Courier Management System

Coding Task 6

Service Provider Interface /Abstract class

Create 2 Interface /Abstract class ICourierUserService and ICourierAdminService interface

ICourierUserService {

// Customer-related functions

placeOrder()

/** Place a new courier order.

* @param courierObj Courier object created using values entered by users

* @return The unique tracking number for the courier order .

Use a static variable to generate unique tracking number. Initialize the static variable in Courier class with some random value. Increment the static variable each time in the constructor to generate next values.

getOrderStatus();

/**Get the status of a courier order.

*@param trackingNumber The tracking number of the courier order.

* @return The status of the courier order (e.g., yetToTransit, In Transit, Delivered). */

cancelOrder()

/** Cancel a courier order.

* @param trackingNumber The tracking number of the courier order to be canceled.

* @return True if the order was successfully canceled, false otherwise.*/

getAssignedOrder();

/** Get a list of orders assigned to a specific courier staff member

* @param courierStaffId The ID of the courier staff member.

* @return A list of courier orders assigned to the staff member.*/

// Admin functions

ICourierAdminService

int addCourierStaff(Employee obj);

/** Add a new courier staff member to the system.

* @param name The name of the courier staff member.

* @param contactNumber The contact number of the courier staff member.

* @return The ID of the newly added courier staff member. */

In this task, I define two key service interfaces that act as contracts for operations within a courier management system:

1. ICourierUserService and
2. ICourierAdminService.

These interfaces ensure that both users and administrators interact with the system in a structured and consistent way. By using interfaces, I promote abstraction and loose coupling, allowing flexibility in implementing different service functionalities while enforcing standard method signatures across various components.

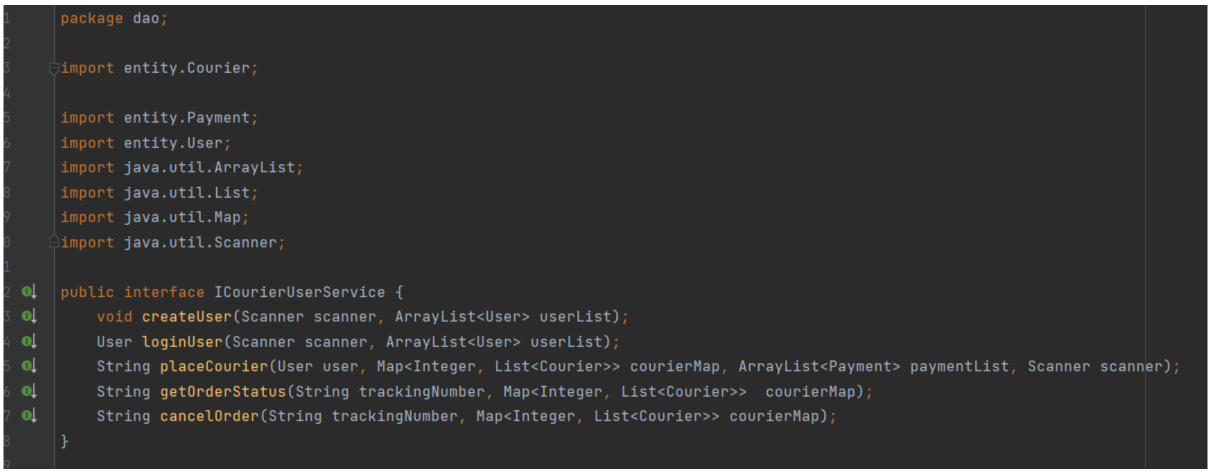
Task 6.1 ICourierUserService

The ICourierUserService interface defines the operations that a **customer (user)** can perform in the courier system. These operations include:

- Placing a courier order
- Checking the status of an order
- Canceling an order
- Viewing orders assigned to a courier staff

The placeOrder() method returns a unique tracking number, auto-generated using a static variable initialized and incremented within the Courier class constructor. This guarantees each order has a distinct identifier.

By providing these method contracts, this interface enables developers to implement consistent and reusable logic for **user-side interactions**.



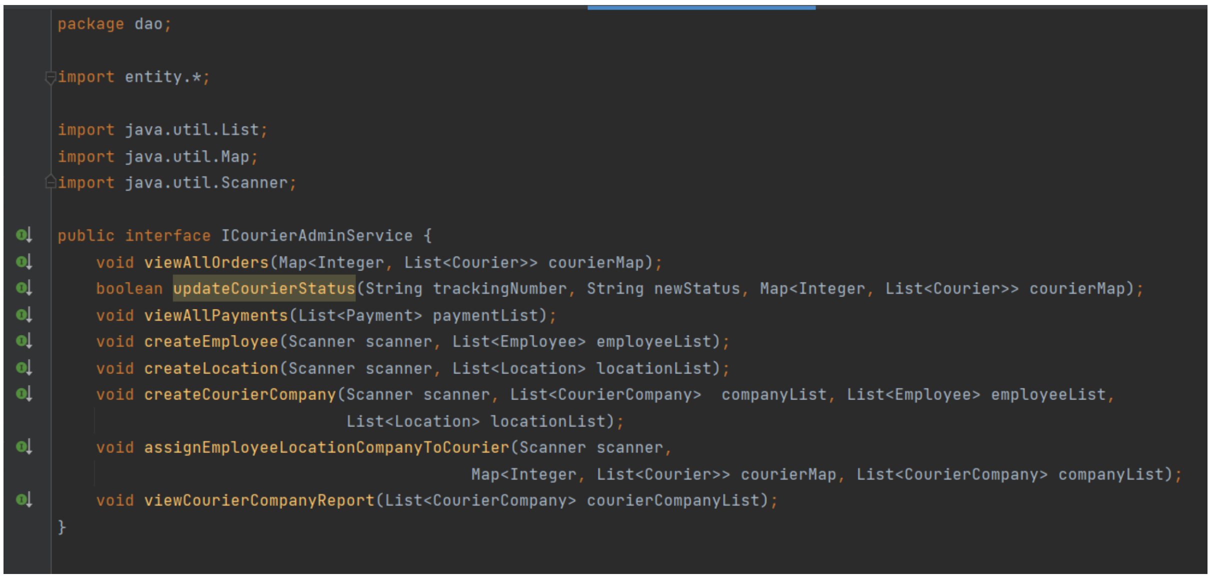
```java
package dao;

import entity.Courier;

import entity.Payment;
import entity.User;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Scanner;

public interface ICourierUserService {
    void createUser(Scanner scanner, ArrayList<User> userList);
    User loginUser(Scanner scanner, ArrayList<User> userList);
    String placeCourier(User user, Map<Integer, List<Courier>> courierMap, ArrayList<Payment> paymentList, Scanner scanner);
    String getOrderStatus(String trackingNumber, Map<Integer, List<Courier>>  courierMap);
    String cancelOrder(String trackingNumber, Map<Integer, List<Courier>> courierMap);
}
```

Task 6.2 ICourierAdminService

The ICourierAdminService interface outlines the responsibilities of an **administrator** in the courier system. Currently, it includes the method addCourierStaff() which:

- Allows the admin to add new courier staff members by accepting an Employee object.
- Returns the **unique ID** of the newly added staff member.

This interface supports **modular design**, making administrative operations such as staff management clean, maintainable, and separated from user logic.

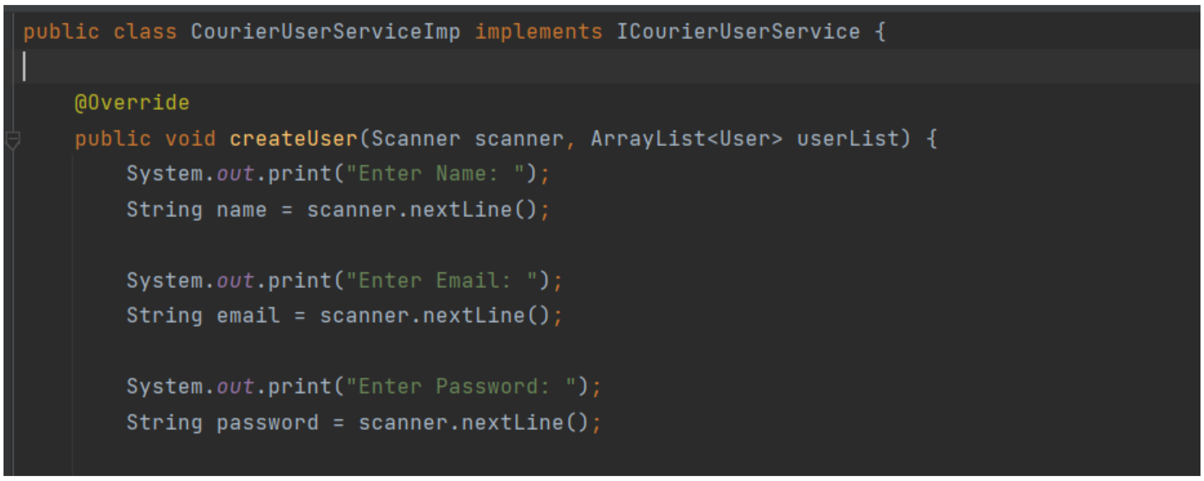


```java
package dao;

import entity.*;

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

public interface ICourierAdminService {
    void viewAllOrders(Map<Integer, List<Courier>> courierMap);
    boolean updateCourierStatus(String trackingNumber, String newStatus, Map<Integer, List<Courier>> courierMap);
    void viewAllPayments(List<Payment> paymentList);
    void createEmployee(Scanner scanner, List<Employee> employeeList);
    void createLocation(Scanner scanner, List<Location> locationList);
    void createCourierCompany(Scanner scanner, List<CourierCompany>  companyList, List<Employee> employeeList,
                              List<Location> locationList);
    void assignEmployeeLocationCompanyToCourier(Scanner scanner,
                                                Map<Integer, List<Courier>> courierMap, List<CourierCompany> companyList);
    void viewCourierCompanyReport(List<CourierCompany> courierCompanyList);
}
```

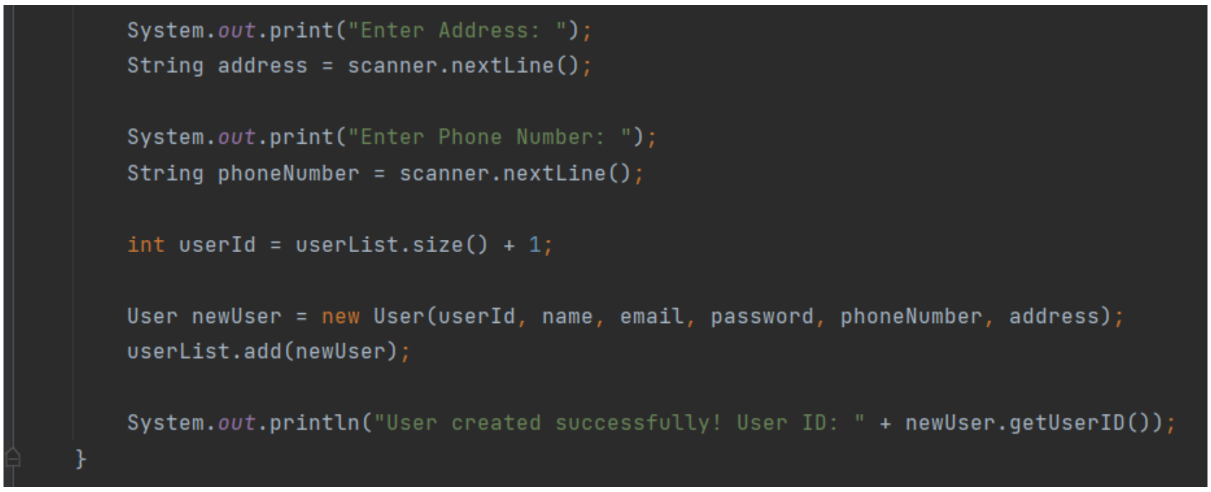
CourierUserServiceImp Methods

1. createUser(Scanner scanner, ArrayList<User> userList)

This method is used to create a new user by collecting user details such as name, email, password, address, and phone number through console input. It generates a unique user ID based on the current size of the user list, creates a User object, adds it to the userList, and confirms the creation to the user. This is the registration function for the courier system.



```java
public class CourierUserServiceImp implements ICourierUserService {

    @Override
    public void createUser(Scanner scanner, ArrayList<User> userList) {
        System.out.print("Enter Name: ");
        String name = scanner.nextLine();

        System.out.print("Enter Email: ");
        String email = scanner.nextLine();

        System.out.print("Enter Password: ");
        String password = scanner.nextLine();
```

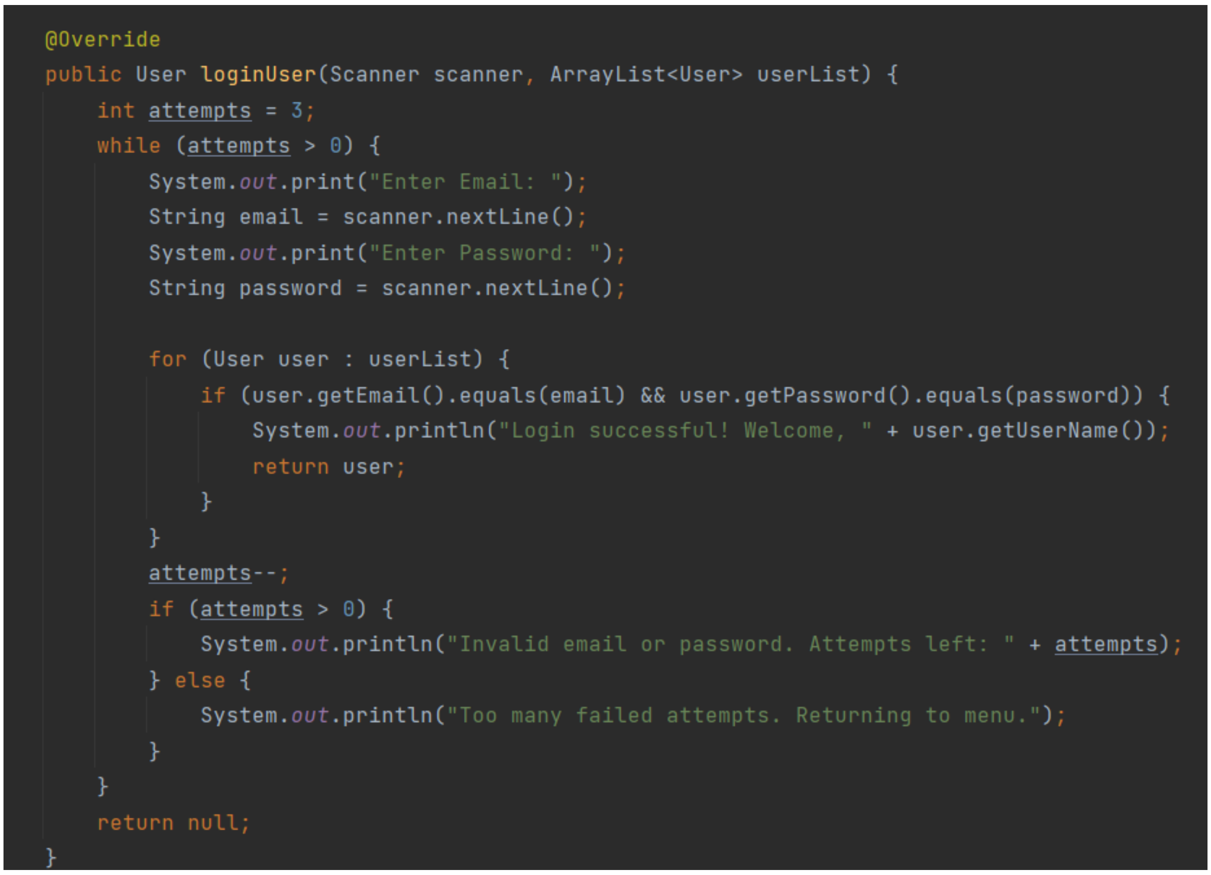


```java
        System.out.print("Enter Address: ");
        String address = scanner.nextLine();

        System.out.print("Enter Phone Number: ");
        String phoneNumber = scanner.nextLine();

        int userId = userList.size() + 1;

        User newUser = new User(userId, name, email, password, phoneNumber, address);
        userList.add(newUser);

        System.out.println("User created successfully! User ID: " + newUser.getUserID());
    }
```

---

2. loginUser(Scanner scanner, ArrayList<User> userList)

This method handles user authentication. It allows a user to log in by verifying their email and password against existing users in the userList. The user gets three attempts to log in correctly; otherwise, the method exits and returns null. Upon successful login, the user object is returned, enabling further personalized actions like booking a courier.

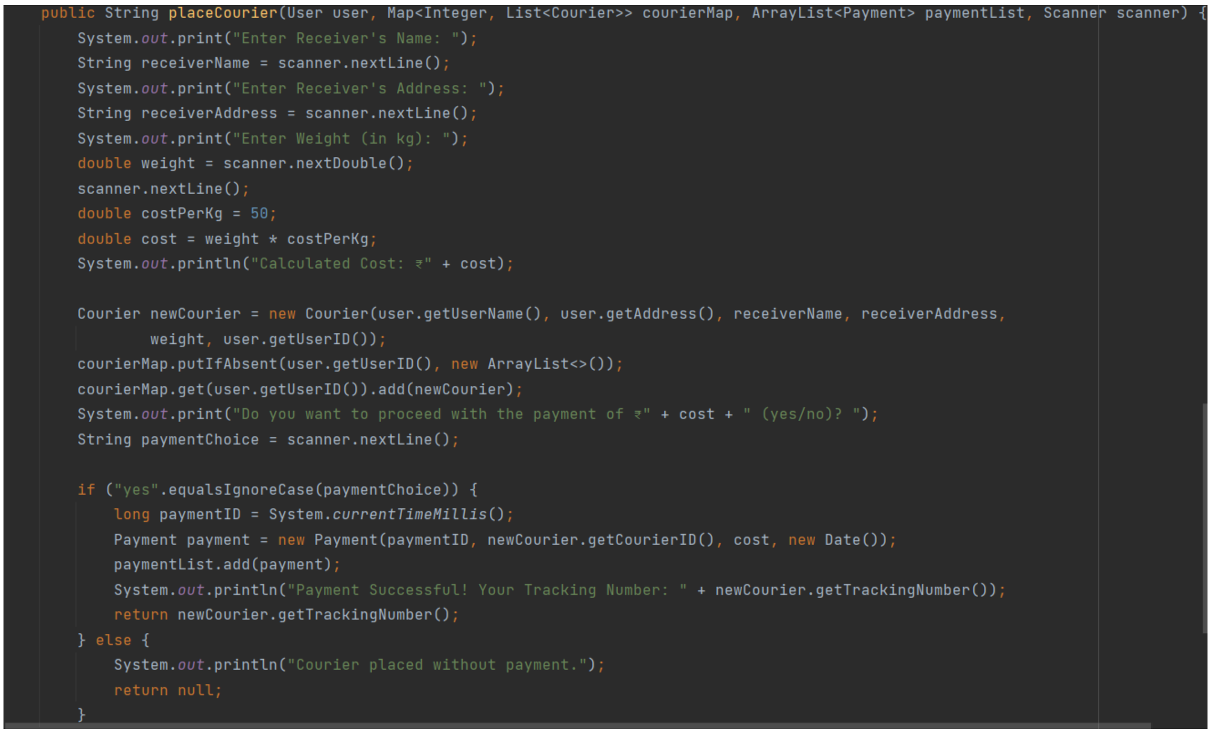


```java
@Override
public User loginUser(Scanner scanner, ArrayList<User> userList) {
    int attempts = 3;
    while (attempts > 0) {
        System.out.print("Enter Email: ");
        String email = scanner.nextLine();
        System.out.print("Enter Password: ");
        String password = scanner.nextLine();

        for (User user : userList) {
            if (user.getEmail().equals(email) && user.getPassword().equals(password)) {
                System.out.println("Login successful! Welcome, " + user.getUserName());
                return user;
            }
        }
        attempts--;
        if (attempts > 0) {
            System.out.println("Invalid email or password. Attempts left: " + attempts);
        } else {
            System.out.println("Too many failed attempts. Returning to menu.");
        }
    }
    return null;
}
```

---

3. placeCourier(User user, Map<Integer, List<Courier>> courierMap, ArrayList<Payment> paymentList, Scanner scanner)

This method enables a logged-in user to place a new courier order. It gathers recipient details and parcel weight, calculates the cost, and creates a new Courier object. The courier is stored in a map associating user IDs with their respective courier lists. It also prompts for payment, and if the user agrees, it generates a payment record and provides a tracking number. This method simulates the end-to-end process of placing and optionally paying for a courier.

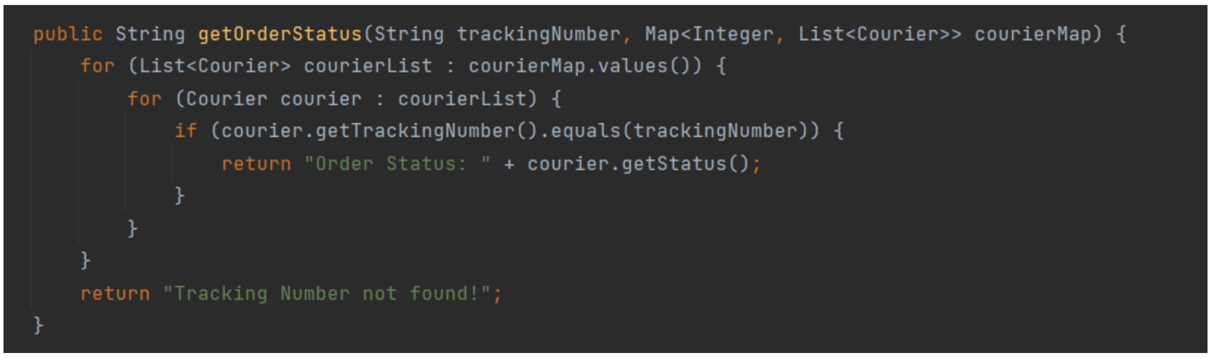


```java
public String placeCourier(User user, Map<Integer, List<Courier>> courierMap, ArrayList<Payment> paymentList, Scanner scanner) {
    System.out.print("Enter Receiver's Name: ");
    String receiverName = scanner.nextLine();
    System.out.print("Enter Receiver's Address: ");
    String receiverAddress = scanner.nextLine();
    System.out.print("Enter Weight (in kg): ");
    double weight = scanner.nextDouble();
    scanner.nextLine();
    double costPerKg = 50;
    double cost = weight * costPerKg;
    System.out.println("Calculated Cost: ₹" + cost);

    Courier newCourier = new Courier(user.getUserName(), user.getAddress(), receiverName, receiverAddress,
            weight, user.getUserID());
    courierMap.putIfAbsent(user.getUserID(), new ArrayList<>());
    courierMap.get(user.getUserID()).add(newCourier);
    System.out.print("Do you want to proceed with the payment of ₹" + cost + " (yes/no)? ");
    String paymentChoice = scanner.nextLine();

    if ("yes".equalsIgnoreCase(paymentChoice)) {
        long paymentID = System.currentTimeMillis();
        Payment payment = new Payment(paymentID, newCourier.getCourierID(), cost, new Date());
        paymentList.add(payment);
        System.out.println("Payment Successful! Your Tracking Number: " + newCourier.getTrackingNumber());
        return newCourier.getTrackingNumber();
    } else {
        System.out.println("Courier placed without payment.");
        return null;
    }
```

---

4. getOrderStatus(String trackingNumber, Map<Integer, List<Courier>> courierMap)

This method retrieves the status of a courier order based on the tracking number. It searches through all the couriers stored in the map and returns the current delivery status if a match is found. If the tracking number doesn't exist, it notifies the user accordingly. It helps users monitor their courier progress.

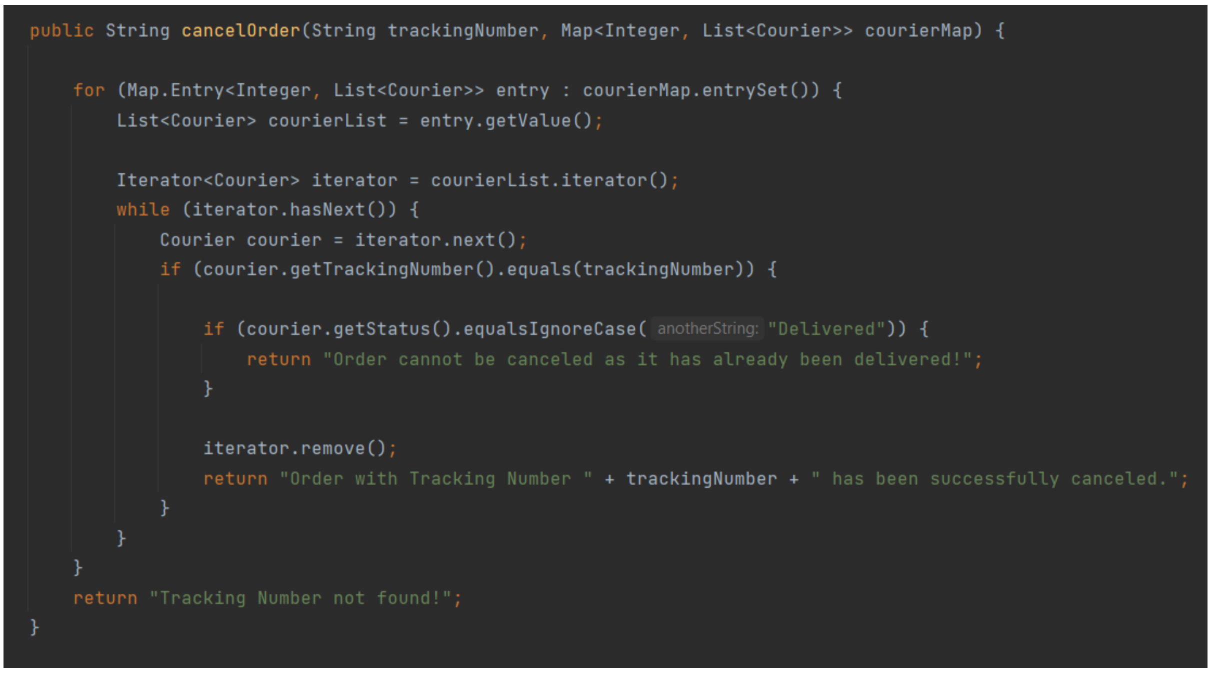


```java
public String getOrderStatus(String trackingNumber, Map<Integer, List<Courier>> courierMap) {
    for (List<Courier> courierList : courierMap.values()) {
        for (Courier courier : courierList) {
            if (courier.getTrackingNumber().equals(trackingNumber)) {
                return "Order Status: " + courier.getStatus();
            }
        }
    }
    return "Tracking Number not found!";
}
```

---

5. cancelOrder(String trackingNumber, Map<Integer, List<Courier>> courierMap)

This method allows a user to cancel a courier order using the tracking number. It checks through the map of courier lists, ensures the order hasn't already been delivered, and then removes it if eligible. It informs the user whether the cancellation was successful or if the operation is invalid. This helps users manage or withdraw courier requests as needed.

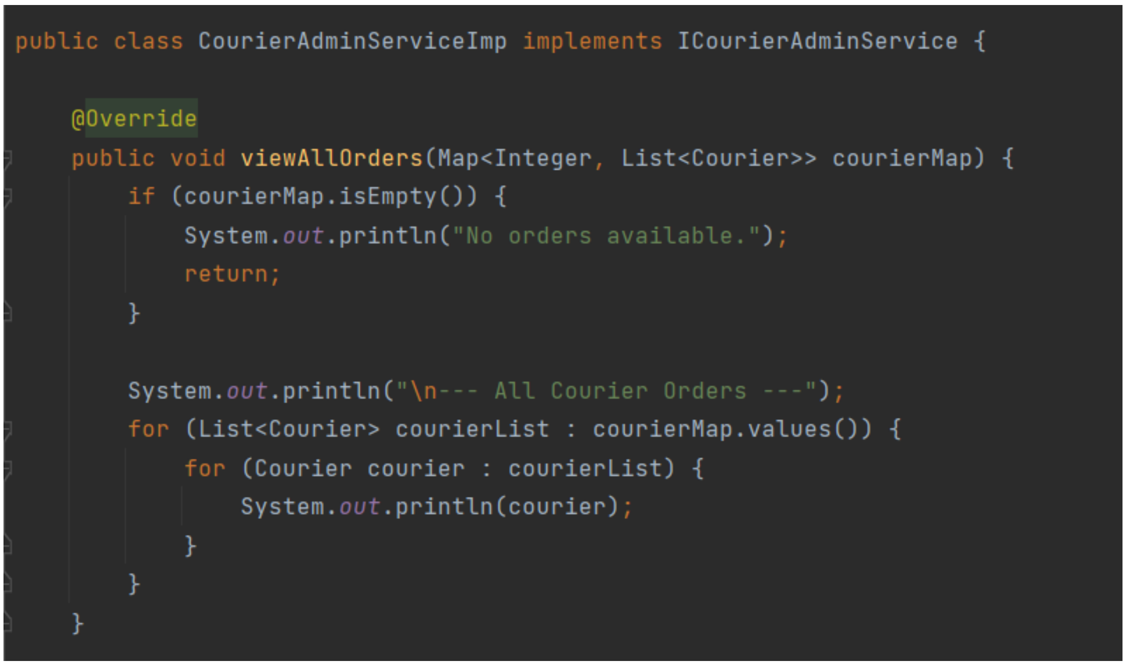


```java
public String cancelOrder(String trackingNumber, Map<Integer, List<Courier>> courierMap) {

    for (Map.Entry<Integer, List<Courier>> entry : courierMap.entrySet()) {
        List<Courier> courierList = entry.getValue();

        Iterator<Courier> iterator = courierList.iterator();
        while (iterator.hasNext()) {
            Courier courier = iterator.next();
            if (courier.getTrackingNumber().equals(trackingNumber)) {

                if (courier.getStatus().equalsIgnoreCase( anotherString: "Delivered")) {
                    return "Order cannot be canceled as it has already been delivered!";
                }

                iterator.remove();
                return "Order with Tracking Number " + trackingNumber + " has been successfully canceled.";
            }
        }
    }
    return "Tracking Number not found!";
}
```

CourierAdminServiceImp Methods

1. viewAllOrders(Map<Integer, List<Courier>> courierMap)

This method displays all courier orders stored in the system. It takes a map where the key is an integer (possibly customer ID or location ID), and the value is a list of courier orders. If there are no orders, it prints a message saying so. Otherwise, it iterates through each list in the map and prints details of each courier object.

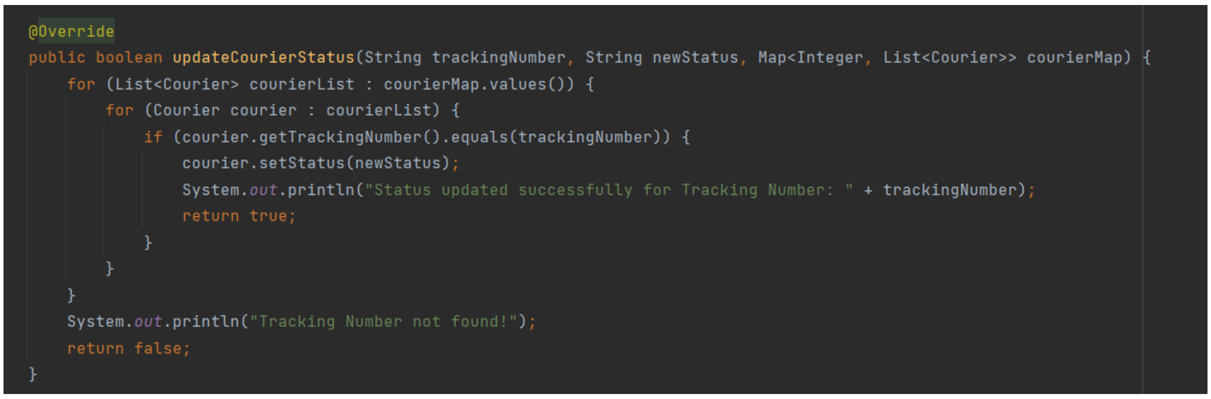


```java
public class CourierAdminServiceImp implements ICourierAdminService {

    @Override
    public void viewAllOrders(Map<Integer, List<Courier>> courierMap) {
        if (courierMap.isEmpty()) {
            System.out.println("No orders available.");
            return;
        }

        System.out.println("\n--- All Courier Orders ---");
        for (List<Courier> courierList : courierMap.values()) {
            for (Courier courier : courierList) {
                System.out.println(courier);
            }
        }
    }
```

---

2. updateCourierStatus(String trackingNumber, String newStatus, Map<Integer, List<Courier>> courierMap)

This method is used to update the status of a courier order based on its tracking number. It searches through all courier orders in the provided map and if it finds the order with the matching tracking number, it updates the status to the new one provided and returns true. If the order is not found, it prints an error and returns false.

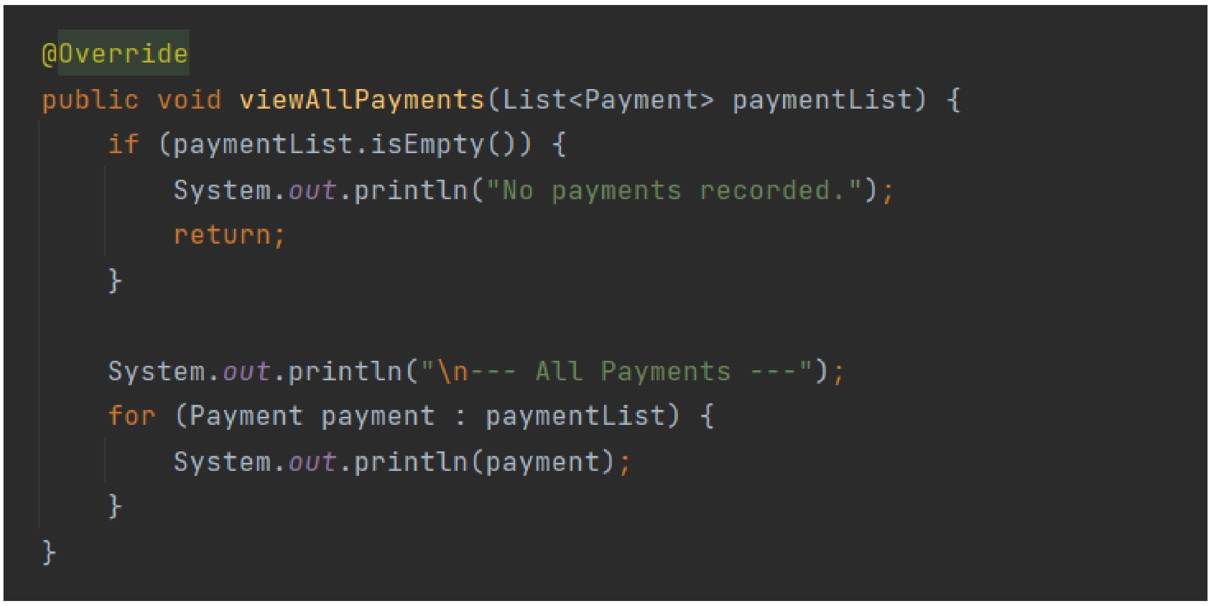


```java
@Override
public boolean updateCourierStatus(String trackingNumber, String newStatus, Map<Integer, List<Courier>> courierMap) {
    for (List<Courier> courierList : courierMap.values()) {
        for (Courier courier : courierList) {
            if (courier.getTrackingNumber().equals(trackingNumber)) {
                courier.setStatus(newStatus);
                System.out.println("Status updated successfully for Tracking Number: " + trackingNumber);
                return true;
            }
        }
    }
    System.out.println("Tracking Number not found!");
    return false;
}
```

---

3. viewAllPayments(List<Payment> paymentList)

This method displays all payment transactions that have occurred in the system. It accepts a list of Payment objects and prints each one. If the list is empty, it prints a message indicating that no payments have been recorded yet.



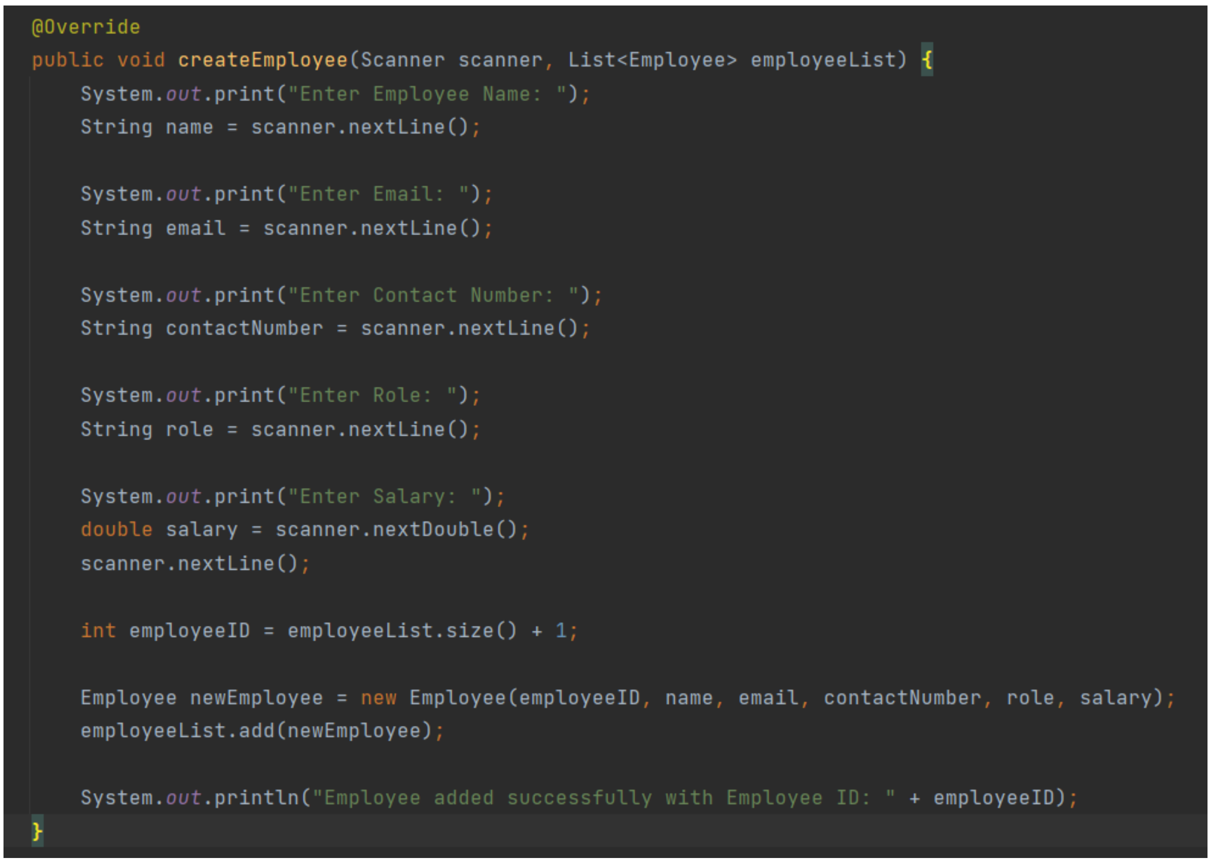
```java
@Override
public void viewAllPayments(List<Payment> paymentList) {
    if (paymentList.isEmpty()) {
        System.out.println("No payments recorded.");
        return;
    }

    System.out.println("\n--- All Payments ---");
    for (Payment payment : paymentList) {
        System.out.println(payment);
    }
}
```

---

4. createEmployee(Scanner scanner, List<Employee> employeeList)

This method facilitates the creation of a new employee by prompting the admin to enter the employee's name, email, contact number, role, and salary. It then generates a unique employee

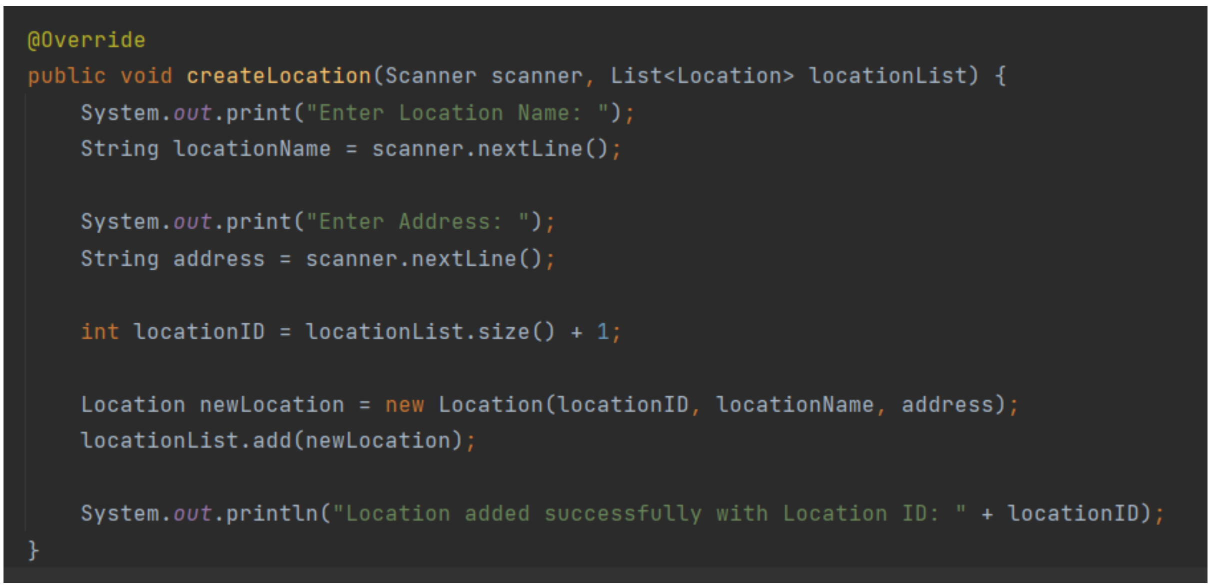
ID and adds the new Employee object to the provided list. It provides confirmation upon successful addition.



```java
@Override
public void createEmployee(Scanner scanner, List<Employee> employeeList) {
    System.out.print("Enter Employee Name: ");
    String name = scanner.nextLine();

    System.out.print("Enter Email: ");
    String email = scanner.nextLine();

    System.out.print("Enter Contact Number: ");
    String contactNumber = scanner.nextLine();

    System.out.print("Enter Role: ");
    String role = scanner.nextLine();

    System.out.print("Enter Salary: ");
    double salary = scanner.nextDouble();
    scanner.nextLine();

    int employeeID = employeeList.size() + 1;

    Employee newEmployee = new Employee(employeeID, name, email, contactNumber, role, salary);
    employeeList.add(newEmployee);

    System.out.println("Employee added successfully with Employee ID: " + employeeID);
}
```

5. createLocation(Scanner scanner, List<Location> locationList)

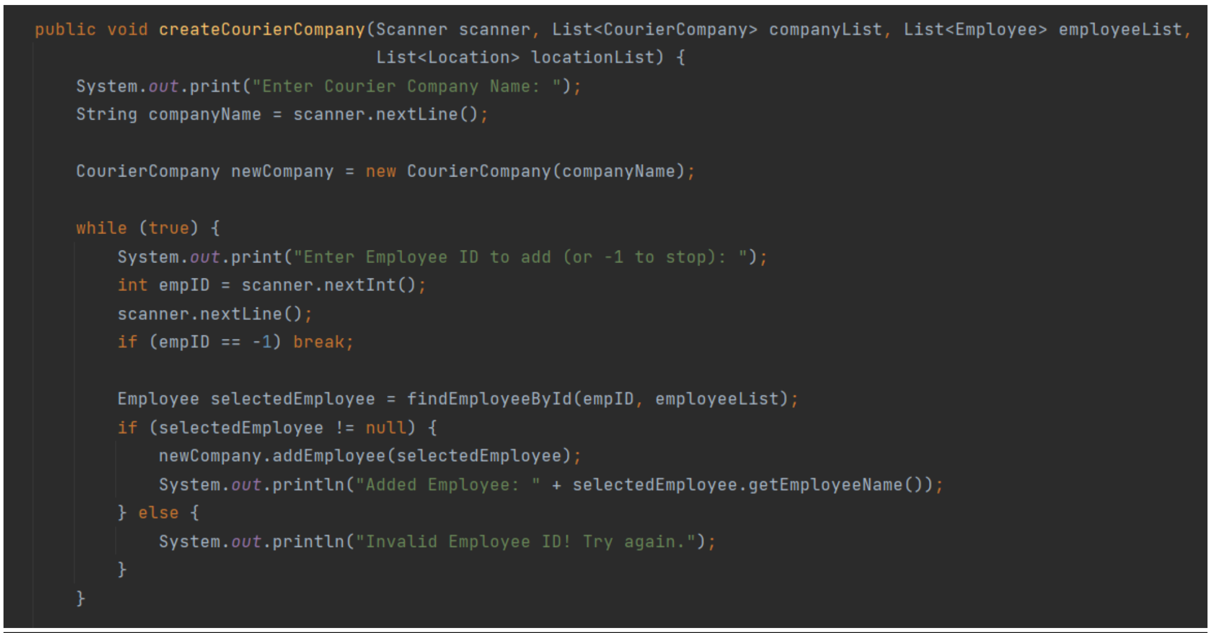
This method adds a new location into the system. The admin is prompted to input the location name and address. It automatically generates a unique location ID and creates a Location object, which is then added to the given location list.



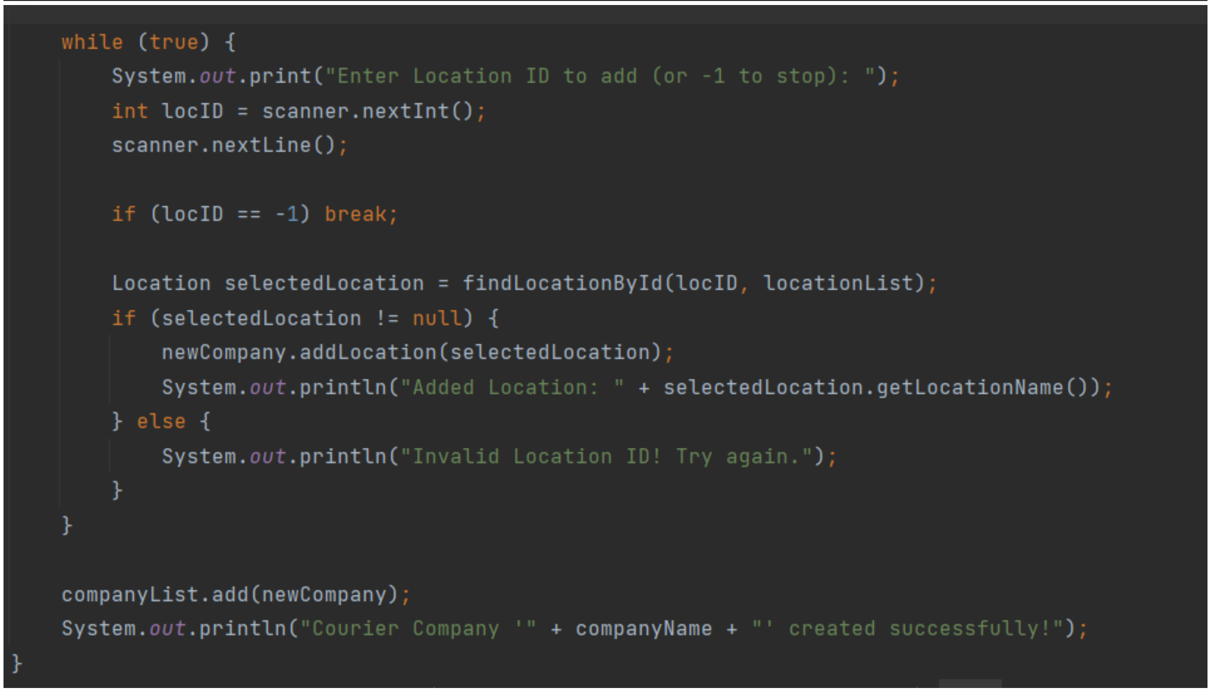
```java
@Override
public void createLocation(Scanner scanner, List<Location> locationList) {
    System.out.print("Enter Location Name: ");
    String locationName = scanner.nextLine();

    System.out.print("Enter Address: ");
    String address = scanner.nextLine();

    int locationID = locationList.size() + 1;

    Location newLocation = new Location(locationID, locationName, address);
    locationList.add(newLocation);

    System.out.println("Location added successfully with Location ID: " + locationID);
}
```

6. createCourierCompany(Scanner scanner, List<CourierCompany> companyList, List<Employee> employeeList, List<Location> locationList)

This method is responsible for creating a new courier company. The admin inputs the company name and selects multiple employees and locations to associate with the company. The method allows iterative selection of employees and locations and adds the completed company to the system's list of courier companies.



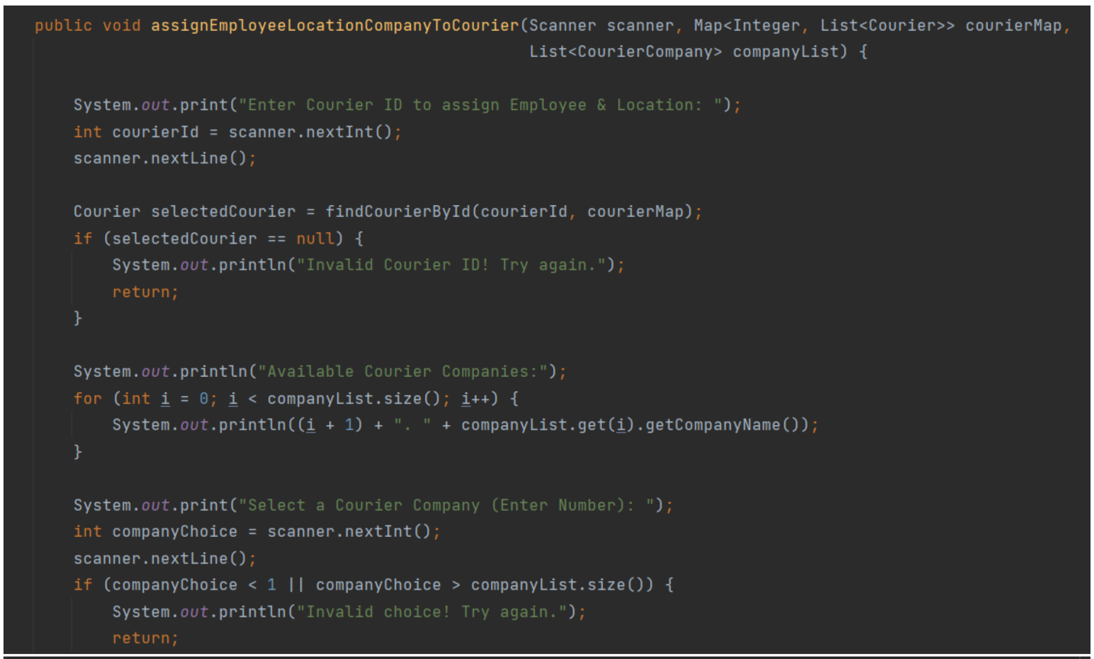
```java
public void createCourierCompany(Scanner scanner, List<CourierCompany> companyList, List<Employee> employeeList,
                                 List<Location> locationList) {
    System.out.print("Enter Courier Company Name: ");
    String companyName = scanner.nextLine();

    CourierCompany newCompany = new CourierCompany(companyName);

    while (true) {
        System.out.print("Enter Employee ID to add (or -1 to stop): ");
        int empID = scanner.nextInt();
        scanner.nextLine();
        if (empID == -1) break;

        Employee selectedEmployee = findEmployeeById(empID, employeeList);
        if (selectedEmployee != null) {
            newCompany.addEmployee(selectedEmployee);
            System.out.println("Added Employee: " + selectedEmployee.getEmployeeName());
        } else {
            System.out.println("Invalid Employee ID! Try again.");
        }
    }
```



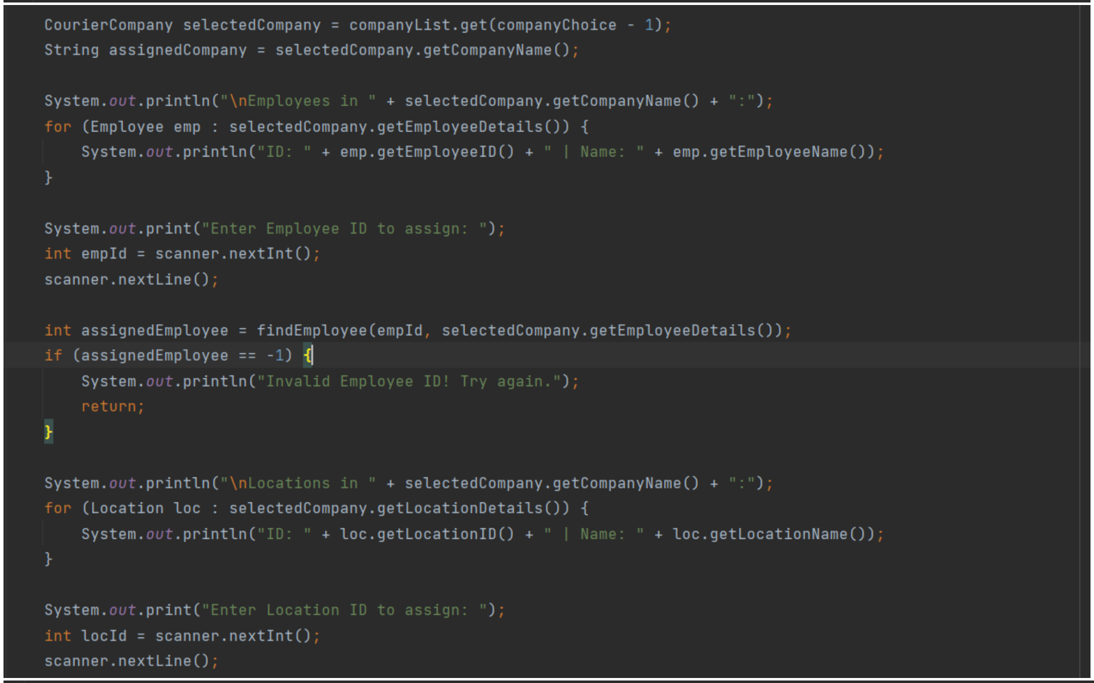
```java
    while (true) {
        System.out.print("Enter Location ID to add (or -1 to stop): ");
        int locID = scanner.nextInt();
        scanner.nextLine();

        if (locID == -1) break;

        Location selectedLocation = findLocationById(locID, locationList);
        if (selectedLocation != null) {
            newCompany.addLocation(selectedLocation);
            System.out.println("Added Location: " + selectedLocation.getLocationName());
        } else {
            System.out.println("Invalid Location ID! Try again.");
        }
    }

    companyList.add(newCompany);
    System.out.println("Courier Company '" + companyName + "' created successfully!");
}
```

7. assignEmployeeLocationCompanyToCourier(Scanner scanner, Map<Integer, List<Courier>> courierMap, List<CourierCompany> companyList)

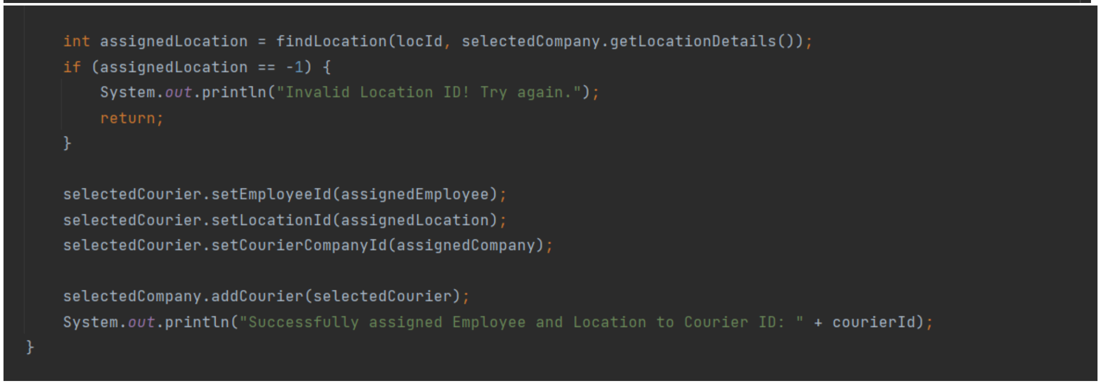
This method assigns an employee, location, and courier company to a specific courier. The admin provides a courier ID and selects the associated company, employee, and location from existing lists. After successful selection, the method sets these attributes in the Courier object and adds the courier to the company's list.



```java
public void assignEmployeeLocationCompanyToCourier(Scanner scanner, Map<Integer, List<Courier>> courierMap,
                                                   List<CourierCompany> companyList) {

    System.out.print("Enter Courier ID to assign Employee & Location: ");
    int courierId = scanner.nextInt();
    scanner.nextLine();

    Courier selectedCourier = findCourierById(courierId, courierMap);
    if (selectedCourier == null) {
        System.out.println("Invalid Courier ID! Try again.");
        return;
    }

    System.out.println("Available Courier Companies:");
    for (int i = 0; i < companyList.size(); i++) {
        System.out.println((i + 1) + ". " + companyList.get(i).getCompanyName());
    }

    System.out.print("Select a Courier Company (Enter Number): ");
    int companyChoice = scanner.nextInt();
    scanner.nextLine();
    if (companyChoice < 1 || companyChoice > companyList.size()) {
        System.out.println("Invalid choice! Try again.");
        return;
```



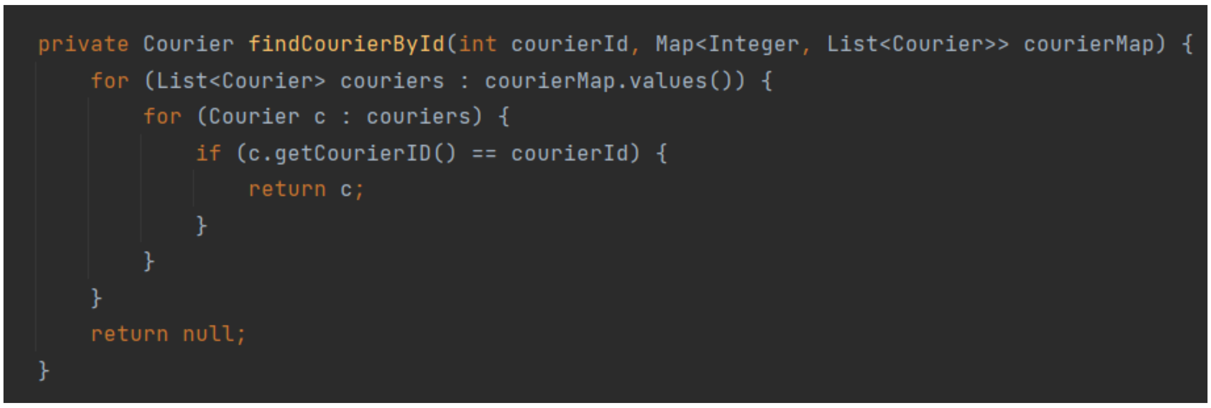
```java
CourierCompany selectedCompany = companyList.get(companyChoice - 1);
String assignedCompany = selectedCompany.getCompanyName();

System.out.println("\nEmployees in " + selectedCompany.getCompanyName() + ":");
for (Employee emp : selectedCompany.getEmployeeDetails()) {
    System.out.println("ID: " + emp.getEmployeeID() + " | Name: " + emp.getEmployeeName());
}

System.out.print("Enter Employee ID to assign: ");
int empId = scanner.nextInt();
scanner.nextLine();

int assignedEmployee = findEmployee(empId, selectedCompany.getEmployeeDetails());
if (assignedEmployee == -1) {
    System.out.println("Invalid Employee ID! Try again.");
    return;
}

System.out.println("\nLocations in " + selectedCompany.getCompanyName() + ":");
for (Location loc : selectedCompany.getLocationDetails()) {
    System.out.println("ID: " + loc.getLocationID() + " | Name: " + loc.getLocationName());
}

System.out.print("Enter Location ID to assign: ");
int locId = scanner.nextInt();
scanner.nextLine();
```



```java
    int assignedLocation = findLocation(locId, selectedCompany.getLocationDetails());
    if (assignedLocation == -1) {
        System.out.println("Invalid Location ID! Try again.");
        return;
    }

    selectedCourier.setEmployeeId(assignedEmployee);
    selectedCourier.setLocationId(assignedLocation);
    selectedCourier.setCourierCompanyId(assignedCompany);

    selectedCompany.addCourier(selectedCourier);
    System.out.println("Successfully assigned Employee and Location to Courier ID: " + courierId);
}
```

8. findCourierById(int courierId, Map<Integer, List<Courier>> courierMap)

This is a helper method that searches for and returns a Courier object by its courier ID from a map of couriers. It returns the courier object if found, or null otherwise.

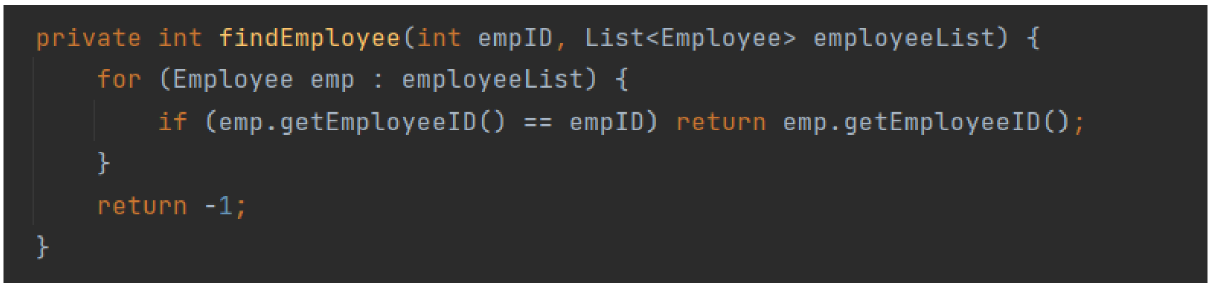


```java
private Courier findCourierById(int courierId, Map<Integer, List<Courier>> courierMap) {
    for (List<Courier> couriers : courierMap.values()) {
        for (Courier c : couriers) {
            if (c.getCourierID() == courierId) {
                return c;
            }
        }
    }
    return null;
}
```

---

9. findEmployee(int empID, List<Employee> employeeList)

This is a utility method that searches for an employee by their ID from a list. If found, it returns the employee ID; otherwise, it returns -1 indicating the employee was not found.

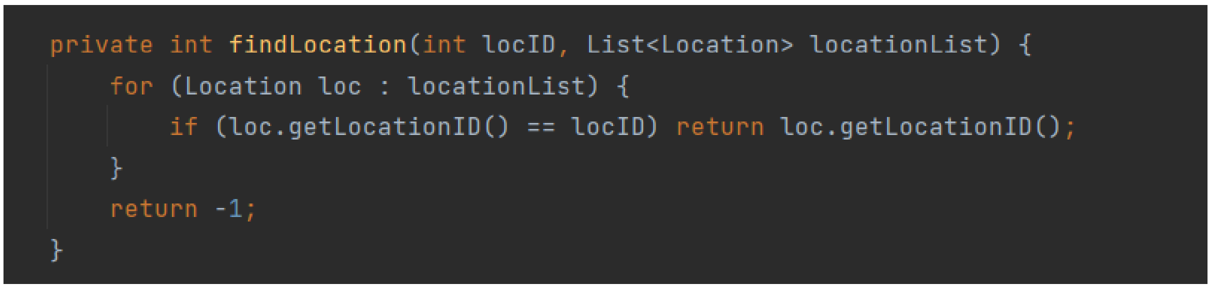


```java
private int findEmployee(int empID, List<Employee> employeeList) {
    for (Employee emp : employeeList) {
        if (emp.getEmployeeID() == empID) return emp.getEmployeeID();
    }
    return -1;
}
```

---

10. findLocation(int locID, List<Location> locationList)

Similar to findEmployee, this helper method looks for a location by its ID in a list of locations. It returns the location ID if found or -1 if not found.

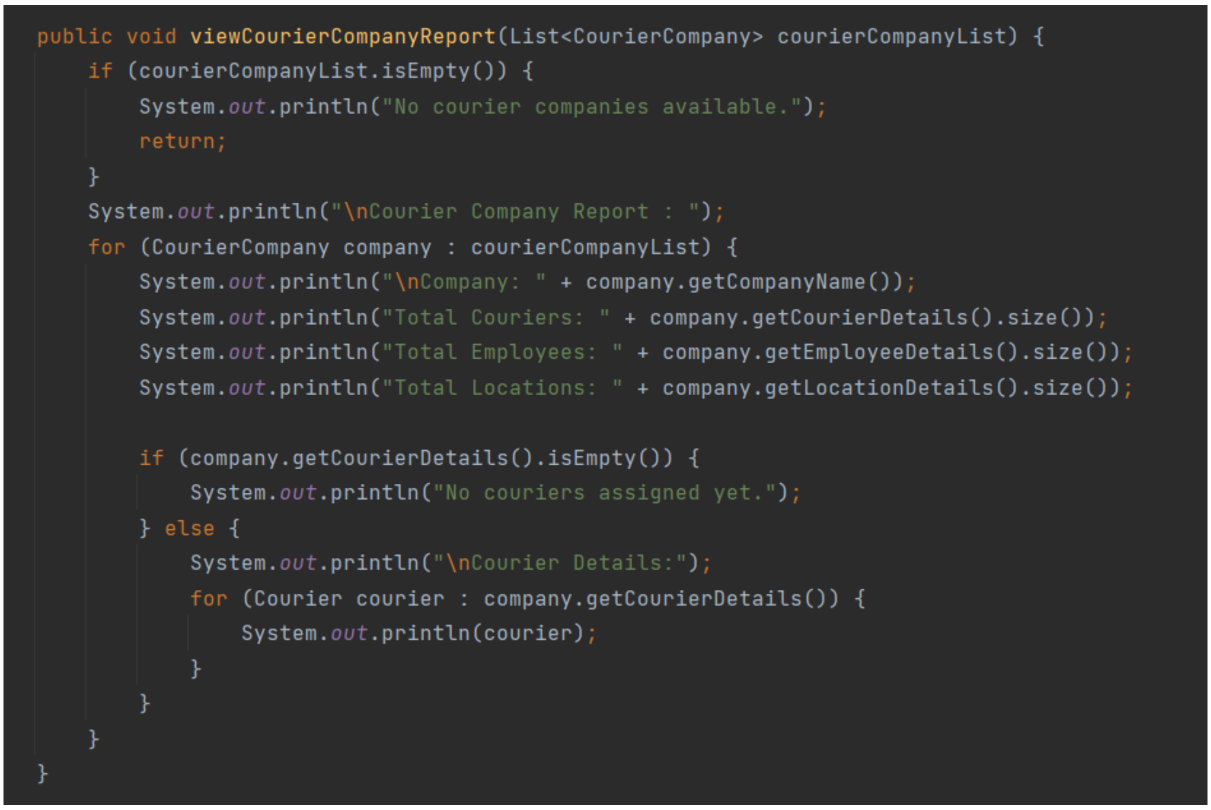


```java
private int findLocation(int locID, List<Location> locationList) {
    for (Location loc : locationList) {
        if (loc.getLocationID() == locID) return loc.getLocationID();
    }
    return -1;
}
```

---

11. viewCourierCompanyReport(List<CourierCompany> courierCompanyList)

This method prints a detailed report of all courier companies in the system. For each company, it shows the company name, total couriers, total employees, and total locations. It also lists detailed courier information if any couriers are assigned to that company, providing an overview for administrative review.



```java
public void viewCourierCompanyReport(List<CourierCompany> courierCompanyList) {
    if (courierCompanyList.isEmpty()) {
        System.out.println("No courier companies available.");
        return;
    }
    System.out.println("\nCourier Company Report : ");
    for (CourierCompany company : courierCompanyList) {
        System.out.println("\nCompany: " + company.getCompanyName());
        System.out.println("Total Couriers: " + company.getCourierDetails().size());
        System.out.println("Total Employees: " + company.getEmployeeDetails().size());
        System.out.println("Total Locations: " + company.getLocationDetails().size());

        if (company.getCourierDetails().isEmpty()) {
            System.out.println("No couriers assigned yet.");
        } else {
            System.out.println("\nCourier Details:");
            for (Courier courier : company.getCourierDetails()) {
                System.out.println(courier);
            }
        }
    }
}
```

Conclusion

In Task 6, I established a robust and well-structured **service layer** for our Courier Management System by defining and implementing the ICourierUserService and ICourierAdminService interfaces. These interfaces serve as **clear contracts**, promoting:

- **Modular design**
- **Separation of concerns**
- **Code reusability**
- **System scalability**

The CourierUserServiceImp class focuses on **user-side interactions** such as placing and tracking couriers, while the CourierAdminServiceImp class handles **administrative functionalities** like employee management, company creation, and courier assignments. This clean architecture ensures our system is maintainable and adaptable for future enhancements.
