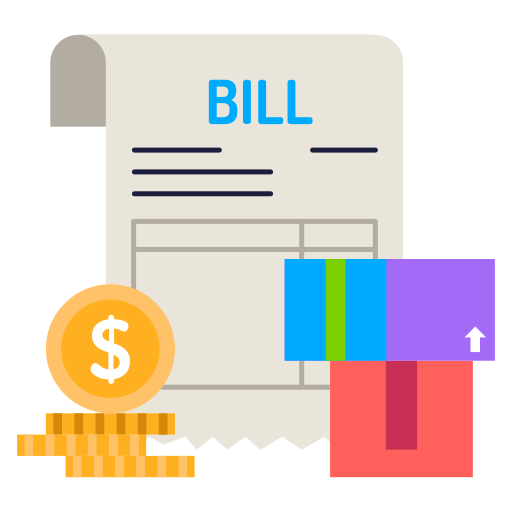
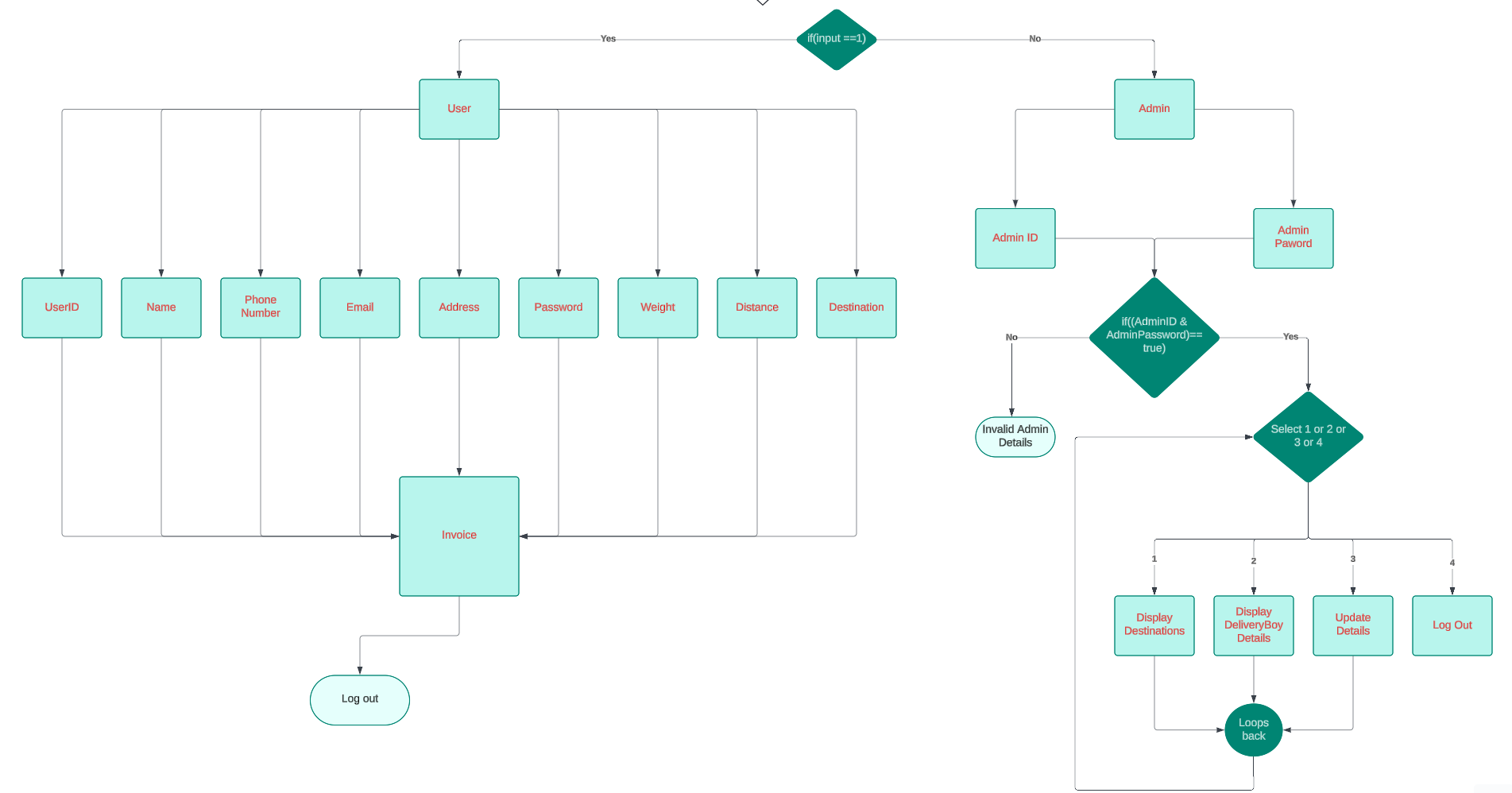
**Logistic System**

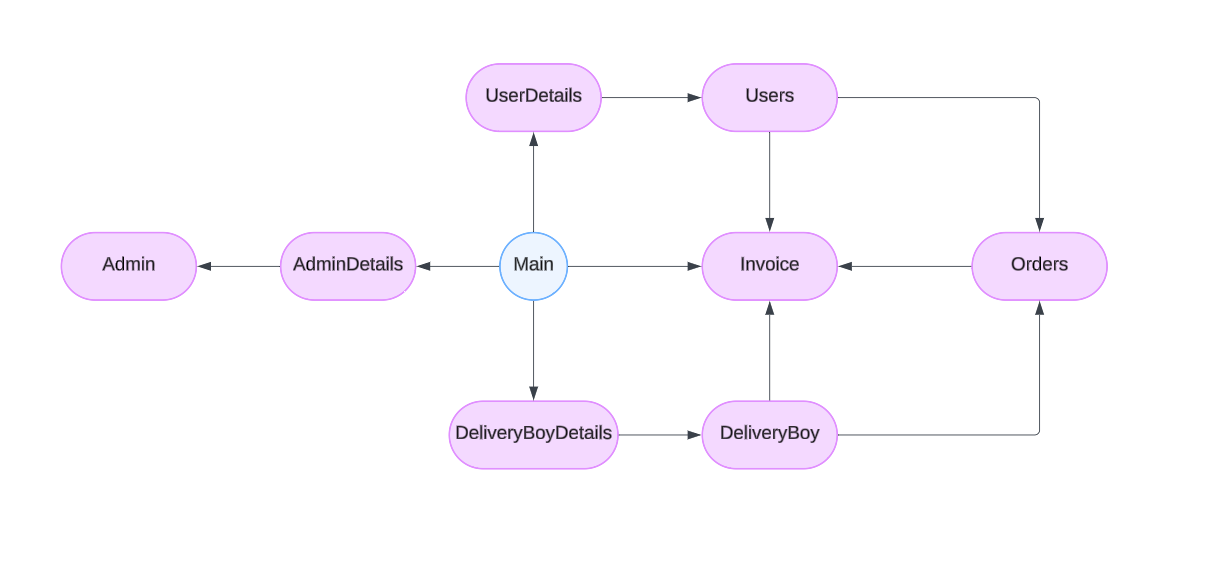
**-------------------------------------------------------------------------------------------------------------------------------------------------------------------**

A logistics system is a comprehensive framework designed to manage the movement of goods, services, and information from their point of origin to the final destination efficiently. It involves coordinating and optimizing various processes, including transportation, warehousing, inventory management, and delivery, to ensure customer satisfaction while minimizing costs.



**Low-Level Design of a Logistics System**

The **Low-Level Design (LLD)** of a logistics system focuses on the detailed implementation of individual components and their interactions. It defines how specific features or modules within the system are structured, including classes, functions, and workflows, ensuring a seamless execution of logistics operations.

**Admin.java**

This code defines a Java class called Admin, which represents an administrator with an ID and a password. The class contains attributes for storing the admin's ID and password, as well as methods to interact with those attributes. Here's a detailed explanation of the class:

**Class Definition: Admin**

The class is used to model an admin entity, which can have an adminId and an adminPassword. The Admin class contains:

* **Instance Variables**: adminId and adminPassword
* **Constructor**: To initialize the object with an adminId and adminPassword
* **Methods**: To get and set the values of adminId and adminPassword, as well as a method to validate the admin's credentials.

**1. Instance Variables (Fields)**

private String adminId;

private String adminPassword;

* **adminId**: A String variable to store the administrator's ID (e.g., a username or email).
* **adminPassword**: A String variable to store the administrator's password.

These fields are **private**, meaning they can only be accessed directly within the Admin class.

**2. Constructor**

public Admin(String adminId, String adminPassword) {

    this.adminId = adminId;

    this.adminPassword = adminPassword;

}

* This is a **constructor** for the Admin class. A constructor is called when you create an instance (object) of the class.
* The constructor takes two parameters:
  + adminId: The admin's ID (e.g., username).
  + adminPassword: The admin's password.
* Inside the constructor, the **this keyword** is used to refer to the current instance of the class. It is used to differentiate the class fields from the constructor parameters.
* This constructor initializes the object with the given adminId and adminPassword values.

**3. validateAdmin Method**

public boolean validateAdmin(String adminId, String adminPassword) {

    return this.adminId.equals(adminId) &&

this.adminPassword.equals(adminPassword);

}

* This method checks whether the provided admin credentials match the stored credentials.
* The method:
  + Takes two parameters: adminId and adminPassword.
  + It compares the passed adminId and adminPassword with the current object's adminId and adminPassword using the .equals() method, which compares the actual content of the strings (not their memory addresses).
  + If both the admin ID and password match, the method returns true; otherwise, it returns false.
* This is useful for authentication purposes, allowing you to validate whether a given adminId and adminPassword match the admin's actual credentials.

**4. Getter and Setter Methods**

Java follows the **JavaBeans** convention, where getters and setters are used to access and modify private variables.

**Getter for adminId**

public String getAdminId() {

    return adminId;

}

* This method returns the value of the adminId.
* It allows external code to retrieve the adminId without directly accessing the private field.

**Setter for adminId**

public void setAdminId(String adminId) {

    this.adminId = adminId;

}

* This method allows you to set the value of adminId from outside the class.
* It takes a String argument (adminId) and assigns it to the instance variable adminId.

**Getter for adminPassword**

public String getAdminPassword() {

  return adminPassword;

}

* This method returns the value of the adminPassword.
* It allows external code to retrieve the adminPassword without directly accessing the private field.

**Setter for adminPassword**

public void setAdminPassword(String adminPassword) {

    this.adminPassword = adminPassword;

}

* This method allows you to set the value of adminPassword from outside the class.
* It takes a String argument (adminPassword) and assigns it to the instance variable adminPassword.

**AdminDetails.java**

This code defines a class AdminDetails that manages a list of Admin objects. The class is used for registering new administrators, logging in with admin credentials, and accessing the list of admins. Let's break down the code in detail:

**Class Definition: AdminDetails**

This class is responsible for maintaining and managing a collection of Admin objects. It supports operations like registering admins, logging in with admin credentials, and getting or updating the list of admins.

**1. Instance Variable: adminList**

private List<Admin> adminList;

* adminList is a List that holds Admin objects. It is declared as a private field, meaning it can only be accessed within the AdminDetails class.

**2. Constructor: AdminDetails()**

public AdminDetails() {

    adminList = new ArrayList<>();

    registerAdmin("admin1", "password1");

    registerAdmin("admin2", "password2");

}

* **Purpose**: The constructor initializes the adminList as an empty ArrayList and registers two default admins with hardcoded IDs and passwords (admin1 and admin2).
* The constructor calls the registerAdmin method twice to add two predefined admins to the adminList.

**3. Method: registerAdmin(String adminId, String adminPassword)**

public void registerAdmin(String adminId, String adminPassword) {

    Admin admin = new Admin(adminId, adminPassword);

    adminList.add(admin);

}

* **Purpose**: This method allows the registration of new admins.
* It creates a new Admin object using the provided adminId and adminPassword and then adds this new Admin to the adminList.
* The registerAdmin method is called in the constructor to register the two default admins.

**4. Method: loginAdmin(String adminId, String password)**

public Admin loginAdmin(String adminId, String password) {

    for (Admin admin : adminList) {

        if (admin.getAdminId().equals(adminId) &&

admin.validateAdmin(adminId, password)) {

            return admin;

        }

    }

    return null;

}

* **Purpose**: This method attempts to authenticate an admin by comparing the provided adminId and password with the stored credentials in the adminList.
* It iterates over the adminList and checks if any Admin object's adminId matches the input adminId and whether the validateAdmin method returns true for the provided password.
* If a match is found, the Admin object is returned.
* If no valid admin is found after checking all admins, it returns null, indicating failed login.

**5. Getter and Setter for adminList**

**Getter: getAdminList()**

public List<Admin> getAdminList() {

    return adminList;

}

* **Purpose**: This method returns the list of all Admin objects stored in the adminList. It allows external code to access the list of admins.

**Setter: setAdminList(List<Admin> adminList)**

public void setAdminList(List<Admin> adminList) {

    this.adminList = adminList;

}

* **Purpose**: This method allows external code to modify the adminList. It takes a new List<Admin> and assigns it to the adminList field.

**Users.java**

The provided code defines a Users class, which models a user account in a system. The class includes several attributes related to the user, such as account information, personal details, and methods for validation and data manipulation.

**Detailed Explanation:**

**1. Class Declaration: Users**

public class Users {

    private String accountId;

    private String name;

    private String mobileNo;

    private String email;

    private String password;

    private String address;

}

* **Purpose**: The Users class represents a user in the system with personal details (like name, mobile number, email, password, and address), and it includes an accountId to uniquely identify each user.

**2. Constructor: Users(String accountId, String name, String mobileNo, String email, String password, String address)**

public Users(String accountId, String name, String mobileNo, String email, String password, String address) {

    this.accountId = accountId;

    this.name = name;

    this.mobileNo = mobileNo;

    this.email = email;

    this.password = password;

    this.address = address;

}

* **Purpose**: This constructor initializes the Users object with the provided values for accountId, name, mobileNo, email, password, and address.
* When a new Users object is created, the constructor ensures that all the user details are set upon instantiation.

**3. Method: validatePassword(String password)**

public boolean validatePassword(String password) {

    return this.password.equals(password);

}

* **Purpose**: This method checks if the provided password matches the stored password of the Users object.
* It returns true if the password is correct and false otherwise. This can be used during the login process to validate the user's credentials.

**4. Getters and Setters for each field**

For each field (i.e., accountId, name, mobileNo, email, password, address), the class provides **getter** and **setter** methods.

For example:

public String getAccountId() {

    return accountId;

}

public void setAccountId(String accountId) {

    this.accountId = accountId;

}

* **Getter Methods**: The getter methods retrieve the current value of each attribute.
  + For example, getAccountId() returns the value of accountId, and getName() returns the value of name.
* **Setter Methods**: The setter methods allow you to modify the value of each attribute.
  + For example, setAccountId(String accountId) allows you to update the accountId of the user, and setEmail(String email) allows you to update the user's email.

**5. Explanation of Fields:**

* **accountId**: A unique identifier for the user account, which can be used to distinguish between different users.
* **name**: The name of the user.
* **mobileNo**: The mobile number of the user.
* **email**: The email address of the user.
* **password**: The password for the user's account.
* **address**: The address associated with the user.

**UsersDetails.java**

The UsersDetails class is designed to manage a list of Users objects. It allows for user registration, login, retrieval of user details, and access to a list of all users.

**Detailed Explanation:**

**1. Class Declaration: UsersDetails**

public class UsersDetails {

    private final List<Users> usersList;

}

* **Purpose**: The UsersDetails class manages a list of Users. It provides functionality to register users, allow them to log in, retrieve individual user details, and access all users.
* The usersList is a private field that stores a list of Users objects. This list is initialized in the constructor and is **immutable** (i.e., the reference to it cannot be changed after initialization) because the usersList is declared as final.

**2. Constructor: UsersDetails()**

public UsersDetails() {

    usersList = new ArrayList<>();

}

* **Purpose**: This constructor initializes the usersList as an empty ArrayList. This allows the UsersDetails object to hold the list of Users objects. This method is called when an instance of UsersDetails is created.

**3. Method: registerUser(String accountId, String name, String mobileNo, String email, String password, String address)**

public void registerUser(String accountId, String name, String mobileNo, String email, String password, String address) {

    Users user = new Users(accountId, name, mobileNo, email, password, address);

    usersList.add(user);

}

* **Purpose**: This method registers a new user by creating a new Users object using the provided details and adds the newly created Users object to the usersList.
* The method allows the system to store user details after they sign up, ensuring they are part of the usersList.

**4. Method: loginUser(String accountId, String password)**

public Users loginUser(String accountId, String password) {

    for (Users user : usersList) {

        if (user.getAccountId().equals(accountId) && user.validatePassword(password)) {

            return user;

        }

    }

    return null;

}

* **Purpose**: This method authenticates a user based on their accountId and password.
* It loops through the usersList, checking each user to see if their accountId matches and if their password is validated by the validatePassword() method of the Users class.
* If the account exists and the password is correct, the method returns the corresponding Users object. Otherwise, it returns null if no match is found.

**5. Method: getUser(String userId)**

public Users getUser(String userId) {

    for (Users user : usersList) {

        if (user.getAccountId().equals(userId)) {

            return user;

        }

    }

    return null;

}

* **Purpose**: This method retrieves a user from the usersList based on their accountId.
* It iterates through the usersList, returning the first Users object that has a matching accountId. If no user with the given accountId is found, it returns null.

**6. Method: getUsersList()**

public List<Users> getUsersList() {

    return usersList;

}

* **Purpose**: This method returns the entire list of users (usersList).
* The list is **read-only**, meaning that it can be accessed but not modified directly outside the UsersDetails class.

**DeliveryBoy.java**

The DeliveryBoy class represents a delivery person in a system, encapsulating details about the delivery boy, such as their ID, name, and contact number. The class provides getters and setters to retrieve and modify these details. Here's a detailed breakdown of the code:

**1. Class Declaration: DeliveryBoy**

public class DeliveryBoy {

    private String deliveryBoyId;

    private String name;

    private String contactNumber;

}

* **Purpose**: This class holds the details of a delivery boy, including their unique ID (deliveryBoyId), name (name), and contact number (contactNumber). The fields are declared as private, ensuring encapsulation of the data.

**2. Constructor: DeliveryBoy(String deliveryBoyId, String name, String contactNumber)**

public DeliveryBoy(String deliveryBoyId, String name, String contactNumber) {

    this.deliveryBoyId = deliveryBoyId;

    this.name = name;

    this.contactNumber = contactNumber;

}

* **Purpose**: This constructor initializes the DeliveryBoy object with the provided values for deliveryBoyId, name, and contactNumber. It ensures that these fields are populated when a new DeliveryBoy object is created.

**3. Getter and Setter Methods**

The class has several getter and setter methods to access and modify the private fields. Here’s the explanation for each:

**Getter Methods:**

* **getDeliveryBoyId()**:

public String getDeliveryBoyId() {

    return deliveryBoyId;

}

* **Purpose**: This method returns the deliveryBoyId of the DeliveryBoy. It's used to access the unique ID of the delivery boy.
* **getName()**:

public String getName() {

    return name;

}

* **Purpose**: This method returns the name of the DeliveryBoy.
* **getContactNumber()**:

public String getContactNumber() {

    return contactNumber;

}

* **Purpose**: This method returns the contactNumber of the DeliveryBoy.

**Setter Methods:**

* **setDeliveryBoyId(String deliveryBoyId)**:

public void setDeliveryBoyId(String deliveryBoyId) {

    this.deliveryBoyId = deliveryBoyId;

}

* **Purpose**: This method sets or updates the deliveryBoyId field of the DeliveryBoy object.
* **setName(String name)**:

public void setName(String name) {

    this.name = name;

}

* + **Purpose**: This method sets or updates the name of the DeliveryBoy.
* **setContactNumber(String contactNumber)**:

public void setContactNumber(String contactNumber) {

    this.contactNumber = contactNumber;

}

* + **Purpose**: This method sets or updates the contactNumber of the DeliveryBoy.

**DeliveryBoyDetails.java**

The DeliveryBoyDetails class is designed to manage and display information about cities, destinations, and their respective delivery boys. Here's a detailed breakdown of the class and its methods:

**Class Declaration and Fields**

public class DeliveryBoyDetails {

    private HashMap<String, DeliveryBoy> deliveryBoyMap;

    private HashMap<Integer, String> cityNumberMap;

}

* **deliveryBoyMap**: A HashMap that associates city names (String) with DeliveryBoy objects.
* **cityNumberMap**: A HashMap that maps unique city numbers (Integer) to city names (String).

**Constructor: DeliveryBoyDetails()**

public DeliveryBoyDetails() {

    deliveryBoyMap = new HashMap<>();

    cityNumberMap = new HashMap<>();

    // Add default destinations and delivery boys

    addDestination("Vijayawada", "Ram", "9087656823");

    ...

}

* Initializes the HashMap fields.
* Pre-populates the maps with default destinations, delivery boy names, and contact numbers using the addDestination() method.

**Method: addDestination()**

public void addDestination(String city, String deliveryBoyName, String contactNumber) {

    int cityNumber = cityNumberMap.size() + 1;

    cityNumberMap.put(cityNumber, city);

    deliveryBoyMap.put(city, new DeliveryBoy(String.valueOf(cityNumber), deliveryBoyName, contactNumber, "Standard"));

}

* **Purpose**: Adds a new city and its associated delivery boy to the maps.
* **Functionality**:
  + Generates a unique cityNumber based on the current size of cityNumberMap.
  + Adds the cityNumber and city to cityNumberMap.
  + Creates a new DeliveryBoy object and associates it with the city in deliveryBoyMap.

**Method: displayCities()**

public void displayCities() {

    System.out.println("Available Destinations:");

    for (int cityNumber : cityNumberMap.keySet()) {

        System.out.println(cityNumber + ". " + cityNumberMap.get(cityNumber));

    }

}

* **Purpose**: Displays the list of all cities along with their assigned city numbers.
* **Output**:

Available Destinations:

1. Vijayawada

2. Bhimavaram

3.Hyderabad

4.Visakhapatnam

**Method: displayDeliveryBoyDetails()**

public void displayDeliveryBoyDetails() {

    System.out.println("Delivery Boy Details:");

    for (DeliveryBoy boy : deliveryBoyMap.values()) {

        System.out.println("Name: " + boy.getName() + ", Contact: " + boy.getContactNumber());

    }

}

* **Purpose**: Displays the details (name and contact) of all delivery boys.

**Method: displayAllDetails()**

public void displayAllDetails() {

    System.out.println("City and Delivery Boy Details:");

    for (int cityNumber : cityNumberMap.keySet()) {

        String city = cityNumberMap.get(cityNumber);

        DeliveryBoy boy = deliveryBoyMap.get(city);

        System.out.println(city + " - " + boy.getName() + " - " + boy.getContactNumber());

    }

}

* **Purpose**: Displays both city names and their corresponding delivery boys' details in one consolidated output.

**Method: getCityByNumber()**

public String getCityByNumber(int cityNumber) {

    return cityNumberMap.get(cityNumber);

}

* **Purpose**: Retrieves the city name associated with the given city number.

**Method: getDeliveryBoy()**

public DeliveryBoy getDeliveryBoy(String city) {

    return deliveryBoyMap.get(city);

}

* **Purpose**: Retrieves the DeliveryBoy object associated with a specific city.

**Orders.java**

The Orders class is a well-structured representation of an order in a logistics or delivery system. Here's an overview of its functionality, along with explanations:

**Fields**

private String orderId;

private String accountId;

private int weight;

private int distance;

private double price;

* **orderId**: A unique identifier for the order. Currently hardcoded as "ORD1" but could be dynamically generated in a real-world application.
* **accountId**: Represents the account associated with the order (e.g., customer or user ID).
* **weight**: Weight of the parcel in kilograms.
* **distance**: Distance for the delivery in kilometers.
* **price**: Total price calculated based on weight and distance.

**Constructor**

public Orders(String accountId, int weight, int distance) {

    this.orderId = "ORD1";

    this.accountId = accountId;

    this.weight = weight;

    this.distance = distance;

    this.price = calculatePrice(weight, distance);

}

* Initializes the order with accountId, weight, and distance.
* **Key Logic**:
  + Sets the orderId (currently fixed).
  + Calculates the price of the order using the calculatePrice() method.

**Price Calculation**

public double calculatePrice(int weight, int distance) {

    double price = distance \* 8; // Base price depends on distance

    if (weight <= 5) price += 50; // Additional charge for weight ≤ 5 kg

    else if (weight <= 10) price += 100; // Additional charge for weight > 5 and ≤ 10 kg

    else price += 150; // Additional charge for weight > 10 kg

    return price;

}

* **Base Rate**: distance \* 8
  + Charges ₹8 per kilometer.
* **Weight-based Additional Charges**:
  + ₹50 for weight ≤ 5 kg.
  + ₹100 for weight between 5 and 10 kg.
  + ₹150 for weight > 10 kg.

**Getters and Setters**

* **Getters** provide read access to private fields:

public String getOrderId();

public int getWeight();

public int getDistance();

public double getPrice();

public String getAccountId();

* **Setters** allow modifying the private fields:

public void setOrderId(String orderId);

public void setPrice(double price);

public void setDistance(int distance);

public void setWeight(int weight);

public void setAccountId(String accountId);

**Invoice.java**

The Invoice class is responsible for generating a detailed invoice for an order. It consolidates the order, customer, and delivery information and presents it in a clear format. Here's a detailed breakdown of the code:

**1. Class Declaration: Invoice**

public class Invoice {

    public static void generateInvoice(Orders order, Users user, DeliveryBoy deliveryBoy) {

        // Invoice generation logic

    }

}

* **Purpose**: The Invoice class acts as a utility class to generate and display the invoice. The method generateInvoice is declared static, indicating that it can be called without creating an instance of the Invoice class.

**2. Method: generateInvoice**

public static void generateInvoice(Orders order, Users user, DeliveryBoy deliveryBoy) {

    // Invoice details

}

* **Purpose**: This method takes three parameters:
  + Orders order: Contains details about the order, such as weight, distance, and price.
  + Users user: Contains details about the customer, such as name, email, and address.
  + DeliveryBoy deliveryBoy: Contains details about the delivery person, such as name and contact number.

**3. Invoice Content**

The method uses System.out.println to format and display the invoice. Here's a breakdown of the sections:

**a. Order Details**

System.out.println("Order ID: " + order.getOrderId());

System.out.println("Weight: " + order.getWeight() + " kg");

System.out.println("Distance: " + order.getDistance() + " km");

System.out.println("Total Bill: Rs " + order.getPrice());

* **Purpose**: Displays key information about the order:
  + orderId: A unique identifier for the order.
  + weight: The weight of the package in kilograms.
  + distance: The delivery distance in kilometers.
  + price: The total cost calculated based on weight and distance.

**b. Customer Details**

System.out.println("Customer: " + user.getName());

System.out.println("Email: " + user.getEmail());

System.out.println("Address: " + user.getAddress());

System.out.println("Phone Number: " + user.getMobileNo());

* **Purpose**: Displays customer information:
  + name: The customer's full name.
  + email: The customer's email address.
  + address: The delivery address.
  + mobileNo: The customer's contact number.

**c. Delivery Boy Details**

System.out.println("Name: " + deliveryBoy.getName());

System.out.println("Contact Number: " + deliveryBoy.getContactNumber());

* **Purpose**: Displays information about the assigned delivery boy:
  + name: The delivery boy's full name.
  + contactNumber: The delivery boy's phone number.

**Main.java**

**Main Functionality**

The program uses a console-based menu-driven interface, allowing interaction for **users** and **admins**. It utilizes supporting classes (UsersDetails, DeliveryBoyDetails, AdminDetails, Orders, and Invoice) to implement its features.

**Code Workflow**

1. **Importing Required Packages**:
   * Scanner is used for input handling.
2. **Main Class**:
   * Main is the entry point of the program.
3. **Object Instantiation**:
   * UsersDetails, DeliveryBoyDetails, and AdminDetails objects are created to handle different functionalities related to users, delivery boys, and admins, respectively.
4. **Menu Display**:
   * The program starts with a menu offering two options: **User** or **Admin**.

System.out.println("Welcome! Select an option:");

System.out.println("1. User\n2. Admin");

**User Flow:**

* If the user selects **User**:
  1. **User Registration**:
     + Collects user details such as ID, name, phone, email, password, and address, and registers the user using usersDetails.registerUser(...).
  2. **Order Details**:
     + The user enters the weight and distance of the package.
     + The program displays available cities for delivery.
     + The user selects a city, and a delivery boy is assigned.
  3. **Invoice Generation**:
     + Creates an order object and fetches user and delivery boy details to generate an invoice using Invoice.generateInvoice().

**Admin Flow:**

* If the user selects **Admin**:
  1. **Admin Login**:
     + Prompts for Admin ID and Password.
     + If login is successful (adminDetails.loginAdmin()), the admin menu is displayed.
  2. **Admin Menu**:
     + Options include:
       1. Displaying available destinations.
       2. Viewing delivery boy details.
       3. Adding new destinations and assigning delivery boys.
       4. Logging out.
     + Admin actions modify the delivery system by updating city and delivery boy details.

**Key Supporting Functionalities:**

* **UsersDetails**: Handles user registration and fetching user details.
* **DeliveryBoyDetails**: Manages city and delivery boy information, including adding new destinations and displaying delivery boy details.
* **AdminDetails**: Validates admin credentials and provides access to admin functionalities.
* **Orders**: Represents order details such as weight and distance.
* **Invoice**: Generates invoices based on user, order, and delivery boy data.

**Error Handling:**

* Invalid city choice is handled with a null check.
* Admin authentication ensures only authorized access.
* Invalid menu choices prompt the user to try again.

**Code Snippet Example:**

Below is the part handling user registration and order placement:

if (choice == 1) {

    System.out.println("Enter User Details:");

    System.out.println("Enter User ID:");

    String userId = scanner.nextLine();

    // Collect other details like name, phone, email, etc.

    usersDetails.registerUser(userId, name, mobileNo, email, password, address);

    System.out.println("User Registered");

    System.out.println("Enter Weight:");

    int weight = scanner.nextInt();

    System.out.println("Enter Distance:");

    int distance = scanner.nextInt();

    scanner.nextLine();

    deliveryBoyDetails.displayCities();

    System.out.print("Enter your choice: ");

    int cityChoice = scanner.nextInt();

    scanner.nextLine();

    String city = deliveryBoyDetails.getCityByNumber(cityChoice);

    if (city != null) {

        DeliveryBoy deliveryBoy = deliveryBoyDetails.getDeliveryBoy(city);

        Orders order = new Orders(userId, weight, distance);

        Users user = usersDetails.getUser(userId);

        Invoice.generateInvoice(order, user, deliveryBoy);

    } else {

        System.out.println("Invalid city");

    }

}

**Purpose:**

* The program manages a basic delivery system workflow, including **user registration, order placement**, and **admin management** of destinations and delivery boys.
* else if (adminChoice == 2) {
* deliveryBoyDetails.displayDeliveryBoyDetails();
* } else if (adminChoice == 3) {
* System.out.println("Enter new City Name:");
* String newCity = scanner.nextLine();
* System.out.println("Enter Delivery Boy Name:");
* String deliveryBoyName = scanner.nextLine();
* System.out.println("Enter Contact Number:");
* String contactNumber = scanner.nextLine();
* deliveryBoyDetails.addDestination(newCity, deliveryBoyName, contactNumber);
* System.out.println("Updated Destinations and Delivery Boys:");
* deliveryBoyDetails.displayAllDetails();
* } else if (adminChoice == 4) {
* System.out.println("Logged out successfully");
* break;
* } else {
* System.out.println("Invalid Please try again");
* }
* }
* } else {
* System.out.println("Wrong admin details");
* }
* } else {
* System.out.println("Invalid option");
* }
* scanner.close();
* }
* }

This code snippet represents the logic for an admin section of a system where an administrator can log in, view available destinations, display delivery boy details, add new destinations, and log out. Here's a breakdown of the key components:

1. **Admin Login (Admin Authentication):**
   * The user is prompted to enter an Admin ID and password.
   * The adminDetails.loginAdmin(adminId, adminPassword) method is called to verify the credentials.
   * If the login is successful (admin != null), the system proceeds to the Admin menu; otherwise, an error message ("Wrong admin details") is displayed.
2. **Admin Menu:**
   * Upon successful login, the admin is presented with a menu with four options:
     1. **Display Available Destinations**: Shows a list of available cities (destinations).
     2. **Display Delivery Boy Details**: Displays information about the delivery boys.
     3. **Add Destination and Delivery Boy**: Allows the admin to add a new city (destination) and a delivery boy along with their contact information.
     4. **Logout**: Logs the admin out of the system.
3. **Admin Menu Logic:**
   * If the admin selects option 1, the method deliveryBoyDetails.displayCities() is called to display the available cities.
   * If option 2 is selected, the method deliveryBoyDetails.displayDeliveryBoyDetails() is invoked to show the details of the delivery boys.
   * If option 3 is selected, the admin is prompted to enter a new city name, delivery boy name, and contact number. These details are passed to the deliveryBoyDetails.addDestination(newCity, deliveryBoyName, contactNumber) method to add the new destination and delivery boy. The updated list is then displayed using deliveryBoyDetails.displayAllDetails().
   * If option 4 is selected, the admin is logged out and the system returns to the main menu.
   * An error message is shown if an invalid menu option is entered.
4. **Ending the Program:**
   * The scanner.close() method is called to close the scanner resource after use.

**Important Notes:**

* The methods adminDetails.loginAdmin(), deliveryBoyDetails.displayCities(), deliveryBoyDetails.displayDeliveryBoyDetails(), deliveryBoyDetails.addDestination(), and deliveryBoyDetails.displayAllDetails() are assumed to be implemented elsewhere in the code, handling the logic for login and data operations.
* The scanner object is used to read input from the user.
* The loop (while (true)) ensures that the admin menu is continuously displayed until the admin logs out (selecting option 4).
* The program gracefully handles invalid inputs by printing an error message.

This code demonstrates basic console-based admin functionality for managing destinations and delivery boys in a logistics or delivery system.

**Conclusion:**

The low-level design of the logistics system outlines a robust and efficient framework for managing administrative tasks related to destinations and delivery boys. The system facilitates secure admin login, with functionalities for viewing available destinations, managing delivery boy details, and adding new destinations. It ensures smooth user interaction through a clear and simple menu structure, allowing the admin to manage the system with ease. By implementing methods for adding and displaying destinations and delivery boy details, the system supports scalability and future enhancements. Overall, this design provides a strong foundation for a logistics platform that can be further extended to accommodate additional features as needed.

