accepted."

Ride.kt:

The Ride class represents a ride between a rider and an assigned driver. It calculates the distance between the riders's starting location and destination, as well as the estimated ride time based on traffic conditions.

```kotlin
data class Ride(
    val rider: Rider,
    val assignedDriver: Driver,
    val distanceToDestination: Double,
    var estimatedTime: Double
) {
    // Calculates distance using the Haversine formula
    fun calculateDistance(): Double {
        return Utils.haversine(
            rider.currentLocation.first, rider.currentLocation.second,
            rider.destinationLocation.first, rider.destinationLocation.second
        )
    }

    // Calculates estimated time for the ride
    fun calculateEstimatedTime(trafficFactor: Double) {
        val distance = calculateDistance()
        val baseTime = distance / 50.0 // Assuming average speed of 50 km/h
        estimatedTime = baseTime * trafficFactor
    }
}
```

Utils.kt

This file contains utility functions, such as the **Haversine formula** to calculate the distance between two geographical points and a **traffic simulator** to simulate real-world traffic delays.

1. **haversine()**: This formula calculates the distance between two locations on the Earth given their latitude and longitude.

```
object Utils {
    fun haversine(lat1: Double, lon1: Double, lat2: Double, lon2: Double): Double {
        val R = 6371e3 // Earth's radius in meters
        val φ1 = Math.toRadians(lat1)
        val φ2 = Math.toRadians(lat2)
        val Δφ = Math.toRadians(lat2 - lat1)
        val Δλ = Math.toRadians(lon2 - lon1)
        val a = Math.sin(Δφ / 2) * Math.sin(Δφ / 2) +
                Math.cos(φ1) * Math.cos(φ2) *
                Math.sin(Δλ / 2) * Math.sin(Δλ / 2)
        val c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a))
        return R * c / 1000 // Distance in kilometers
    }
```

2. **simulateTraffic()**: This function simulates traffic delays by generating a random traffic factor between 1.0 and 2.0.

```
    fun simulateTraffic(): Double {
        return (1 ≤ .. ≤ 20).random() / 10.0 // Random traffic factor from 1.0 to 2.0
    }
}
```

Special Features & Kotlin Specifics:

1. **Data Classes**: Kotlin's **data classes** are used to create Driver, Rider, and Ride classes. These provide built-in features like equals (), hashCode (), and toString().
2. **Null Safety**: The system uses Kotlin's **null safety** features, such as the ?.let construct, to avoid null pointer exceptions when assigning rides.
3. **Comparator Functions**: The **minWithOrNull** method is used to compare drivers based on both **distance** and **rating**. This method allows multiple criteria for driver selection in a concise way.

CONCLUSION

This project provides a robust **Smart Ride-Sharing System** that manages drivers, riders, and ride assignments based on proximity and traffic conditions. By leveraging Kotlin's advanced features such as **data classes**, **null safety**, and concise **comparator functions**, the system is highly modular, easy to maintain, and efficient.

The system effectively allows users to interact through a simple console interface, where they can add drivers and riders, request rides, and rate drivers. Additionally, the use of the **Haversine formula** ensures accurate distance calculations between the rider and driver locations, while the **traffic simulation** adds realism by introducing variable ride times. The system also updates rider statuses and driver availability dynamically, making it practical for real-world use cases.

Overall, this project demonstrates how Kotlin can be used to create a fully functional, interactive ride-sharing system that handles complex logic while remaining user-friendly and efficient.

OUTPUTS ARE ATTACHED IN A PDF FILE.