Ride-Sharing Service

Imagine you are tasked with building a ride-sharing service system for a startup. The service needs to manage users, drivers, rides, and locations effectively. To do this, you need to design and implement a program in Java that follows Object-Oriented Programming (OOP) principles.

The service revolves around five main components:

1. **Location**: Represents a place in the city, such as "Downtown" or "Airport". Each location is identified by its name.
2. **User:** Represents the customers using the ride-sharing service. Each user has a name and an ID. Users can request rides, provide the starting and ending locations, and retrieve their personal information when needed.
3. **Ride:** Represents a trip from one location to another. Each ride has a unique ID, a starting location, a destination, and a status indicating whether the ride is completed. Users can view the details of the ride, such as its ID and location.
4. **Driver:** Represents the drivers in the system who fulfil ride requests. Each driver has a name, ID, vehicle type (Sedan, SUV), and a status indicating whether they are available to take rides. Drivers can accept rides, complete rides, and toggle their availability status.
5. **RideSharingService:** The central system that manages the interaction between users, drivers, and rides. It allows users and drivers to register, handles ride requests, and manages the completion of rides.

How the System Works

- **Setup:**
  - ➥ Locations are created to represent real-world places (e.g., "Downtown", "Airport").
  - ➥ Users and drivers are registered in the system.
  - ➥ Drivers may start as "available" or "unavailable" based on their status.
- **Ride Request:**
  - ➥ A user requests a ride by specifying a starting location and destination.
  - ➥ The system creates a Ride object to represent the trip and adds it to the list of rides.
- **Driver Assignment:**
  - ➥ A driver checks if they are available and accepts the ride. If unavailable, the system shows an appropriate message.
  - ➥ Once a ride is accepted, the driver becomes unavailable.
- **Ride Completion:**
  - ➥ After the trip is complete, the driver marks the ride as completed.
  - ➥ The driver's availability status is updated so they can take another ride.
- **Error Handling:**
  - ➥ The system handles invalid scenarios, such as:
  - ➥ Trying to accept a ride when a driver is unavailable.
  - ➥ Completing a ride that has already been completed.
  - ➥ Requesting a ride with invalid or missing locations.

Your Task

You are provided with the Main.java file, which outlines how the program should work. The main program contains commented-out test cases and instructions to guide you. Your task is to write the code for all the necessary classes and methods, so the program functions as described.

What to Do

1. Implement the following classes:
   - ➥ Location
   - ➥ User
   - ➥ Ride
   - ➥ Driver
   - ➥ RideSharingService
2. Uncomment the test cases in the Main.java file one at a time and implement the corresponding functionality. For example:
   - ➥ Implement registerUser() before uncommenting its test case.
   - ➥ Implement acceptRide() to handle the ride assignment.
3. Handle edge cases:
   - ➥ Prevent a driver from accepting a ride if unavailable.
   - ➥ Ensure completed rides cannot be re-completed.
   - ➥ Prevent rides from being created with invalid locations.
4. Ensure all methods work correctly by testing each before moving to the next.

Completing this task will teach you how to design a fully functional object-oriented system with interdependent classes.

Test Case

Scenario 1: Request a Ride and Complete It:

1. A user ("Sahid") registers in the system.
2. A driver ("Sakib") registers in the system.
3. The user requests a ride from one location to another ("Downtown" to "Airport").
4. The driver accepts the ride and becomes unavailable.
5. The driver completes the ride.

Steps to Implement:

1. **Create Locations:** Create the locations ("Downtown" and "Airport") to represent real-world places.
2. **Register the User:** Register by creating a User object and passing it to User.registerUser().
3. **Register the Driver:** Register the driver by creating a Driver object and passing it to Driver.registerDriver().

4. **Request a Ride:** Call the requestRide() method from the RideSharingService class, passing the user and the locations.
5. **Accept the Ride:** Call the acceptRide() method from RideSharingService, passing the driver and the ride.
6. **Complete the Ride:** Call the completeRide() method from RideSharingService, passing the driver and the ride.

Final Output:

```
User registered:
Name: Sahid, ID: 1
Driver registered:
Name: Sakib, ID: 101
Ride Requested:
Ride ID: R1, From: Downtown, To: Airport
Ride Accepted
Ride completed for R1
```

Main.java Skeleton for test case 1

```
public class Main {
    public static void main(String[] args) {
        Location loc1 = new Location("Downtown");
        Location loc2 = new Location("Airport");

        // Create User and Driver
        User user1 = new User("Sahid", 1);
        Driver driver1 = new Driver("Sakib", 101, "Sedan", true); // Driver initially available

        // Register User and Driver
        User.registerUser(user1);
        Driver.registerDriver(driver1);

        // Create RideSharingService
        RideSharingService service = new RideSharingService();

        // Request Ride
        Ride ride1 = service.requestRide(user1, loc1, loc2);

        // Accept Ride
        service.acceptRide(driver1, ride1);

        // Complete Ride
        service.completeRide(driver1, ride1);
    }
}
```

Scenario 2: Unavailable Driver and Completed Ride Attempt

1. A user ("Hossain") registers in the system.
2. Two drivers ("Sakib" and "Alisha") register in the system.
    - Driver "Sakib" is unavailable initially.
    - Driver "Alisha" is available initially.
3. The user requests a ride from one location to another ("Downtown" to "Airport").
4. An unavailable driver tries to accept the ride, but the system prevents this.
5. An available driver accepts the ride and becomes unavailable.
6. The driver completes the ride.
7. The system prevents the driver from completing the same ride again.

Steps to Implement:

1. **Create Locations:** Same as Test Case 1.
2. **Register the User:** Create and register a User object for "Hossain."
3. **Register the Drivers:** Create and register two Driver objects:
    - "Sakib" (initially unavailable).
    - "Alisha" (initially available).
4. **Request a Ride:** Call the requestRide() method.
5. **Driver Accepts Ride:**
    - Try to accept the ride with "Sakib" (unavailable).
    - Accept the ride with "Alisha" (available).
6. **Complete the Ride:** Call the completeRide() method for "Alisha."
7. **Attempt to Complete the Same Ride Again:** Call the completeRide() method again for "Alisha" to test the error handling.

Final Output:

```
User registered:
Name: Hossain, ID: 2
Driver registered:
Name: Sakib, ID: 101
Driver registered:
Name: Alisha, ID: 102
Ride Requested:
Ride ID: R2, From: Downtown, To: Airport
Driver is not available.
Ride Accepted
Ride completed for R2
Ride is already completed!
```

Main.java Skeleton for test case 2

```
public class Main {
    public static void main(String[] args) {
        Location loc1 = new Location("Downtown");
        Location loc2 = new Location("Airport");

        // Create User and Drivers
        User user2 = new User("Hossain", 2);
        Driver driver1 = new Driver("Sakib", 101, "Sedan", false); // Initially unavailable
        Driver driver2 = new Driver("Alisha", 102, "SUV", true);  // Initially available

        // Register User and Drivers
        User.registerUser(user2);
        Driver.registerDriver(driver1);
        Driver.registerDriver(driver2);

        // Create RideSharingService
        RideSharingService service = new RideSharingService();

        // Request Ride
        Ride ride2 = service.requestRide(user2, loc1, loc2);

        // Try to accept Ride with an unavailable driver
        service.acceptRide(driver1, ride2);

        // Accept Ride with an available driver
        service.acceptRide(driver2, ride2);

        // Complete Ride
        service.completeRide(driver2, ride2);

        // Attempt to complete the same ride again
        service.completeRide(driver2, ride2);
    }
}
```

Final Output:

```
User registered:
Name: Hossain, ID: 2
Driver registered:
Name: Sakib, ID: 101
Driver registered:
Name: Alisha, ID: 102
Ride Requested:
Ride ID: R2, From: Downtown, To: Airport
Driver is not available.
```

```
Ride Accepted
Ride completed for R2
Ride is already completed!
```

Bonus Marks Question: Handling Duplicate User IDs

The current ride-sharing system allows users to register with their name and a unique ID. However, what happens if a user tries to register with an ID that already exists in the system?

You may solve this problem without using inheritance; however, incorporating inheritance will significantly enhance code reusability and maintainability by reducing redundancy. If you can implement inheritance effectively, bonus marks will be awarded!