CS F213

**Smart Parking Management System**

Object Oriented Programming Project

horizontal line

# RUBRIC

| Category | Count | Specific Examples |
| --- | --- | --- |
| Overloaded Methods | 3 | PricingManager.calculateFee |
| Overloaded Constructors | 5 | Vehicle, Car, Bike, Truck, ElectricVehicle |
| Vararg Overloading | 2 | PricingManager.calculateFee |
| Nested Classes | 1 | RenewalPeriod(its a static record) |
| Abstract Classes | 1 | Vehicle |
| Interface | 2 | PaymentMethod |
| Hierarchial Inheritance | 1 | Vehicle → Car, Bike, Truck, ElectricVehicle |
| Multiple Inheritance | 1 | CreditCardPayment implements PaymentMethod and Loggable |
| Wrappers | 2 | Double in PricingManager |
| Package | 5 | com.parkinglot.system.model |
| Exception Handling | 3 | NoAvailableSpotException |
| I/O: File Handling, Scanner Class | 2 | ParkingLot.saveDataToFile |

# 

# 

# 

# 

# 

# 

# Introduction

The Parking Lot Management System is a Java-based command-line application designed to manage parking operations in a multi-vehicle parking facility. The system supports parking, reservations, subscriptions, payment processing, and security logging, catering to various vehicle types (cars, bikes, trucks, electric vehicles) and spot types (car, bike, truck, electric, handicapped). It provides a user-friendly interface for parking lot administrators to perform tasks such as checking in vehicles, managing reservations, processing payments, and viewing logs. The application accomplishes the following:

* Assign parking spots based on vehicle and spot compatibility.
* Allow users to reserve spots for specific times.
* Offer monthly, quarterly, semi-annual, and annual subscriptions for frequent users.
* Calculate fees based on vehicle type, spot type, duration, and dynamic factors (e.g., peak hours, weekends).
* Support multiple payment methods (credit card, UPI).
* Track vehicle entry/exit and transactions for audit purposes.
* Save and load parking lot data to/from a file for continuity.
* Uses features such as Exception handling, Serialization, Object oriented programming, abstraction, Inheritance and File handling.

## Architecture

The application consists of a user defined package called parkinglot is subdivided into:

* Com.parkinglot.system.model:
  + Contains classes namely Vehicle, Car, Bike, Truck, ElectricVehicle, ParkingSpot, Reservation, EntryExitLog and Subscription.
* Com.parkinglot.system.management
  + These include classes such as ParkingLot, PricingManager, SubscriptionManager
* Com.parkinglot.system.payment
  + This contains Payment, PaymentMethod, CreditCardPayment, UPIPayment and Loggable
* Com.parkinglot.system.exceptions
  + This contains custom exceptions such as NoAvailableSpotException, InvalidReservationException, VehicleNotFoundException.
* com.parkinglot.system.ui
  + Contains ParkingLotSystem
  + Provides a user interface through a menu system

## Class Details

**Com.parkinglot.system.model**

* **Vehicle(Abstract):** Base class for all vehicle types, defining common properties and an abstract method for parking rates.

| package com.parkinglot.system.model;  import java.io.Serializable;  /\*\*  \* Abstract base class for vehicles in the parking lot system.  \* Defines common properties and behaviors for all vehicle types.  \* Implements Serializable to support persistence.  \*/ public abstract class Vehicle implements Serializable {  private static final long serialVersionUID = 1L;   public static final String TYPE\_CAR = "CAR";  public static final String TYPE\_BIKE = "BIKE";  public static final String TYPE\_TRUCK = "TRUCK";  public static final String TYPE\_BUS = "BUS";   private String licensePlate;  private String type;  private boolean handicappedPermit;   /\*\*  \* Constructs a Vehicle with a specified license plate and type.  \*  \* @param licensePlate The license plate of the vehicle  \* @param type The type of vehicle (e.g., CAR, BIKE, TRUCK, BUS)  \* @throws IllegalArgumentException If the vehicle type is invalid  \*/  public Vehicle(String licensePlate, String type) {  validateVehicleType(type);  this.licensePlate = licensePlate.toUpperCase();  this.type = type;  this.handicappedPermit = false;  }   /\*\*  \* Constructs a Vehicle with a specified license plate, type, and handicapped permit status.  \*  \* @param licensePlate The license plate of the vehicle  \* @param type The type of vehicle (e.g., CAR, BIKE, TRUCK, BUS)  \* @param handicappedPermit Whether the vehicle has a handicapped permit  \* @throws IllegalArgumentException If the vehicle type is invalid  \*/  public Vehicle(String licensePlate, String type, boolean handicappedPermit) {  validateVehicleType(type);  this.licensePlate = licensePlate.toUpperCase();  this.type = type;  this.handicappedPermit = handicappedPermit;  }   /\*\*  \* Validates the vehicle type to ensure it is one of the allowed types.  \*  \* @param type The vehicle type to validate  \* @throws IllegalArgumentException If the vehicle type is invalid  \*/  public static void validateVehicleType(String type) {  if (type == null || !(type.equals(TYPE\_CAR) || type.equals(TYPE\_BIKE) ||  type.equals(TYPE\_TRUCK) || type.equals(TYPE\_BUS))) {  throw new IllegalArgumentException("Invalid vehicle type: " + type);  }  }   /\*\*  \* Retrieves the base parking rate for the vehicle.  \*  \* @return The base parking rate  \*/  public abstract double getBaseParkingRate();   /\*\*  \* Retrieves the license plate of the vehicle.  \*  \* @return The license plate  \*/  public String getLicensePlate() {  return licensePlate;  }   /\*\*  \* Retrieves the type of the vehicle.  \*  \* @return The vehicle type (e.g., CAR, BIKE, TRUCK, BUS)  \*/  public String getType() {  return type;  }   /\*\*  \* Checks if the vehicle has a handicapped permit.  \*  \* @return True if the vehicle has a handicapped permit, false otherwise  \*/  public boolean hasHandicappedPermit() {  return handicappedPermit;  }   /\*\*  \* Sets the handicapped permit status of the vehicle.  \*  \* @param handicappedPermit Whether the vehicle has a handicapped permit  \*/  public void setHandicappedPermit(boolean handicappedPermit) {  this.handicappedPermit = handicappedPermit;  } } |
| --- |

* **Car:** Represents cars, with optional luxury status affecting parking rates.

| package com.parkinglot.system.model;  import java.io.Serializable;  /\*\*  \* Represents a car in the parking lot system.  \* Extends Vehicle to provide specific properties and behaviors for cars.  \* Implements Serializable to support persistence.  \*/  public class Car extends Vehicle implements Serializable {  private static final long serialVersionUID = 1L;  private boolean isLuxury;  /\*\*  \* Constructs a car with the specified license plate.  \*  \* @param licensePlate The car's license plate  \*/  public Car(String licensePlate) {  super(licensePlate, TYPE\_CAR);  this.isLuxury = false;  }  /\*\*  \* Constructs a car with a luxury option.  \*  \* @param licensePlate The car's license plate  \* @param isLuxury Whether the car is a luxury model  \*/  public Car(String licensePlate, boolean isLuxury) {  super(licensePlate, TYPE\_CAR);  this.isLuxury = isLuxury;  }  /\*\*  \* Returns whether the car is a luxury model.  \*  \* @return True if the car is luxury, false otherwise  \*/  public boolean isLuxury() {  return isLuxury;  }  /\*\*  \* Returns the base hourly parking rate for the car.  \*  \* @return The base parking rate (40.0 per hour, 48.0 for luxury)  \*/  @Override  public double getBaseParkingRate() {  return isLuxury ? 48.0 : 40.0;  }  } |
| --- |

* **Bike:** Represents bikes with a fixed parking rate.

| package com.parkinglot.system.model;  import java.io.Serializable;  /\*\*  \* Represents a bike in the parking lot system.  \* Extends Vehicle to provide specific properties and behaviors for bikes.  \* Implements Serializable to support persistence.  \*/ public class Bike extends Vehicle implements Serializable {  private static final long serialVersionUID = 1L;   /\*\*  \* Constructs a bike with the specified license plate.  \*  \* @param licensePlate The bike's license plate  \*/  public Bike(String licensePlate) {  super(licensePlate, TYPE\_BIKE);  }   /\*\*  \* Constructs a bike with a handicapped permit option.  \*  \* @param licensePlate The bike's license plate  \* @param handicappedPermit Whether the bike has a handicapped permit  \*/  public Bike(String licensePlate, boolean handicappedPermit) {  super(licensePlate, TYPE\_BIKE, handicappedPermit);  }   /\*\*  \* Returns the base hourly parking rate for the bike.  \*  \* @return The base parking rate (20.0 per hour)  \*/  @Override  public double getBaseParkingRate() {  return 20.0;  } } |
| --- |

* **Truck:** Represents trucks with weight-based properties.

| package com.parkinglot.system.model;  import java.io.Serializable;  /\*\*  \* Represents a truck in the parking lot system.  \* Extends Vehicle to provide specific properties and behaviors for trucks.  \* Implements Serializable to support persistence.  \*/ public class Truck extends Vehicle implements Serializable {  private static final long serialVersionUID = 1L;  private double weight;   /\*\*  \* Constructs a truck with the specified license plate and weight.  \*  \* @param licensePlate The truck's license plate  \* @param weight The truck's weight in tons  \*/  public Truck(String licensePlate, double weight) {  super(licensePlate, TYPE\_TRUCK);  this.weight = weight;  }   /\*\*  \* Constructs a truck with a handicapped permit option.  \*  \* @param licensePlate The truck's license plate  \* @param handicappedPermit Whether the truck has a handicapped permit  \*/  public Truck(String licensePlate, boolean handicappedPermit) {  super(licensePlate, TYPE\_TRUCK, handicappedPermit);  this.weight = 1.0; // Default weight  }   /\*\*  \* Returns the truck's weight.  \*  \* @return The weight in tons  \*/  public double getWeight() {  return weight;  }   /\*\*  \* Returns the base hourly parking rate for the truck.  \*  \* @return The base parking rate (80.0 per hour)  \*/  @Override  public double getBaseParkingRate() {  return 80.0;  } } |
| --- |

* **ElectricVehicle:** Represents electric vehicles with battery capacity.

| package com.parkinglot.system.model;  import java.io.Serializable;  /\*\*  \* Represents an electric vehicle in the parking lot system.  \* Extends Vehicle to provide specific properties and behaviors for electric vehicles.  \* Implements Serializable to support persistence.  \*/ public class ElectricVehicle extends Vehicle implements Serializable {  private static final long serialVersionUID = 1L;  private double batteryCapacity;   /\*\*  \* Constructs an electric vehicle with the specified license plate and battery capacity.  \*  \* @param licensePlate The electric vehicle's license plate  \* @param batteryCapacity The battery capacity in kWh  \*/  public ElectricVehicle(String licensePlate, double batteryCapacity) {  super(licensePlate, TYPE\_CAR);  this.batteryCapacity = batteryCapacity;  }   /\*\*  \* Constructs an electric vehicle with a handicapped permit option.  \*  \* @param licensePlate The electric vehicle's license plate  \* @param handicappedPermit Whether the electric vehicle has a handicapped permit  \*/  public ElectricVehicle(String licensePlate, boolean handicappedPermit) {  super(licensePlate, TYPE\_CAR, handicappedPermit);  this.batteryCapacity = 50.0; // Default capacity  }   /\*\*  \* Returns the battery capacity.  \*  \* @return The battery capacity in kWh  \*/  public double getBatteryCapacity() {  return batteryCapacity;  }   /\*\*  \* Returns the base hourly parking rate for the electric vehicle.  \*  \* @return The base parking rate (48.0 per hour)  \*/  @Override  public double getBaseParkingRate() {  return 48.0;  } } |
| --- |

* **ParkingSpot:** Represents a parking spot with type and availability status.

| package com.parkinglot.system.model;  import java.io.Serializable; import java.util.Date;  /\*\*  \* Represents a parking spot in a parking lot, capable of holding a vehicle or a reservation.  \* Each spot has a unique ID, a specific type (e.g., car, bike, electric), and tracks its availability.  \* Implements Serializable to allow the spot's state to be saved or transmitted.  \*/ public class ParkingSpot implements Serializable {   private static final long serialVersionUID = 1L;   // Constants for parking spot types  public static final String TYPE\_CAR = "CAR";  public static final String TYPE\_BIKE = "BIKE";  public static final String TYPE\_TRUCK = "TRUCK";  public static final String TYPE\_ELECTRIC = "ELECTRIC";  public static final String TYPE\_HANDICAPPED = "HANDICAPPED";   private String spotId;  private String type;  private boolean available;  private Vehicle occupyingVehicle;  private Reservation currentReservation;   /\*\*  \* Constructs a new parking spot with a given ID and type.  \* The spot starts as available with no vehicle or reservation.  \*  \* @param spotId The unique identifier for the spot  \* @param type The type of parking spot (e.g., CAR, BIKE)  \* @throws IllegalArgumentException If the type is invalid  \*/  public ParkingSpot(String spotId, String type) {  validateType(type);  this.spotId = spotId;  this.type = type;  this.available = true;  this.occupyingVehicle = null;  this.currentReservation = null;  }   /\*\*  \* Validates the parking spot type.  \*  \* @param type The type to validate  \* @throws IllegalArgumentException If the type is invalid  \*/  private void validateType(String type) {  if (type == null || !(type.equals(TYPE\_CAR) || type.equals(TYPE\_BIKE) || type.equals(TYPE\_TRUCK) ||  type.equals(TYPE\_ELECTRIC) || type.equals(TYPE\_HANDICAPPED))) {  throw new IllegalArgumentException("Invalid parking spot type: " + type);  }  }   /\*\*  \* Checks if the parking spot is available for use.  \* A spot is available if it's not occupied and has no active reservation.  \*  \* @return True if the spot is available, false otherwise  \*/  public boolean isAvailable() {  return available && currentReservation == null;  }   /\*\*  \* Determines if a vehicle can park in this spot based on the spot's type and vehicle properties.  \*  \* @param vehicleType The type of vehicle  \* @return True if the vehicle can park in this spot, false otherwise  \*/  public boolean canFit(String vehicleType) {  Vehicle.validateVehicleType(vehicleType);  return switch (type) {  case TYPE\_CAR -> vehicleType.equals(Vehicle.TYPE\_CAR) || vehicleType.equals(Vehicle.TYPE\_BIKE);  case TYPE\_BIKE -> vehicleType.equals(Vehicle.TYPE\_BIKE);  case TYPE\_TRUCK -> true; // Truck spots can accommodate any vehicle  case TYPE\_ELECTRIC -> vehicleType.equals(Vehicle.TYPE\_CAR);  case TYPE\_HANDICAPPED -> vehicleType.equals(Vehicle.TYPE\_CAR) || vehicleType.equals(Vehicle.TYPE\_BIKE);  default -> false;  };  }   /\*\*  \* Marks the spot as occupied by a vehicle.  \* Sets the spot as unavailable and assigns the vehicle to it.  \*  \* @param vehicle The vehicle occupying the spot  \*/  public void parkVehicle(Vehicle vehicle) {  this.available = false;  this.occupyingVehicle = vehicle;  }   /\*\*  \* Releases the spot, making it available again.  \* Clears the occupying vehicle.  \*/  public void removeVehicle() {  this.available = true;  this.occupyingVehicle = null;  if (currentReservation != null && !currentReservation.getStatus().equals(Reservation.STATUS\_CHECKED\_IN)) {  currentReservation = null;  }  }   /\*\*  \* Assigns a reservation to the spot.  \*  \* @param reservation The reservation to assign  \*/  public void reserve(Reservation reservation) {  this.currentReservation = reservation;  this.available = false;  }   /\*\*  \* Cancels the current reservation, freeing the spot for other uses.  \*/  public void cancelReservation() {  this.currentReservation = null;  if (this.occupyingVehicle == null) {  this.available = true;  }  }   /\*\*  \* Checks if the spot is available for a reservation during the specified time period.  \*  \* @param startTime The start time of the reservation  \* @param endTime The end time of the reservation  \* @return True if the spot is available, false otherwise  \*/  public boolean isAvailableForReservation(Date startTime, Date endTime) {  if (currentReservation == null) {  return isAvailable();  }  return currentReservation.getEndTime().before(startTime) || currentReservation.getStartTime().after(endTime);  }   /\*\*  \* Returns the unique ID of the parking spot.  \*  \* @return The spot ID  \*/  public String getSpotId() {  return spotId;  }   /\*\*  \* Returns the type of the parking spot.  \*  \* @return The spot type (e.g., CAR, BIKE)  \*/  public String getType() {  return type;  }   /\*\*  \* Returns the vehicle currently occupying the spot, if any.  \*  \* @return The occupying vehicle, or null if the spot is empty  \*/  public Vehicle getOccupyingVehicle() {  return occupyingVehicle;  }   /\*\*  \* Returns the current reservation for the spot, if any.  \*  \* @return The current reservation, or null if none exists  \*/  public Reservation getCurrentReservation() {  return currentReservation;  } } |
| --- |

* **Reservation:** Manages parking reservations with time and status.

| package com.parkinglot.system.model;  import java.io.Serializable; import java.util.Date;  /\*\*  \* Represents a reservation for a parking spot in the parking lot system.  \* Tracks the parking spot, vehicle, reservation period, and status.  \* Implements Serializable to support persistence.  \*/ public class Reservation implements Serializable {  private static final long serialVersionUID = 1L;   // Constants for reservation status  public static final String STATUS\_PENDING = "PENDING";  public static final String STATUS\_CHECKED\_IN = "CHECKED\_IN";  public static final String STATUS\_COMPLETED = "COMPLETED";  public static final String STATUS\_CANCELLED = "CANCELLED";   private ParkingSpot spot;  private Vehicle vehicle;  private Date startTime;  private Date endTime;  private String status;   /\*\*  \* Constructs a reservation for a parking spot.  \*  \* @param spot The parking spot reserved  \* @param vehicle The vehicle for the reservation  \* @param startTime The start time of the reservation  \* @param endTime The end time of the reservation  \*/  public Reservation(ParkingSpot spot, Vehicle vehicle, Date startTime, Date endTime) {  this.spot = spot;  this.vehicle = vehicle;  this.startTime = startTime;  this.endTime = endTime;  this.status = STATUS\_PENDING;  }   /\*\*  \* Checks if the reservation is active.  \*  \* @return True if the reservation is pending or checked in, false otherwise  \*/  public boolean isActive() {  Date now = new Date();  return (status.equals(STATUS\_PENDING) || status.equals(STATUS\_CHECKED\_IN)) &&  now.after(startTime) && now.before(endTime);  }   /\*\*  \* Sets the status of the reservation.  \*  \* @param status The new status (e.g., CHECKED\_IN, COMPLETED)  \* @throws IllegalArgumentException If the status is invalid  \*/  public void setStatus(String status) {  validateStatus(status);  this.status = status;  }   /\*\*  \* Validates the reservation status.  \*  \* @param status The status to validate  \* @throws IllegalArgumentException If the status is invalid  \*/  private void validateStatus(String status) {  if (status == null || !(status.equals(STATUS\_PENDING) || status.equals(STATUS\_CHECKED\_IN) ||  status.equals(STATUS\_COMPLETED) || status.equals(STATUS\_CANCELLED))) {  throw new IllegalArgumentException("Invalid reservation status: " + status);  }  }   /\*\*  \* Returns the parking spot for the reservation.  \*  \* @return The parking spot  \*/  public ParkingSpot getSpot() {  return spot;  }   /\*\*  \* Returns the vehicle for the reservation.  \*  \* @return The vehicle  \*/  public Vehicle getVehicle() {  return vehicle;  }   /\*\*  \* Returns the start time of the reservation.  \*  \* @return The start time  \*/  public Date getStartTime() {  return startTime;  }   /\*\*  \* Returns the end time of the reservation.  \*  \* @return The end time  \*/  public Date getEndTime() {  return endTime;  }   /\*\*  \* Returns the status of the reservation.  \*  \* @return The status  \*/  public String getStatus() {  return status;  } } |
| --- |

* **EntryExitLog:** Tracks vehicle entry and exit times for logging.

| package com.parkinglot.system.model;  import java.io.Serializable; import java.util.Date;  /\*\*  \* Represents a log entry for a vehicle's entry and exit in the parking lot.  \* Tracks the vehicle, parking spot, entry time, and exit time.  \* Implements Serializable to support persistence.  \*/ public class EntryExitLog implements Serializable {  private static final long serialVersionUID = 1L;   private Vehicle vehicle;  private ParkingSpot spot;  private Date entryTime;  private Date exitTime;   /\*\*  \* Constructs an entry-exit log for a vehicle.  \*  \* @param vehicle The vehicle entering the parking lot  \* @param spot The assigned parking spot  \* @param entryTime The time of entry  \*/  public EntryExitLog(Vehicle vehicle, ParkingSpot spot, Date entryTime) {  this.vehicle = vehicle;  this.spot = spot;  this.entryTime = entryTime;  this.exitTime = null;  }   /\*\*  \* Sets the exit time for the vehicle.  \*  \* @param exitTime The time of exit  \*/  public void setExitTime(Date exitTime) {  this.exitTime = exitTime;  }   /\*\*  \* Calculates the parking duration in hours.  \*  \* @return The duration in hours, or 0.0 if exit time is not set  \*/  public Double getParkingDuration() {  if (exitTime == null || entryTime == null) {  return 0.0;  }  long durationMillis = exitTime.getTime() - entryTime.getTime();  return durationMillis / (1000.0 \* 60 \* 60); // Convert milliseconds to hours  }   /\*\*  \* Returns the vehicle associated with the log.  \*  \* @return The vehicle  \*/  public Vehicle getVehicle() {  return vehicle;  }   /\*\*  \* Returns the parking spot associated with the log.  \*  \* @return The parking spot  \*/  public ParkingSpot getSpot() {  return spot;  }   /\*\*  \* Returns the entry time.  \*  \* @return The entry time  \*/  public Date getEntryTime() {  return entryTime;  }   /\*\*  \* Returns the exit time.  \*  \* @return The exit time, or null if not set  \*/  public Date getExitTime() {  return exitTime;  } } |
| --- |

* **Subscription:** Manages vehicle subscriptions (monthly, quarterly, etc.).

| package com.parkinglot.system.model;  import java.io.Serializable; import java.util.Calendar; import java.util.Date; import java.util.UUID;  /\*\*  \* Represents a parking subscription for a vehicle in the parking lot system.  \* Tracks the vehicle, subscription period, type, and spot type.  \* Implements Serializable to support persistence.  \*/ public class Subscription implements Serializable {  private static final long serialVersionUID = 1L;   public static final String TYPE\_MONTHLY = "MONTHLY";  public static final String TYPE\_QUARTERLY = "QUARTERLY";  public static final String TYPE\_SEMI\_ANNUAL = "SEMI\_ANNUAL";  public static final String TYPE\_ANNUAL = "ANNUAL";   private String subscriptionId;  private Vehicle vehicle;  private Date startDate;  private Date endDate;  private String type;  private String spotType;  private boolean active;   /\*\*  \* Constructs a Subscription for a vehicle with specified details.  \*  \* @param vehicle The vehicle associated with the subscription  \* @param startDate The start date of the subscription  \* @param endDate The end date of the subscription  \* @param type The type of subscription (e.g., MONTHLY, QUARTERLY)  \* @param spotType The type of parking spot associated with the subscription  \* @throws IllegalArgumentException If the subscription type is invalid  \*/  public Subscription(Vehicle vehicle, Date startDate, Date endDate, String type, String spotType) {  validateType(type);  this.subscriptionId = UUID.randomUUID().toString();  this.vehicle = vehicle;  this.startDate = startDate;  this.endDate = endDate;  this.type = type;  this.spotType = spotType;  this.active = true;  }   /\*\*  \* Validates the subscription type to ensure it is one of the allowed types.  \*  \* @param type The subscription type to validate  \* @throws IllegalArgumentException If the subscription type is invalid  \*/  private void validateType(String type) {  if (type == null || !(type.equals(TYPE\_MONTHLY) || type.equals(TYPE\_QUARTERLY) ||  type.equals(TYPE\_SEMI\_ANNUAL) || type.equals(TYPE\_ANNUAL))) {  throw new IllegalArgumentException("Invalid subscription type: " + type);  }  }   /\*\*  \* Checks if the subscription is currently active based on its date range and status.  \*  \* @return True if the subscription is active, false otherwise  \*/  public boolean isActive() {  Date now = new Date();  return active && now.after(startDate) && now.before(endDate);  }   /\*\*  \* Cancels the subscription by setting its active status to false.  \*/  public void cancel() {  this.active = false;  }   /\*\*  \* Renews the subscription by extending its end date by the specified number of months.  \*  \* @param months The number of months to extend the subscription  \*/  public void renew(int months) {  if (active) {  Calendar calendar = Calendar.getInstance();  calendar.setTime(endDate);  calendar.add(Calendar.MONTH, months);  this.endDate = calendar.getTime();  }  }   /\*\*  \* Retrieves the unique identifier of the subscription.  \*  \* @return The subscription ID  \*/  public String getSubscriptionId() {  return subscriptionId;  }   /\*\*  \* Retrieves the vehicle associated with the subscription.  \*  \* @return The vehicle object  \*/  public Vehicle getVehicle() {  return vehicle;  }   /\*\*  \* Retrieves the start date of the subscription.  \*  \* @return The start date  \*/  public Date getStartDate() {  return startDate;  }   /\*\*  \* Retrieves the end date of the subscription.  \*  \* @return The end date  \*/  public Date getEndDate() {  return endDate;  }   /\*\*  \* Retrieves the type of the subscription.  \*  \* @return The subscription type (e.g., MONTHLY, QUARTERLY)  \*/  public String getType() {  return type;  }   /\*\*  \* Retrieves the type of parking spot associated with the subscription.  \*  \* @return The parking spot type  \*/  public String getSpotType() {  return spotType;  } } |
| --- |

**Com.parkinglot.system.management**

* **ParkingLot:** Core class managing parking operations, reservations, and transactions.

| package com.parkinglot.system.management;  import com.parkinglot.system.model.\*; import com.parkinglot.system.payment.Payment; import com.parkinglot.system.payment.Loggable; import com.parkinglot.system.exceptions.\*; import java.io.\*; import java.util.\*; import java.text.SimpleDateFormat;  /\*\*  \* Represents a parking lot with multiple parking spots, managing vehicle check-in, check-out, and reservations.  \* Implements Serializable to support persistence.  \*/ public class ParkingLot implements Serializable, Loggable {  private static final long serialVersionUID = 1L;   private String name;  private List<ParkingSpot> spots;  private Map<String, EntryExitLog> activeVehicles;  private List<Reservation> reservations;  private PricingManager pricingManager;  private SubscriptionManager subscriptionManager;  private transient SimpleDateFormat dateFormat;  private Map<String, List<EntryExitLog>> securityLogs;  private List<Payment> transactions;   /\*\*  \* Constructs a ParkingLot with a specified name and total number of spots.  \*  \* @param name The name of the parking lot  \* @param totalSpots The total number of parking spots  \*/  public ParkingLot(String name, int totalSpots) {  this.name = name;  this.spots = new ArrayList<>();  this.activeVehicles = new HashMap<>();  this.reservations = new ArrayList<>();  this.pricingManager = new PricingManager();  this.subscriptionManager = new SubscriptionManager(pricingManager, this);  this.dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm");  this.securityLogs = new HashMap<>();  this.transactions = new ArrayList<>();  initializeSpots(totalSpots);  }   /\*\*  \* Initializes parking spots based on the total number of spots, distributing them across different types.  \*  \* @param totalSpots The total number of parking spots to initialize  \*/  private void initializeSpots(int totalSpots) {  int carSpots = totalSpots / 2;  int bikeSpots = totalSpots / 4;  int truckSpots = totalSpots / 8;  int electricSpots = totalSpots / 8;  int handicappedSpots = totalSpots - carSpots - bikeSpots - truckSpots - electricSpots;   for (int i = 0; i < carSpots; i++) {  spots.add(new ParkingSpot("C" + (i + 1), ParkingSpot.TYPE\_CAR));  }  for (int i = 0; i < bikeSpots; i++) {  spots.add(new ParkingSpot("B" + (i + 1), ParkingSpot.TYPE\_BIKE));  }  for (int i = 0; i < truckSpots; i++) {  spots.add(new ParkingSpot("T" + (i + 1), ParkingSpot.TYPE\_TRUCK));  }  for (int i = 0; i < electricSpots; i++) {  spots.add(new ParkingSpot("E" + (i + 1), ParkingSpot.TYPE\_ELECTRIC));  }  for (int i = 0; i < handicappedSpots; i++) {  spots.add(new ParkingSpot("H" + (i + 1), ParkingSpot.TYPE\_HANDICAPPED));  }  }   /\*\*  \* Checks in a vehicle to the parking lot, assigning it to an available spot.  \*  \* @param vehicle The vehicle to check in  \* @return The assigned parking spot  \* @throws NoAvailableSpotException If no suitable spot is available  \*/  public ParkingSpot checkIn(Vehicle vehicle) throws NoAvailableSpotException {  ParkingSpot spot = findAvailableSpot(vehicle.getType());  if (spot == null) {  throw new NoAvailableSpotException("No available spot for vehicle type: " + vehicle.getType());  }  spot.parkVehicle(vehicle);  EntryExitLog log = new EntryExitLog(vehicle, spot, new Date());  activeVehicles.put(vehicle.getLicensePlate(), log);  securityLogs.computeIfAbsent(vehicle.getLicensePlate(), k -> new ArrayList<>()).add(log);  logTransaction("Check-in for vehicle: " + vehicle.getLicensePlate() + " at spot: " + spot.getSpotId());  return spot;  }   /\*\*  \* Checks in a vehicle with a reservation, assigning it to the reserved spot.  \*  \* @param vehicle The vehicle to check in  \* @param reservation The reservation associated with the vehicle  \* @return The assigned parking spot  \* @throws InvalidReservationException If the reservation is invalid or inactive  \*/  public ParkingSpot checkIn(Vehicle vehicle, Reservation reservation) throws InvalidReservationException {  if (!reservation.isActive() || !reservation.getVehicle().getLicensePlate().equals(vehicle.getLicensePlate())) {  throw new InvalidReservationException("Invalid or inactive reservation");  }  ParkingSpot spot = reservation.getSpot();  spot.parkVehicle(vehicle);  reservation.setStatus(Reservation.STATUS\_CHECKED\_IN);  EntryExitLog log = new EntryExitLog(vehicle, spot, new Date());  activeVehicles.put(vehicle.getLicensePlate(), log);  securityLogs.computeIfAbsent(vehicle.getLicensePlate(), k -> new ArrayList<>()).add(log);  logTransaction("Check-in with reservation for vehicle: " + vehicle.getLicensePlate() + " at spot: " + spot.getSpotId());  return spot;  }   /\*\*  \* Prepares the checkout process for a vehicle, calculating the parking fee.  \*  \* @param vehicle The vehicle to check out  \* @return The payment object containing the fee details  \* @throws VehicleNotFoundException If the vehicle is not found in the parking lot  \*/  public Payment prepareCheckOut(Vehicle vehicle) throws VehicleNotFoundException {  EntryExitLog log = activeVehicles.get(vehicle.getLicensePlate());  if (log == null) {  throw new VehicleNotFoundException("Vehicle with license plate " + vehicle.getLicensePlate() + " not found");  }  log.setExitTime(new Date());  Double hours = log.getParkingDuration();  Double fee=0.0;  if (!subscriptionManager.hasActiveSubscription(vehicle.getLicensePlate())) {  fee = pricingManager.calculateFee(vehicle.getType(), hours, log.getSpot().getType());  } else {  System.out.println("No checkout fee for vehicle: " + vehicle.getLicensePlate() + " due to active subscription");  }  Payment payment = new Payment(fee, vehicle, log);  transactions.add(payment);  return payment;  }   /\*\*  \* Finalizes the checkout process for a vehicle, removing it from the parking lot if payment is successful.  \*  \* @param vehicle The vehicle to check out  \* @param paymentSuccess Indicates whether the payment was successful  \*/  public void finalizeCheckOut(Vehicle vehicle, boolean paymentSuccess) {  if (paymentSuccess) {  EntryExitLog log = activeVehicles.get(vehicle.getLicensePlate());  if (log != null) {  log.getSpot().removeVehicle();  activeVehicles.remove(vehicle.getLicensePlate());  logTransaction("Check-out for vehicle: " + vehicle.getLicensePlate() + " with payment success");  }  } else {  logTransaction("Check-out failed for vehicle: " + vehicle.getLicensePlate() + " due to payment failure");  }  }   /\*\*  \* Creates a reservation for a vehicle for a specified time period.  \*  \* @param vehicle The vehicle to reserve a spot for  \* @param startTime The start time of the reservation  \* @param endTime The end time of the reservation  \* @return The created reservation  \* @throws NoAvailableSpotException If no suitable spot is available  \* @throws InvalidReservationException If the reservation time period is invalid  \*/  public Reservation makeReservation(Vehicle vehicle, Date startTime, Date endTime)  throws NoAvailableSpotException, InvalidReservationException {  if (startTime.after(endTime) || startTime.equals(endTime)) {  throw new InvalidReservationException("Invalid reservation time period");  }  ParkingSpot spot = findAvailableSpotForReservation(vehicle.getType(), startTime, endTime);  if (spot == null) {  throw new NoAvailableSpotException("No available spot for reservation");  }  Reservation reservation = new Reservation(spot, vehicle, startTime, endTime);  spot.reserve(reservation);  reservations.add(reservation);  logTransaction("Reservation made for vehicle: " + vehicle.getLicensePlate() + " from " + startTime + " to " + endTime);  return reservation;  }   /\*\*  \* Checks if a vehicle is currently parked in the parking lot.  \*  \* @param licensePlate The license plate of the vehicle  \* @return True if the vehicle is parked, false otherwise  \*/  public boolean isVehicleParked(String licensePlate) {  return activeVehicles.containsKey(licensePlate.toUpperCase());  }   /\*\*  \* Checks if a reservation conflicts with existing reservations or active vehicles.  \*  \* @param vehicle The vehicle to check for conflicts  \* @param startTime The start time of the proposed reservation  \* @param endTime The end time of the proposed reservation  \* @return True if there is a conflict, false otherwise  \*/  public boolean hasReservationConflict(Vehicle vehicle, Date startTime, Date endTime) {  for (Reservation res : reservations) {  if (res.getVehicle().getLicensePlate().equalsIgnoreCase(vehicle.getLicensePlate()) &&  res.isActive() &&  !(endTime.before(res.getStartTime()) || startTime.after(res.getEndTime()))) {  return true;  }  }  return activeVehicles.containsKey(vehicle.getLicensePlate());  }   /\*\*  \* Finds an active reservation by the vehicle's license plate.  \*  \* @param licensePlate The license plate of the vehicle  \* @return The active reservation, or null if none is found  \*/  public Reservation findReservationByLicensePlate(String licensePlate) {  for (Reservation res : reservations) {  if (res.getVehicle().getLicensePlate().equalsIgnoreCase(licensePlate) &&  res.isActive() &&  res.getStatus().equals(Reservation.STATUS\_PENDING)) {  return res;  }  }  return null;  }   /\*\*  \* Cancels a reservation by the vehicle's license plate.  \*  \* @param licensePlate The license plate of the vehicle  \* @return True if the reservation was canceled, false if no reservation was found  \*/  public boolean cancelReservationByLicensePlate(String licensePlate) {  Reservation reservation = findReservationByLicensePlate(licensePlate);  if (reservation == null) {  return false;  }  reservation.setStatus(Reservation.STATUS\_CANCELLED);  reservation.getSpot().cancelReservation();  logTransaction("Reservation cancelled for vehicle: " + licensePlate);  return true;  }   /\*\*  \* Retrieves all active reservations in the parking lot.  \*  \* @return A list of active reservations  \*/  public List<Reservation> getAllActiveReservations() {  List<Reservation> activeReservations = new ArrayList<>();  for (Reservation res : reservations) {  if (res.isActive()) {  activeReservations.add(res);  }  }  return activeReservations;  }   /\*\*  \* Retrieves all security logs for vehicles in the parking lot.  \*  \* @return A map of vehicle license plates to their security logs  \*/  public Map<String, List<EntryExitLog>> getSecurityLogs() {  return new HashMap<>(securityLogs);  }   /\*\*  \* Retrieves security logs for a specific date.  \*  \* @param date The date to filter logs by  \* @return A list of security logs for the specified date  \*/  public List<EntryExitLog> getSecurityLogsByDate(Date date) {  List<EntryExitLog> logs = new ArrayList<>();  Calendar cal = Calendar.getInstance();  cal.setTime(date);  int targetDay = cal.get(Calendar.DAY\_OF\_MONTH);  int targetMonth = cal.get(Calendar.MONTH);  int targetYear = cal.get(Calendar.YEAR);   for (List<EntryExitLog> logList : securityLogs.values()) {  for (EntryExitLog log : logList) {  cal.setTime(log.getEntryTime());  if (cal.get(Calendar.DAY\_OF\_MONTH) == targetDay &&  cal.get(Calendar.MONTH) == targetMonth &&  cal.get(Calendar.YEAR) == targetYear) {  logs.add(log);  }  }  }  return logs;  }   /\*\*  \* Counts the number of available parking spots for a specific spot type.  \*  \* @param spotType The type of parking spot  \* @return The number of available spots of the specified type  \*/  public int getAvailableSpotsByType(String spotType) {  int count = 0;  for (ParkingSpot spot : spots) {  if (spot.getType().equals(spotType) && spot.isAvailable()) {  count++;  }  }  return count;  }   /\*\*  \* Retrieves the subscription manager for the parking lot.  \*  \* @return The subscription manager instance  \*/  public SubscriptionManager getSubscriptionManager() {  return subscriptionManager;  }   /\*\*  \* Retrieves all payment transactions recorded in the parking lot.  \*  \* @return A list of payment transactions  \*/  public List<Payment> getAllTransactions() {  return new ArrayList<>(transactions);  }   /\*\*  \* Finds an available parking spot for a given vehicle type.  \*  \* @param vehicleType The type of vehicle  \* @return An available parking spot, or null if none is found  \*/  private ParkingSpot findAvailableSpot(String vehicleType) {  for (ParkingSpot spot : spots) {  if (spot.isAvailable() && spot.canFit(vehicleType)) {  return spot;  }  }  return null;  }   /\*\*  \* Finds an available parking spot for a reservation for a given vehicle type and time period.  \*  \* @param vehicleType The type of vehicle  \* @param startTime The start time of the reservation  \* @param endTime The end time of the reservation  \* @return An available parking spot, or null if none is found  \*/  private ParkingSpot findAvailableSpotForReservation(String vehicleType, Date startTime, Date endTime) {  for (ParkingSpot spot : spots) {  if (spot.isAvailableForReservation(startTime, endTime) && spot.canFit(vehicleType)) {  return spot;  }  }  return null;  }   /\*\*  \* Saves the parking lot data to a file.  \*  \* @param filePath The path to the file where data will be saved  \* @throws IOException If an I/O error occurs  \*/  public void saveDataToFile(String filePath) throws IOException {  try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filePath))) {  oos.writeObject(this);  }  }   /\*\*  \* Loads parking lot data from a file.  \*  \* @param filePath The path to the file containing the data  \* @return The loaded ParkingLot instance  \* @throws IOException If an I/O error occurs  \* @throws ClassNotFoundException If the class of a serialized object cannot be found  \*/  public static ParkingLot loadDataFromFile(String filePath) throws IOException, ClassNotFoundException {  try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filePath))) {  ParkingLot lot = (ParkingLot) ois.readObject();  lot.dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm");  return lot;  }  }   /\*\*  \* Logs a transaction with the specified details.  \*  \* @param transactionDetails The details of the transaction to log  \*/  @Override  public void logTransaction(String transactionDetails) {  System.out.println("Transaction Log: " + transactionDetails);  }   /\*\*  \* Retrieves the name of the parking lot.  \*  \* @return The name of the parking lot  \*/  public String getName() {  return name;  }   /\*\*  \* Retrieves the list of parking spots in the parking lot.  \*  \* @return The list of parking spots  \*/  public List<ParkingSpot> getSpots() {  return spots;  }   /\*\*  \* Retrieves the list of reservations in the parking lot.  \*  \* @return The list of reservations  \*/  public List<Reservation> getReservations() {  return reservations;  } } |
| --- |

* **PricingManager:** Calculates parking and subscription fees dynamically.

| package com.parkinglot.system.management;  import com.parkinglot.system.model.ParkingSpot; import com.parkinglot.system.model.Vehicle; import java.io.Serializable; import java.time.LocalDateTime; import java.time.DayOfWeek; import java.util.HashMap; import java.util.Map;  /\*\*  \* Manages pricing calculations for parking fees in the parking lot system.  \* Calculates fees based on vehicle type, parking spot type, duration, and dynamic factors like peak hours or weekends.  \* Implements Serializable to allow the pricing configuration to be saved or transmitted.  \*/ public class PricingManager implements Serializable {   private static final long serialVersionUID = 1L;   private Map<String, Double> baseRates;  private Map<String, Double> spotTypeMultipliers;  private Double peakHourSurcharge;  private Double weekendSurcharge;   /\*\*  \* Constructs a PricingManager with default pricing rates and surcharges.  \*/  public PricingManager() {  initializeDefaultRates();  }   /\*\*  \* Constructs a PricingManager with specified base rates and spot type multipliers.  \*  \* @param baseRates The base rates for different vehicle types  \* @param spotTypeMultipliers The multipliers for different parking spot types  \*/  public PricingManager(Map<String, Double> baseRates, Map<String, Double> spotTypeMultipliers) {  this.baseRates = new HashMap<>(baseRates);  this.spotTypeMultipliers = new HashMap<>(spotTypeMultipliers);  this.peakHourSurcharge = 1.5;  this.weekendSurcharge = 1.2;  }   /\*\*  \* Initializes default pricing rates and multipliers for vehicle and spot types.  \*/  private void initializeDefaultRates() {  baseRates = new HashMap<>();  baseRates.put(Vehicle.TYPE\_CAR, 40.0);  baseRates.put(Vehicle.TYPE\_BIKE, 20.0);  baseRates.put(Vehicle.TYPE\_TRUCK, 80.0);  baseRates.put(Vehicle.TYPE\_BUS, 100.0);   spotTypeMultipliers = new HashMap<>();  spotTypeMultipliers.put(ParkingSpot.TYPE\_CAR, 1.0);  spotTypeMultipliers.put(ParkingSpot.TYPE\_BIKE, 0.5);  spotTypeMultipliers.put(ParkingSpot.TYPE\_TRUCK, 1.5);  spotTypeMultipliers.put(ParkingSpot.TYPE\_ELECTRIC, 1.2);  spotTypeMultipliers.put(ParkingSpot.TYPE\_HANDICAPPED, 0.8);   peakHourSurcharge = 1.5;  weekendSurcharge = 1.2;  }   /\*\*  \* Calculates the parking fee based on vehicle type, parking duration, and spot type.  \*  \* @param vehicleType The type of vehicle  \* @param hours The duration of parking in hours  \* @param spotType The type of parking spot  \* @return The calculated parking fee  \* @throws IllegalArgumentException If vehicle type, spot type, or duration is invalid  \*/  public Double calculateFee(String vehicleType, Double hours, String spotType) {  validateType(vehicleType, spotType);  Double baseRate = baseRates.getOrDefault(vehicleType, 50.0);  Double spotMultiplier = spotTypeMultipliers.getOrDefault(spotType, 1.0);  Double duration = hours != null ? hours : 0.0;  if (duration < 0) {  throw new IllegalArgumentException("Parking duration cannot be negative");  }  Double fee = baseRate \* duration \* spotMultiplier;   LocalDateTime now = LocalDateTime.now();  int hour = now.getHour();  boolean isPeakHour = (hour >= 8 && hour <= 10) || (hour >= 17 && hour <= 19);  boolean isWeekend = now.getDayOfWeek() == DayOfWeek.SATURDAY || now.getDayOfWeek() == DayOfWeek.SUNDAY;   if (isPeakHour && !isWeekend) {  fee \*= peakHourSurcharge;  } else if (isWeekend) {  fee \*= weekendSurcharge;  }   return fee;  }   /\*\*  \* Validates the vehicle type and spot type for pricing calculations.  \*  \* @param vehicleType The type of vehicle  \* @param spotType The type of parking spot  \* @throws IllegalArgumentException If vehicle type or spot type is invalid  \*/  private void validateType(String vehicleType, String spotType) {  if (vehicleType == null || !(vehicleType.equals(Vehicle.TYPE\_CAR) || vehicleType.equals(Vehicle.TYPE\_BIKE) ||  vehicleType.equals(Vehicle.TYPE\_TRUCK) || vehicleType.equals(Vehicle.TYPE\_BUS))) {  throw new IllegalArgumentException("Invalid vehicle type: " + vehicleType);  }  if (spotType == null || !(spotType.equals(ParkingSpot.TYPE\_CAR) || spotType.equals(ParkingSpot.TYPE\_BIKE) ||  spotType.equals(ParkingSpot.TYPE\_TRUCK) || spotType.equals(ParkingSpot.TYPE\_ELECTRIC) ||  spotType.equals(ParkingSpot.TYPE\_HANDICAPPED))) {  throw new IllegalArgumentException("Invalid spot type: " + spotType);  }  }   /\*\*  \* Calculates the parking fee for a variable number of hours.  \*  \* @param vehicleType The type of vehicle  \* @param spotType The type of parking spot  \* @param hours Variable number of hours to sum for the total duration  \* @return The calculated parking fee  \*/  public Double calculateFee(String vehicleType, String spotType, Double... hours) {  Double totalHours = 0.0;  for (Double hour : hours) {  totalHours += hour != null ? hour : 0.0;  }  return calculateFee(vehicleType, totalHours, spotType);  }   /\*\*  \* Calculates the monthly subscription fee based on a daily rate for a vehicle and spot type.  \*  \* @param vehicleType The type of vehicle  \* @param spotType The type of parking spot  \* @return The calculated monthly subscription fee  \*/  public double calculateMonthlySubscriptionFee(String vehicleType, String spotType) {  Double dailyRate = calculateFee(vehicleType, 8.0, spotType);  return dailyRate \* 22 \* 0.7;  }   /\*\*  \* Calculates parking fees for multiple vehicle types for a given duration and spot type.  \*  \* @param hours The duration of parking in hours  \* @param spotType The type of parking spot  \* @param vehicleTypes The types of vehicles  \* @return An array of calculated fees for each vehicle type  \*/  public Double[] calculateFeesForTypes(Double hours, String spotType, String... vehicleTypes) {  Double[] fees = new Double[vehicleTypes.length];  for (int i = 0; i < vehicleTypes.length; i++) {  fees[i] = calculateFee(vehicleTypes[i], hours, spotType);  }  return fees;  }   /\*\*  \* Sets the base rate for a specific vehicle type.  \*  \* @param vehicleType The type of vehicle  \* @param rate The base rate to set  \* @throws IllegalArgumentException If the vehicle type is invalid  \*/  public void setBaseRate(String vehicleType, Double rate) {  validateType(vehicleType, ParkingSpot.TYPE\_CAR);  baseRates.put(vehicleType, rate);  }   /\*\*  \* Sets the multiplier for a specific parking spot type.  \*  \* @param spotType The type of parking spot  \* @param multiplier The multiplier to set  \* @throws IllegalArgumentException If the spot type is invalid  \*/  public void setSpotTypeMultiplier(String spotType, Double multiplier) {  validateType(Vehicle.TYPE\_CAR, spotType);  spotTypeMultipliers.put(spotType, multiplier);  }   /\*\*  \* Sets the surcharge applied during peak hours.  \*  \* @param surcharge The peak hour surcharge to set  \*/  public void setPeakHourSurcharge(Double surcharge) {  this.peakHourSurcharge = surcharge;  }   /\*\*  \* Sets the surcharge applied during weekends.  \*  \* @param surcharge The weekend surcharge to set  \*/  public void setWeekendSurcharge(Double surcharge) {  this.weekendSurcharge = surcharge;  }   /\*\*  \* Retrieves the base rate for a specific vehicle type.  \*  \* @param vehicleType The type of vehicle  \* @return The base rate, or 0.0 if not found  \*/  public Double getBaseRate(String vehicleType) {  return baseRates.getOrDefault(vehicleType, 0.0);  }   /\*\*  \* Retrieves the multiplier for a specific parking spot type.  \*  \* @param spotType The type of parking spot  \* @return The multiplier, or 1.0 if not found  \*/  public Double getSpotTypeMultiplier(String spotType) {  return spotTypeMultipliers.getOrDefault(spotType, 1.0);  } } |
| --- |

* **SubscriptionManager:** Manages subscription creation, renewal, and cancellation.

| package com.parkinglot.system.management;  import com.parkinglot.system.model.Subscription; import com.parkinglot.system.model.Vehicle; import com.parkinglot.system.payment.Payment; import com.parkinglot.system.payment.PaymentMethod;  import java.io.Serializable; import java.util.\*;  /\*\*  \* Manages parking subscriptions for vehicles in the parking lot system.  \* Handles subscription creation, renewal, cancellation, and payment processing.  \* Implements Serializable to support persistence.  \*/ public class SubscriptionManager implements Serializable {  private static final long serialVersionUID = 1L;   private Map<String, Subscription> subscriptions;  private PricingManager pricingManager;  private ParkingLot parkingLot; // Reference to ParkingLot for transaction logging   /\*\*  \* Constructs a SubscriptionManager with a PricingManager and ParkingLot.  \*  \* @param pricingManager The pricing manager for calculating fees  \* @param parkingLot The parking lot for logging transactions  \*/  public SubscriptionManager(PricingManager pricingManager, ParkingLot parkingLot) {  this.subscriptions = new HashMap<>();  this.pricingManager = pricingManager;  this.parkingLot = parkingLot;  }   /\*\*  \* Registers a new subscription for a vehicle.  \*  \* @param vehicle The vehicle to subscribe  \* @param type The subscription type (e.g., MONTHLY, ANNUAL)  \* @param spotType The parking spot type (e.g., CAR, ELECTRIC)  \* @param paymentMethod The payment method for the subscription  \* @return True if the subscription is successfully registered, false otherwise  \*/  public boolean registerSubscription(Vehicle vehicle, String type, String spotType, PaymentMethod paymentMethod) {  double fee = calculateSubscriptionFee(vehicle, type, spotType);  Payment payment = new Payment(fee, vehicle, null);  boolean paymentSuccess = payment.processPayment(paymentMethod);   if (paymentSuccess) {  Calendar calendar = Calendar.getInstance();  Date startDate = new Date();  calendar.setTime(startDate);  int months = switch (type) {  case Subscription.TYPE\_QUARTERLY -> 3;  case Subscription.TYPE\_SEMI\_ANNUAL -> 6;  case Subscription.TYPE\_ANNUAL -> 12;  default -> 1; // MONTHLY  };  calendar.add(Calendar.MONTH, months);  Date endDate = calendar.getTime();   Subscription subscription = new Subscription(vehicle, startDate, endDate, type, spotType);  subscriptions.put(vehicle.getLicensePlate(), subscription);  parkingLot.getAllTransactions().add(payment); // Add payment to transactions  parkingLot.logTransaction("Subscription registered for vehicle: " + vehicle.getLicensePlate() +  " type: " + type + " amount: Rs. " + fee);  }  return paymentSuccess;  }   /\*\*  \* Calculates the subscription fee for a vehicle.  \*  \* @param vehicle The vehicle  \* @param type The subscription type  \* @param spotType The parking spot type  \* @return The subscription fee  \*/  public double calculateSubscriptionFee(Vehicle vehicle, String type, String spotType) {  double baseFee = pricingManager.calculateMonthlySubscriptionFee(vehicle.getType(), spotType);  return switch (type) {  case Subscription.TYPE\_QUARTERLY -> baseFee \* 3 \* 0.9; // 10% discount  case Subscription.TYPE\_SEMI\_ANNUAL -> baseFee \* 6 \* 0.85; // 15% discount  case Subscription.TYPE\_ANNUAL -> baseFee \* 12 \* 0.8; // 20% discount  default -> baseFee; // MONTHLY  };  }   /\*\*  \* Processes payment for a subscription.  \*  \* @param vehicle The vehicle  \* @param type The subscription type  \* @param spotType The parking spot type  \* @param paymentMethod The payment method  \* @return True if payment is successful, false otherwise  \*/  public boolean processSubscriptionPayment(Vehicle vehicle, String type, String spotType, PaymentMethod paymentMethod) {  boolean success = registerSubscription(vehicle, type, spotType, paymentMethod);  if (!success) {  parkingLot.logTransaction("Subscription payment failed for vehicle: " + vehicle.getLicensePlate() +  " type: " + type + " amount: Rs. " + calculateSubscriptionFee(vehicle, type, spotType));  }  return success;  }   /\*\*  \* Renews a subscription by extending its end date.  \*  \* @param licensePlate The vehicle's license plate  \* @param months The number of months to extend  \*/  public void renewSubscription(String licensePlate, int months) {  Subscription subscription = subscriptions.get(licensePlate.toUpperCase());  if (subscription != null && subscription.isActive()) {  subscription.renew(months);  parkingLot.logTransaction("Subscription renewed for vehicle: " + licensePlate + " for " + months + " months");  }  }   /\*\*  \* Cancels a subscription.  \*  \* @param licensePlate The vehicle's license plate  \*/  public void cancelSubscription(String licensePlate) {  Subscription subscription = subscriptions.get(licensePlate.toUpperCase());  if (subscription != null) {  subscription.cancel();  parkingLot.logTransaction("Subscription cancelled for vehicle: " + licensePlate);  }  }   /\*\*  \* Checks if a vehicle has an active subscription.  \*  \* @param licensePlate The vehicle's license plate  \* @return True if the vehicle has an active subscription, false otherwise  \*/  public boolean hasActiveSubscription(String licensePlate) {  Subscription subscription = subscriptions.get(licensePlate.toUpperCase());  return subscription != null && subscription.isActive();  }   /\*\*  \* Retrieves a subscription by license plate.  \*  \* @param licensePlate The vehicle's license plate  \* @return The subscription, or null if not found  \*/  public Subscription getSubscription(String licensePlate) {  return subscriptions.get(licensePlate.toUpperCase());  }   /\*\*  \* Retrieves all active subscriptions.  \*  \* @return A list of active subscriptions  \*/  public List<Subscription> getAllActiveSubscriptions() {  List<Subscription> activeSubscriptions = new ArrayList<>();  for (Subscription sub : subscriptions.values()) {  if (sub.isActive()) {  activeSubscriptions.add(sub);  }  }  return activeSubscriptions;  } } |
| --- |

**Com.parkinglot.system.payment**

* **Payment:** Represents a payment transaction for a parking session.

| package com.parkinglot.system.payment;  import com.parkinglot.system.model.Vehicle; import com.parkinglot.system.model.EntryExitLog; import java.io.Serializable; import java.util.Date;  /\*\*  \* Represents a payment transaction for a parking session in the parking lot system.  \* Tracks the payment amount, vehicle, parking session details, and payment status.  \* Implements Serializable to allow the payment state to be saved or transmitted.  \*/ public class Payment implements Serializable {  private static final long serialVersionUID = 1L;   private Double amount; // Changed to wrapper class Double  private Vehicle vehicle;  private EntryExitLog parkingSession;  private Date paymentTime;  private PaymentStatus status;  private PaymentMethod method;   /\*\*  \* Constructs a new payment for a parking session.  \* Initializes the payment as pending with the current timestamp.  \*  \* @param amount The payment amount  \* @param vehicle The vehicle being charged  \* @param parkingSession The parking session log  \*/  public Payment(Double amount, Vehicle vehicle, EntryExitLog parkingSession) {  this.amount = amount != null ? amount : 0.0; // Handle null amount  this.vehicle = vehicle;  this.parkingSession = parkingSession;  this.paymentTime = new Date();  this.status = PaymentStatus.PENDING;  this.method = null;  }   /\*\*  \* Processes the payment using the specified payment method.  \* Simulates interaction with a payment gateway and updates the payment status.  \*  \* @param method The payment method to use (e.g., Credit Card, UPI)  \* @return True if the payment is successful, false otherwise  \*/  public boolean processPayment(PaymentMethod method) {  this.method = method;  if (amount == null) {  System.out.println("Error: Payment amount is not specified.");  this.status = PaymentStatus.FAILED;  return false;  }  boolean success = method.processTransaction(amount);  this.status = success ? PaymentStatus.COMPLETED : PaymentStatus.FAILED;  return success;  }   /\*\*  \* Returns the payment amount.  \*  \* @return The amount to be paid  \*/  public Double getAmount() {  return amount;  }   /\*\*  \* Returns the vehicle associated with the payment.  \*  \* @return The vehicle  \*/  public Vehicle getVehicle() {  return vehicle;  }   /\*\*  \* Returns the parking session log for this payment.  \*  \* @return The parking session log  \*/  public EntryExitLog getParkingSession() {  return parkingSession;  }   /\*\*  \* Returns the timestamp when the payment was initiated.  \*  \* @return The payment timestamp  \*/  public Date getPaymentTime() {  return paymentTime;  }   /\*\*  \* Returns the current status of the payment.  \*  \* @return The payment status (e.g., PENDING, COMPLETED)  \*/  public PaymentStatus getStatus() {  return status;  }   /\*\*  \* Returns the payment method used for the transaction.  \*  \* @return The payment method, or null if not yet processed  \*/  public PaymentMethod getMethod() {  return method;  } }  /\*\*  \* Enum defining the possible statuses of a payment transaction.  \*/ enum PaymentStatus {  PENDING, // Payment is initiated but not yet processed  COMPLETED, // Payment was successfully processed  FAILED, // Payment processing failed  REFUNDED // Payment was refunded to the customer } |
| --- |

* **PaymentMethod:** Defines a contract for payment processing.

| package com.parkinglot.system.payment; /\*\*  \* Defines a contract for processing payment transactions in the parking lot system.  \* Implementations handle specific payment methods, such as credit cards or UPI.  \*/ public interface PaymentMethod {   /\*\*  \* Processes a payment transaction for the specified amount.  \* Implementations should connect to a payment gateway or simulate the transaction.  \*  \* @param amount The amount to be charged  \* @return True if the transaction is successful, false otherwise  \*/  boolean processTransaction(double amount);   /\*\*  \* Retrieves details about the payment method, such as card number or UPI ID.  \* Useful for logging or displaying transaction information.  \*  \* @return A string containing the payment method details  \*/  String getPaymentDetails(); } |
| --- |

* **CreditCardPayment:** Implements credit card payment processing.

| package com.parkinglot.system.payment;  import java.io.Serializable;  /\*\*  \* Represents a Credit Card payment method for processing transactions.  \* Implements PaymentMethod and Loggable to support payment processing and transaction logging.  \*/ public class CreditCardPayment implements PaymentMethod, Loggable, Serializable {  private static final long serialVersionUID = 1L;   private String cardNumber;  private String expiryDate;  private String cvv;  private String cardHolderName;   /\*\*  \* Creates a CreditCardPayment with the specified details.  \*  \* @param cardNumber The credit card number  \* @param expiryDate The expiry date (MM/YY)  \* @param cvv The CVV code  \* @param cardHolderName The cardholder's name  \*/  public CreditCardPayment(String cardNumber, String expiryDate, String cvv, String cardHolderName) {  this.cardNumber = cardNumber;  this.expiryDate = expiryDate;  this.cvv = cvv;  this.cardHolderName = cardHolderName;  }   /\*\*  \* Processes a credit card transaction for the specified amount.  \* Logs the transaction details upon processing.  \*  \* @param amount The transaction amount  \* @return true if the transaction is successful, false otherwise  \*/  @Override  public boolean processTransaction(double amount) {  System.out.println("Processing credit card payment of Rs. " + amount);  String details = "Credit card payment of Rs. " + amount + " processed for card ending in " +  cardNumber.substring(cardNumber.length() - 4);  logTransaction(details);  return isValidCard();  }   /\*\*  \* Retrieves the payment details for this credit card.  \*  \* @return A string containing the masked card number  \*/  @Override  public String getPaymentDetails() {  return "Credit Card: \*\*\*\* \*\*\*\* \*\*\*\* " + cardNumber.substring(cardNumber.length() - 4);  }   /\*\*  \* Logs the transaction details to the console.  \* Can be extended to log to a file or database.  \*  \* @param transactionDetails The details of the transaction  \*/  @Override  public void logTransaction(String transactionDetails) {  System.out.println("Transaction Log: " + transactionDetails);  // Future extension: Write to a log file or database  }   /\*\*  \* Validates the credit card details.  \*  \* @return true if the card is valid, false otherwise  \*/  private boolean isValidCard() {  return cardNumber != null && cardNumber.length() == 16 && cvv != null && cvv.length() == 3;  } } |
| --- |

* **UPIPayment:** Implements UPI payment processing.

| package com.parkinglot.system.payment; import java.io.Serializable;  /\*\*  \* Represents a UPI (Unified Payments Interface) payment method for processing transactions.  \* Implements the PaymentMethod interface and is serializable for persistence.  \*/ public class UPIPayment implements PaymentMethod, Serializable {  private static final long serialVersionUID = 1L;   // The UPI ID used for transactions (e.g., user@bank)  private final String upiId;   /\*\*  \* Creates a UPIPayment instance with the specified UPI ID.  \*  \* @param upiId The UPI ID for the payment method  \*/  public UPIPayment(String upiId) {  this.upiId = upiId;  }   /\*\*  \* Processes a UPI transaction for the specified amount.  \* Currently simulates a connection to a UPI gateway and validates the UPI ID.  \*  \* @param amount The transaction amount in dollars  \* @return true if the transaction is successful, false otherwise  \*/  @Override  public boolean processTransaction(double amount) {  // Log the transaction attempt (in a real application, this would connect to a UPI gateway)  System.out.println("Processing UPI payment of Rs. " + amount);   // Simple validation: checks if UPI ID contains "@" (e.g., user@bank)  return upiId != null && upiId.contains("@");  }   /\*\*  \* Retrieves the payment details for this UPI payment method.  \*  \* @return A string containing the UPI ID  \*/  @Override  public String getPaymentDetails() {  return "UPI ID: " + upiId;  } } |
| --- |

* **Loggable:** Defines a contract for transaction logging.

| package com.parkinglot.system.payment;  /\*\*  \* Interface for logging transaction details.  \*/ public interface Loggable {  /\*\*  \* Logs the transaction details to a specified destination.  \*  \* @param transactionDetails The details of the transaction  \*/  void logTransaction(String transactionDetails); } |
| --- |

**Com.parkinglot.system.exceptions**

* **NoAvailableSpotException:** Thrown when no suitable parking spot is available.

| package com.parkinglot.system.exceptions; import java.io.Serializable;  /\*\*  \* A custom exception thrown when no parking spots are available in the parking lot.  \* This exception helps handle situations where a vehicle cannot be parked due to full occupancy.  \* Implements Serializable to allow the exception to be saved or transmitted.  \*/ public class NoAvailableSpotException extends Exception implements Serializable {   // Unique identifier for serialization, ensuring compatibility during deserialization  private static final long serialVersionUID = 1L;   /\*\*  \* Constructs a NoAvailableSpotException with a specific error message.  \*  \* @param message A description of why no parking spots are available  \*/  public NoAvailableSpotException(String message) {  super(message);  }   /\*\*  \* Constructs a NoAvailableSpotException with an error message and a cause.  \* Useful for wrapping other exceptions to provide additional context about the failure.  \*  \* @param message A description of why no parking spots are available  \* @param cause The underlying exception that triggered this one  \*/  public NoAvailableSpotException(String message, Throwable cause) {  super(message, cause);  } } |
| --- |

* **InvalidReservationException:** Thrown for invalid reservation attempts.

| package com.parkinglot.system.exceptions; import java.io.Serializable;  /\*\*  \* A custom exception thrown when a parking reservation is invalid.  \* This exception helps handle errors like invalid dates, unavailable spots, or incorrect reservation details.  \* Implements Serializable to allow the exception to be saved or transmitted.  \*/ public class InvalidReservationException extends Exception implements Serializable {   // Unique identifier for serialization, ensuring compatibility during deserialization  private static final long serialVersionUID = 1L;   /\*\*  \* Constructs an InvalidReservationException with a specific error message.  \*  \* @param message A description of what went wrong with the reservation  \*/  public InvalidReservationException(String message) {  super(message);  }   /\*\*  \* Constructs an InvalidReservationException with an error message and a cause.  \* Useful for wrapping other exceptions to provide more context.  \*  \* @param message A description of what went wrong with the reservation  \* @param cause The underlying exception that triggered this one  \*/  public InvalidReservationException(String message, Throwable cause) {  super(message, cause);  } } |
| --- |

* **VehicleNotFoundException:** Thrown when a vehicle is not found in the system.

| package com.parkinglot.system.exceptions; import java.io.Serializable;  /\*\*  \* Custom exception thrown when a vehicle is not found in the system.  \* Implements Serializable to support persistence.  \*/ public class VehicleNotFoundException extends Exception implements Serializable {  private static final long serialVersionUID = 1L;   /\*\*  \* Constructs a VehicleNotFoundException with the specified message.  \*  \* @param message The detail message explaining why the vehicle was not found  \*/  public VehicleNotFoundException(String message) {  super(message);  }   /\*\*  \* Constructs a VehicleNotFoundException with the specified message and cause.  \*  \* @param Alphabetic message The detail message explaining why the vehicle was not found  \* @param cause The underlying cause of the exception  \*/  public VehicleNotFoundException(String message, Throwable cause) {  super(message, cause);  } } |
| --- |

**Com.parkinglot.system.ui**

* **ParkingLotSystem:** Provides the command-line interface for user interaction.

| package com.parkinglot.system.ui;  import com.parkinglot.system.exceptions.InvalidReservationException; import com.parkinglot.system.exceptions.NoAvailableSpotException; import com.parkinglot.system.exceptions.VehicleNotFoundException; import com.parkinglot.system.management.ParkingLot; import com.parkinglot.system.model.\*; import com.parkinglot.system.payment.\*;  import java.io.\*; import java.util.\*; import java.text.SimpleDateFormat; import java.text.ParseException;  /\*\*  \* A command-line interface for managing a parking lot system.  \* Provides options to park vehicles, make reservations, manage subscriptions, view logs, and more.  \* Implements Serializable to save and load the parking lot state.  \*/ public class ParkingLotSystem implements Serializable {   private static final long serialVersionUID = 1L;  private static final String DATA\_FILE = "parking\_lot\_data.dat";   /\*\*  \* Main entry point for the parking lot system, initializing and running the command-line interface.  \*  \* @param args Command-line arguments (not used)  \*/  public static void main(String[] args) {  try {  ParkingLot parkingLot = initializeParkingLot();  Scanner scanner = new Scanner(System.in);  SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy HH:mm");   boolean running = true;  while (running) {  displayMainMenu();  int option = getIntInput(scanner, "Choose an option: ", 1, 11);   switch (option) {  case 1 -> parkVehicle(parkingLot, scanner);  case 2 -> makeReservation(parkingLot, scanner, dateFormat);  case 3 -> useReservation(parkingLot, scanner, dateFormat);  case 4 -> cancelReservation(parkingLot, scanner);  case 5 -> checkActiveReservations(parkingLot, dateFormat);  case 6 -> vehicleExit(parkingLot, scanner);  case 7 -> checkAvailability(parkingLot);  case 8 -> manageSubscriptions(parkingLot, scanner);  case 9 -> viewSecurityLogs(parkingLot, scanner, dateFormat);  case 10 -> viewAllTransactions(parkingLot, dateFormat);  case 11 -> {  saveAndExit(parkingLot);  running = false;  }  default -> System.out.println("Invalid option. Please try again.");  }  }  scanner.close();   } catch (Exception e) {  System.err.println("An error occurred: " + e.getMessage());  e.printStackTrace();  }  }   /\*\*  \* Initializes the parking lot by loading existing data or creating a new instance.  \*  \* @return The initialized ParkingLot instance  \*/  private static ParkingLot initializeParkingLot() {  File file = new File(DATA\_FILE);  ParkingLot parkingLot;   if (file.exists()) {  try {  parkingLot = ParkingLot.loadDataFromFile(DATA\_FILE);  System.out.println("Parking lot data loaded successfully.");  } catch (IOException | ClassNotFoundException e) {  System.out.println("Could not load parking lot data: " + quote(e.getMessage()));  parkingLot = new ParkingLot("City Center Parking", 100);  }  } else {  parkingLot = new ParkingLot("City Center Parking", 100);  }  return parkingLot;  }   /\*\*  \* Displays the main menu options for the parking lot system.  \*/  private static void displayMainMenu() {  System.out.println("\n===== PARKING LOT MANAGEMENT SYSTEM =====");  System.out.println("1. Park a vehicle");  System.out.println("2. Make a reservation");  System.out.println("3. Use existing reservation");  System.out.println("4. Cancel reservation");  System.out.println("5. Check active reservations");  System.out.println("6. Vehicle exit and payment");  System.out.println("7. Check parking availability");  System.out.println("8. Subscription management");  System.out.println("9. View security logs");  System.out.println("10. View all transactions");  System.out.println("11. Exit program");  }   /\*\*  \* Saves the parking lot data and exits the program.  \*  \* @param parkingLot The ParkingLot instance to save  \*/  private static void saveAndExit(ParkingLot parkingLot) {  try {  parkingLot.saveDataToFile(DATA\_FILE);  System.out.println("All data saved successfully.");  } catch (IOException e) {  System.out.println("Error saving data: " + quote(e.getMessage()));  }  System.out.println("Thank you for using the Parking Lot Management System!");  }   /\*\*  \* Handles the process of parking a vehicle in the parking lot.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void parkVehicle(ParkingLot parkingLot, Scanner scanner) {  try {  System.out.println("\n-- Park a Vehicle --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   Vehicle vehicle = createVehicle(scanner, licensePlate);  if (vehicle == null) return;   if (parkingLot.isVehicleParked(licensePlate)) {  System.out.println("Error: A vehicle with this license plate is already parked.");  return;  }   ParkingSpot spot = parkingLot.checkIn(vehicle);  System.out.println("Vehicle parked successfully!");  System.out.println("Spot assigned: " + spot.getSpotId());   } catch (NoAvailableSpotException e) {  System.out.println("Error: " + quote(e.getMessage()));  } catch (Exception e) {  System.out.println("An unexpected error occurred: " + quote(e.getMessage()));  }  }   /\*\*  \* Creates a vehicle instance based on user input.  \*  \* @param scanner The Scanner for user input  \* @param licensePlate The license plate of the vehicle  \* @return The created Vehicle instance, or null if invalid  \*/  private static Vehicle createVehicle(Scanner scanner, String licensePlate) {  System.out.println("Vehicle type:");  System.out.println("1. Car");  System.out.println("2. Bike");  System.out.println("3. Truck");  System.out.println("4. Electric Vehicle");   int vehicleType = getIntInput(scanner, "Choose vehicle type: ", 1, 4);  return switch (vehicleType) {  case 1 -> {  boolean isLuxury = getStringInput(scanner, "Is it a luxury car? (y/n): ").equalsIgnoreCase("y");  yield new Car(licensePlate, isLuxury);  }  case 2 -> new Bike(licensePlate);  case 3 -> {  double weight = getDoubleInput(scanner, "Enter truck weight in tons: ", 0.1, 50.0);  yield new Truck(licensePlate, weight);  }  case 4 -> {  double batteryCapacity = getDoubleInput(scanner, "Enter battery capacity in kWh: ", 1.0, 200.0);  yield new ElectricVehicle(licensePlate, batteryCapacity);  }  default -> {  System.out.println("Invalid vehicle type.");  yield null;  }  };  }   /\*\*  \* Handles the process of making a parking reservation.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \* @param dateFormat The date format for parsing input  \*/  private static void makeReservation(ParkingLot parkingLot, Scanner scanner, SimpleDateFormat dateFormat) {  try {  System.out.println("\n-- Make a Reservation --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   System.out.println("Vehicle type:");  System.out.println("1. Car");  System.out.println("2. Bike");  System.out.println("3. Truck");   int vehicleType = getIntInput(scanner, "Choose vehicle type: ", 1, 3);  Vehicle vehicle = switch (vehicleType) {  case 1 -> new Car(licensePlate, false);  case 2 -> new Bike(licensePlate);  case 3 -> new Truck(licensePlate, 1.0);  default -> {  System.out.println("Invalid vehicle type.");  yield null;  }  };  if (vehicle == null) return;   Date startTime = getDateInput(scanner, "Enter start date and time (dd-MM-yyyy HH:mm): ", dateFormat);  Date endTime = getDateInput(scanner, "Enter end date and time (dd-MM-yyyy HH:mm): ", dateFormat);   if (parkingLot.hasReservationConflict(vehicle, startTime, endTime)) {  System.out.println("Error: This vehicle already has a reservation or is parked during this time.");  return;  }   Reservation reservation = parkingLot.makeReservation(vehicle, startTime, endTime);  System.out.println("Reservation made successfully!");  System.out.println("Spot assigned: " + reservation.getSpot().getSpotId());  System.out.println("Start time: " + dateFormat.format(startTime));  System.out.println("End time: " + dateFormat.format(endTime));   } catch (NoAvailableSpotException | InvalidReservationException e) {  System.out.println("Error: " + quote(e.getMessage()));  } catch (Exception e) {  System.out.println("An unexpected error occurred: " + quote(e.getMessage()));  }  }   /\*\*  \* Handles checking in a vehicle using an existing reservation.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \* @param dateFormat The date format for displaying times  \*/  private static void useReservation(ParkingLot parkingLot, Scanner scanner, SimpleDateFormat dateFormat) {  try {  System.out.println("\n-- Use Existing Reservation --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   Reservation reservation = parkingLot.findReservationByLicensePlate(licensePlate);  if (reservation == null) {  System.out.println("No active reservation found for this vehicle.");  return;  }   Date now = new Date();  long tenMinutesInMillis = 10 \* 60 \* 1000;  Date adjustedStartTime = new Date(reservation.getStartTime().getTime() - tenMinutesInMillis);  Date adjustedEndTime = new Date(reservation.getEndTime().getTime() + tenMinutesInMillis);   if (now.before(adjustedStartTime) || now.after(adjustedEndTime)) {  System.out.println("Current time is outside the reservation period (±10 minutes).");  return;  }   ParkingSpot spot = parkingLot.checkIn(reservation.getVehicle(), reservation);  reservation.setStatus(Reservation.STATUS\_CHECKED\_IN);   System.out.println("Vehicle checked in successfully using reservation!");  System.out.println("Spot assigned: " + spot.getSpotId());  System.out.println("Entry time: " + dateFormat.format(now));   } catch (Exception e) {  System.out.println("Error: " + quote(e.getMessage()));  }  }   /\*\*  \* Handles canceling a reservation for a vehicle.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void cancelReservation(ParkingLot parkingLot, Scanner scanner) {  System.out.println("\n-- Cancel Reservation --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   boolean success = parkingLot.cancelReservationByLicensePlate(licensePlate);  System.out.println(success ? "Reservation cancelled successfully."  : "No active reservation found for this vehicle.");  }   /\*\*  \* Displays all active reservations in the parking lot.  \*  \* @param parkingLot The ParkingLot instance  \* @param dateFormat The date format for displaying times  \*/  private static void checkActiveReservations(ParkingLot parkingLot, SimpleDateFormat dateFormat) {  System.out.println("\n-- Active Reservations --");  List<Reservation> activeReservations = parkingLot.getAllActiveReservations();   if (activeReservations.isEmpty()) {  System.out.println("No active reservations found.");  return;  }   System.out.println("------------------------------------------------------------------------------------------");  System.out.printf("%-15s %-15s %-12s %-20s %-20s %-12s\n",  "License Plate", "Vehicle Type", "Spot ID", "Start Time", "End Time", "Status");  System.out.println("------------------------------------------------------------------------------------------");   for (Reservation res : activeReservations) {  System.out.printf("%-15s %-15s %-12s %-20s %-20s %-12s\n",  res.getVehicle().getLicensePlate(),  res.getVehicle().getType(),  res.getSpot().getSpotId(),  dateFormat.format(res.getStartTime()),  dateFormat.format(res.getEndTime()),  res.getStatus());  }  System.out.println("------------------------------------------------------------------------------------------");  }   /\*\*  \* Handles the vehicle checkout and payment process.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void vehicleExit(ParkingLot parkingLot, Scanner scanner) {  try {  System.out.println("\n-- Vehicle Exit and Payment --");  String licensePlate = getStringInput(scanner, "Enter license plate of exiting vehicle: ").toUpperCase();   Vehicle vehicle = new Car(licensePlate); // Temporary vehicle for checkout  Payment payment = parkingLot.prepareCheckOut(vehicle);  if (payment.getAmount() == 0.0)  return;  System.out.println("Vehicle checkout prepared!");  System.out.println("Duration: " + payment.getParkingSession().getParkingDuration() + " hours");  System.out.println("Amount due: Rs. " + payment.getAmount());    PaymentMethod method = selectPaymentMethod(scanner);  if (method == null) return;   boolean paymentSuccess = payment.processPayment(method);  parkingLot.finalizeCheckOut(vehicle, paymentSuccess);   System.out.println(paymentSuccess ? "Payment successful! Thank you for using our parking services."  : "Payment failed. Please try again. Vehicle is still parked.");   } catch (VehicleNotFoundException e) {  System.out.println("Error: " + quote(e.getMessage()));  } catch (Exception e) {  System.out.println("An unexpected error occurred: " + quote(e.getMessage()));  }  }   /\*\*  \* Prompts the user to select a payment method.  \*  \* @param scanner The Scanner for user input  \* @return The selected PaymentMethod, or null if invalid  \*/  private static PaymentMethod selectPaymentMethod(Scanner scanner) {  System.out.println("\nPayment method:");  System.out.println("1. Credit Card");  System.out.println("2. UPI");   int paymentMethod = getIntInput(scanner, "Choose payment method: ", 1, 2);  return switch (paymentMethod) {  case 1 -> {  String cardNumber = getStringInput(scanner, "Enter card number: ");  String expiryDate = getStringInput(scanner, "Enter expiry date (MM/YY): ");  String cvv = getStringInput(scanner, "Enter CVV: ");  String cardHolderName = getStringInput(scanner, "Enter card holder name: ");  yield new CreditCardPayment(cardNumber, expiryDate, cvv, cardHolderName);  }  case 2 -> {  String upiId = getStringInput(scanner, "Enter UPI ID: ");  yield new UPIPayment(upiId);  }  default -> {  System.out.println("Invalid payment method.");  yield null;  }  };  }   /\*\*  \* Displays the availability of parking spots by type.  \*  \* @param parkingLot The ParkingLot instance  \*/  private static void checkAvailability(ParkingLot parkingLot) {  System.out.println("\n-- Parking Availability --");  System.out.println("Available spots by type:");  System.out.println("Car spots: " + parkingLot.getAvailableSpotsByType(ParkingSpot.TYPE\_CAR));  System.out.println("Bike spots: " + parkingLot.getAvailableSpotsByType(ParkingSpot.TYPE\_BIKE));  System.out.println("Truck spots: " + parkingLot.getAvailableSpotsByType(ParkingSpot.TYPE\_TRUCK));  System.out.println("Electric spots: " + parkingLot.getAvailableSpotsByType(ParkingSpot.TYPE\_ELECTRIC));  System.out.println("Handicapped spots: " + parkingLot.getAvailableSpotsByType(ParkingSpot.TYPE\_HANDICAPPED));  }   /\*\*  \* Manages subscription-related operations.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void manageSubscriptions(ParkingLot parkingLot, Scanner scanner) {  System.out.println("\n-- Subscription Management --");  System.out.println("1. Register new subscription");  System.out.println("2. Check subscription status");  System.out.println("3. Renew subscription");  System.out.println("4. Cancel subscription");  System.out.println("5. View all active subscriptions");   int option = getIntInput(scanner, "Choose an option: ", 1, 5);  switch (option) {  case 1 -> registerNewSubscription(parkingLot, scanner);  case 2 -> checkSubscriptionStatus(parkingLot, scanner);  case 3 -> renewSubscription(parkingLot, scanner);  case 4 -> cancelSubscription(parkingLot, scanner);  case 5 -> viewActiveSubscriptions(parkingLot);  default -> System.out.println("Invalid option.");  }  }   /\*\*  \* Registers a new subscription for a vehicle.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void registerNewSubscription(ParkingLot parkingLot, Scanner scanner) {  System.out.println("\n-- Register New Subscription --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   if (parkingLot.getSubscriptionManager().hasActiveSubscription(licensePlate)) {  System.out.println("This vehicle already has an active subscription.");  return;  }   Vehicle vehicle = createVehicle(scanner, licensePlate);  if (vehicle == null) return;   String spotType = selectSpotType(vehicle);  String subType = selectSubscriptionType(scanner);  double fee = parkingLot.getSubscriptionManager().calculateSubscriptionFee(vehicle, subType, spotType);   System.out.println("\nSubscription fee: Rs. " + fee);  PaymentMethod method = selectPaymentMethod(scanner);  if (method == null) return;   if (confirmAction(scanner, "Confirm subscription? (y/n): ")) {  System.out.println("Processing payment...");  boolean paymentSuccess = parkingLot.getSubscriptionManager().processSubscriptionPayment(  vehicle, subType, spotType, method);   if (paymentSuccess) {  Subscription subscription = parkingLot.getSubscriptionManager().getSubscription(licensePlate);  System.out.println("Subscription registered successfully!");  System.out.println("Subscription ID: " + subscription.getSubscriptionId());  System.out.println("Valid until: " + subscription.getEndDate());  saveData(parkingLot);  } else {  System.out.println("Payment failed. Subscription not registered.");  }  } else {  System.out.println("Subscription registration cancelled.");  }  }   /\*\*  \* Selects the appropriate parking spot type for a vehicle.  \*  \* @param vehicle The vehicle to assign a spot type  \* @return The selected parking spot type  \*/  private static String selectSpotType(Vehicle vehicle) {  if (vehicle instanceof Car) {  return ParkingSpot.TYPE\_CAR;  } else if (vehicle instanceof Bike) {  return ParkingSpot.TYPE\_BIKE;  } else if (vehicle instanceof Truck) {  return ParkingSpot.TYPE\_TRUCK;  } else if (vehicle instanceof ElectricVehicle) {  return ParkingSpot.TYPE\_ELECTRIC;  } else {  return ParkingSpot.TYPE\_CAR; // Default fallback  }  }   /\*\*  \* Prompts the user to select a subscription type.  \*  \* @param scanner The Scanner for user input  \* @return The selected subscription type  \*/  private static String selectSubscriptionType(Scanner scanner) {  System.out.println("\nSubscription type:");  System.out.println("1. Monthly");  System.out.println("2. Quarterly (3 months - 10% discount)");  System.out.println("3. Semi-Annual (6 months - 15% discount)");  System.out.println("4. Annual (12 months - 20% discount)");   int subTypeChoice = getIntInput(scanner, "Choose subscription type: ", 1, 4);  return switch (subTypeChoice) {  case 1 -> Subscription.TYPE\_MONTHLY;  case 2 -> Subscription.TYPE\_QUARTERLY;  case 3 -> Subscription.TYPE\_SEMI\_ANNUAL;  case 4 -> Subscription.TYPE\_ANNUAL;  default -> Subscription.TYPE\_MONTHLY;  };  }   /\*\*  \* Checks the status of a subscription for a vehicle.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void checkSubscriptionStatus(ParkingLot parkingLot, Scanner scanner) {  System.out.println("\n-- Check Subscription Status --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   Subscription subscription = parkingLot.getSubscriptionManager().getSubscription(licensePlate);  if (subscription == null) {  System.out.println("No subscription found for this vehicle.");  } else {  System.out.println("\nSubscription details:");  System.out.println("ID: " + subscription.getSubscriptionId());  System.out.println("Type: " + subscription.getType());  System.out.println("Start date: " + subscription.getStartDate());  System.out.println("End date: " + subscription.getEndDate());  System.out.println("Status: " + (subscription.isActive() ? "Active" : "Inactive"));  System.out.println("Parking spot type: " + subscription.getSpotType());  }  }   /\*\*  \* Handles renewing an existing subscription.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void renewSubscription(ParkingLot parkingLot, Scanner scanner) {  System.out.println("\n-- Renew Subscription --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   Subscription subscription = parkingLot.getSubscriptionManager().getSubscription(licensePlate);  if (subscription == null) {  System.out.println("No subscription found for this vehicle.");  return;  }   System.out.println("\nCurrent subscription ends on: " + subscription.getEndDate());  System.out.println("\nRenewal period:");  System.out.println("1. 1 month");  System.out.println("2. 3 months");  System.out.println("3. 6 months");  System.out.println("4. 12 months");   int periodChoice = getIntInput(scanner, "Choose renewal period: ", 1, 4);  var renewal = selectRenewalPeriod(periodChoice);  int months = renewal.months();  String subType = renewal.subType();   double fee = parkingLot.getSubscriptionManager().calculateSubscriptionFee(  subscription.getVehicle(), subType, subscription.getSpotType());  System.out.println("\nRenewal fee: Rs. " + fee);   PaymentMethod method = selectPaymentMethod(scanner);  if (method == null) return;   if (confirmAction(scanner, "Confirm renewal? (y/n): ")) {  System.out.println("Processing payment...");  boolean paymentSuccess = parkingLot.getSubscriptionManager().processSubscriptionPayment(  subscription.getVehicle(), subType, subscription.getSpotType(), method);   if (paymentSuccess) {  parkingLot.getSubscriptionManager().renewSubscription(licensePlate, months);  System.out.println("Subscription renewed successfully!");  System.out.println("New end date: " + subscription.getEndDate());  saveData(parkingLot);  } else {  System.out.println("Payment failed. Subscription not renewed.");  }  } else {  System.out.println("Renewal cancelled.");  }  }   /\*\*  \* Represents a renewal period with the number of months and subscription type.  \*  \* @param months The number of months for renewal  \* @param subType The subscription type for renewal  \*/  private record RenewalPeriod(int months, String subType) {  }   /\*\*  \* Selects the renewal period based on user choice.  \*  \* @param periodChoice The user's choice for renewal period  \* @return The selected RenewalPeriod  \*/  private static RenewalPeriod selectRenewalPeriod(int periodChoice) {  return switch (periodChoice) {  case 1 -> new RenewalPeriod(1, Subscription.TYPE\_MONTHLY);  case 2 -> new RenewalPeriod(3, Subscription.TYPE\_QUARTERLY);  case 3 -> new RenewalPeriod(6, Subscription.TYPE\_SEMI\_ANNUAL);  case 4 -> new RenewalPeriod(12, Subscription.TYPE\_ANNUAL);  default -> new RenewalPeriod(1, Subscription.TYPE\_MONTHLY);  };  }   /\*\*  \* Handles canceling an existing subscription.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \*/  private static void cancelSubscription(ParkingLot parkingLot, Scanner scanner) {  System.out.println("\n-- Cancel Subscription --");  String licensePlate = getStringInput(scanner, "Enter license plate: ").toUpperCase();   Subscription subscription = parkingLot.getSubscriptionManager().getSubscription(licensePlate);  if (subscription == null) {  System.out.println("No subscription found for this vehicle.");  return;  }  if (!subscription.isActive()) {  System.out.println("This subscription is already inactive.");  return;  }   System.out.println("\nSubscription details:");  System.out.println("ID: " + subscription.getSubscriptionId());  System.out.println("Type: " + subscription.getType());  System.out.println("End date: " + subscription.getEndDate());   if (confirmAction(scanner, "Are you sure you want to cancel this subscription? (y/n): ")) {  parkingLot.getSubscriptionManager().cancelSubscription(licensePlate);  System.out.println("Subscription cancelled successfully.");  saveData(parkingLot);  } else {  System.out.println("Cancellation aborted.");  }  }   /\*\*  \* Displays all active subscriptions in the parking lot.  \*  \* @param parkingLot The ParkingLot instance  \*/  private static void viewActiveSubscriptions(ParkingLot parkingLot) {  System.out.println("\n-- Active Subscriptions --");  List<Subscription> activeSubscriptions = parkingLot.getSubscriptionManager().getAllActiveSubscriptions();   if (activeSubscriptions.isEmpty()) {  System.out.println("No active subscriptions found.");  return;  }   System.out.println("------------------------------------------------------------------------------------------");  System.out.printf("%-10s %-15s %-12s %-12s %-15s\n",  "ID", "License Plate", "Type", "Spot Type", "End Date");  System.out.println("------------------------------------------------------------------------------------------");   SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");  for (Subscription sub : activeSubscriptions) {  System.out.printf("%-10s %-15s %-12s %-12s %-15s\n",  sub.getSubscriptionId().substring(0, 8),  sub.getVehicle().getLicensePlate(),  sub.getType(),  sub.getSpotType(),  dateFormat.format(sub.getEndDate()));  }  System.out.println("------------------------------------------------------------------------------------------");  }   /\*\*  \* Displays all payment transactions in the parking lot.  \*  \* @param parkingLot The ParkingLot instance  \* @param dateFormat The date format for displaying times  \*/  private static void viewAllTransactions(ParkingLot parkingLot, SimpleDateFormat dateFormat) {  System.out.println("\n-- All Transactions --");  List<Payment> transactions = parkingLot.getAllTransactions();   if (transactions.isEmpty()) {  System.out.println("No transactions found.");  return;  }   System.out.println("------------------------------------------------------------------------------------------");  System.out.printf("%-15s %-12s %-10s %-20s %-12s\n",  "License Plate", "Amount", "Status", "Payment Time", "Method");  System.out.println("------------------------------------------------------------------------------------------");   for (Payment payment : transactions) {  String method = payment.getMethod() != null ? payment.getMethod().getPaymentDetails() : "N/A";  System.out.printf("%-15s Rs. %-9.2f %-10s %-20s %-12s\n",  payment.getVehicle().getLicensePlate(),  payment.getAmount(),  payment.getStatus(),  dateFormat.format(payment.getPaymentTime()),  method);  }  System.out.println("------------------------------------------------------------------------------------------");  }   /\*\*  \* Handles viewing security logs, either all logs or logs for a specific date.  \*  \* @param parkingLot The ParkingLot instance  \* @param scanner The Scanner for user input  \* @param dateFormat The date format for displaying times  \*/  private static void viewSecurityLogs(ParkingLot parkingLot, Scanner scanner, SimpleDateFormat dateFormat) {  System.out.println("\n-- Security Logs --");  System.out.println("1. View all logs");  System.out.println("2. View logs by date");   int option = getIntInput(scanner, "Choose an option: ", 1, 2);  switch (option) {  case 1 -> displayAllLogs(parkingLot, dateFormat);  case 2 -> {  try {  SimpleDateFormat inputFormat = new SimpleDateFormat("dd-MM-yyyy");  Date date = getDateInput(scanner, "Enter date (dd-MM-yyyy): ", inputFormat);  displayLogsByDate(parkingLot, date, dateFormat);  } catch (Exception e) {  System.out.println("Error: " + quote(e.getMessage()));  }  }  default -> System.out.println("Invalid option.");  }  }   /\*\*  \* Displays all security logs in the parking lot.  \*  \* @param parkingLot The ParkingLot instance  \* @param dateFormat The date format for displaying times  \*/  private static void displayAllLogs(ParkingLot parkingLot, SimpleDateFormat dateFormat) {  Map<String, List<EntryExitLog>> logs = parkingLot.getSecurityLogs();  if (logs.isEmpty()) {  System.out.println("No security logs found.");  return;  }   System.out.println("\n-- All Security Logs --");  System.out.println("------------------------------------------------------------------------------------------");  System.out.printf("%-15s %-12s %-20s %-20s %-12s\n",  "License Plate", "Spot ID", "Entry Time", "Exit Time", "Vehicle Type");  System.out.println("------------------------------------------------------------------------------------------");   for (Map.Entry<String, List<EntryExitLog>> entry : logs.entrySet()) {  String licensePlate = entry.getKey();  for (EntryExitLog log : entry.getValue()) {  String exitTime = log.getExitTime() != null ? dateFormat.format(log.getExitTime()) : "Still Parked";  System.out.printf("%-15s %-12s %-20s %-20s %-12s\n",  licensePlate,  log.getSpot().getSpotId(),  dateFormat.format(log.getEntryTime()),  exitTime,  log.getVehicle().getType());  }  }  System.out.println("------------------------------------------------------------------------------------------");  }   /\*\*  \* Displays security logs for a specific date.  \*  \* @param parkingLot The ParkingLot instance  \* @param date The date to filter logs  \* @param dateFormat The date format for displaying times  \*/  private static void displayLogsByDate(ParkingLot parkingLot, Date date, SimpleDateFormat dateFormat) {  List<EntryExitLog> logs = parkingLot.getSecurityLogsByDate(date);  if (logs.isEmpty()) {  System.out.println("No logs found for the specified date.");  return;  }   System.out.println("\nSecurity Logs for " + new SimpleDateFormat("dd-MM-yyyy").format(date) + ":");  System.out.println("------------------------------------------------------------");  System.out.printf("%-15s %-12s %-20s %-20s\n", "License Plate", "Spot ID", "Entry Time", "Exit Time");  System.out.println("------------------------------------------------------------");   for (EntryExitLog log : logs) {  String exitTime = log.getExitTime() != null ? dateFormat.format(log.getExitTime()) : "Still Parked";  System.out.printf("%-15s %-12s %-20s %-20s\n",  log.getVehicle().getLicensePlate(),  log.getSpot().getSpotId(),  dateFormat.format(log.getEntryTime()),  exitTime);  }  System.out.println("------------------------------------------------------------");  }   /\*\*  \* Saves the parking lot data to a file.  \*  \* @param parkingLot The ParkingLot instance  \*/  private static void saveData(ParkingLot parkingLot) {  try {  parkingLot.saveDataToFile(DATA\_FILE);  System.out.println("All data saved.");  } catch (IOException e) {  System.out.println("Error saving data: " + quote(e.getMessage()));  }  }   /\*\*  \* Prompts the user to confirm an action.  \*  \* @param scanner The Scanner for user input  \* @param prompt The confirmation prompt  \* @return True if the user confirms, false otherwise  \*/  private static boolean confirmAction(Scanner scanner, String prompt) {  return getStringInput(scanner, prompt).equalsIgnoreCase("y");  }   /\*\*  \* Retrieves an integer input from the user within a specified range.  \*  \* @param scanner The Scanner for user input  \* @param prompt The input prompt  \* @param min The minimum allowed value  \* @param max The maximum allowed value  \* @return The validated integer input  \*/  private static int getIntInput(Scanner scanner, String prompt, int min, int max) {  while (true) {  System.out.print(prompt);  try {  int value = scanner.nextInt();  scanner.nextLine();  if (value >= min && value <= max) return value;  System.out.println("Please enter a number between " + min + " and " + max);  } catch (InputMismatchException e) {  System.out.println("Please enter a valid number");  scanner.nextLine();  }  }  }   /\*\*  \* Retrieves a double input from the user within a specified range.  \*  \* @param scanner The Scanner for user input  \* @param prompt The input prompt  \* @param min The minimum allowed value  \* @param max The maximum allowed value  \* @return The validated double input  \*/  private static double getDoubleInput(Scanner scanner, String prompt, double min, double max) {  while (true) {  System.out.print(prompt);  try {  double value = scanner.nextDouble();  scanner.nextLine();  if (value >= min && value <= max) return value;  System.out.println("Please enter a number between " + min + " and " + max);  } catch (InputMismatchException e) {  System.out.println("Please enter a valid number");  scanner.nextLine();  }  }  }   /\*\*  \* Retrieves a non-empty string input from the user.  \*  \* @param scanner The Scanner for user input  \* @param prompt The input prompt  \* @return The validated string input  \*/  private static String getStringInput(Scanner scanner, String prompt) {  System.out.print(prompt);  String input = scanner.nextLine().trim();  while (input.isEmpty()) {  System.out.print("Input cannot be empty. " + prompt);  input = scanner.nextLine().trim();  }  return input;  }   /\*\*  \* Retrieves a date input from the user in the specified format.  \*  \* @param scanner The Scanner for user input  \* @param prompt The input prompt  \* @param dateFormat The date format for parsing  \* @return The parsed Date object  \*/  private static Date getDateInput(Scanner scanner, String prompt, SimpleDateFormat dateFormat) {  while (true) {  System.out.print(prompt);  try {  return dateFormat.parse(scanner.nextLine().trim());  } catch (ParseException e) {  System.out.println("Invalid date format. Please use " + dateFormat.toPattern());  }  }  }   /\*\*  \* Wraps a string in quotation marks for error messages.  \*  \* @param str The string to quote  \* @return The quoted string  \*/  private static String quote(String str) {  return "\"" + str + "\"";  } } |
| --- |

## Functionalities

The system supports the following functionalities:

* Parking Management
* Reservations
* Subscription
* Payment Gateway
* Security Logs
* Data persistence