# Code for the Parking Lot

Let's write the code for the classes that we have designed, in different languages in this lesson.

**We'll cover the following**



We've gone over the different aspects of the parking lot system and observed the attributes attached to the problem using various UML diagrams. Let's explore the more practical side of things, where we will work on implementing the parking lot system using multiple languages. This is usually the last step in an object-oriented design interview process.

We have chosen the following languages to write the skeleton code of the different classes present in the parking lot system:

- Java
- C#
- Python
- C++
- JavaScript

## Parking lot classes

In this section, we will provide the skeleton code of the classes designed in the class diagram lesson.

> **Note**: For simplicity, we aren't defining getter and setter functions. The reader can assume that all class attributes are private and accessed through their respective public getter methods and

> modified only through their public method functions.

## Enumerations and custom data type

First of all, we will define all the enumerations required in the parking lot. According to the class diagram, there are two enumerations used in the system i.e., `PaymentStatus` and `AccountStatus`. The code to implement these enumerations and custom data types is as follows:

> **Note**: JavaScript does not support enumerations, so we will be using the `Object.freeze()` method as an alternative that freezes an object and prevents further modifications.

```
// Enumeration
enum PaymentStatus {
  COMPLETED,
  FAILED,
  PENDING,
  UNPAID,
  REFUNDED
}

enum AccountStatus {
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  NONE
}

// Custom Person data type class
class Person {
  private:
    string name;
    string address;
    string phone;
    string email;
};

// Custom Address data type class
class Address {
  private:
    int zipCode;
    string address;
```

Definition for the constants

## Parking spots

The first section of the parking lot system that we will work on is the `ParkingSpot` class, which will act as a base class for four different types of parking spots: handicapped, compact, large, and motorcycle. This will have an instance of the `Vehicle` class. The definition of the `ParkingSpot` class and the classes being derived from it are given below:

```
// ParkingSpot is an abstract class
class ParkingSpot {
```

```
  private:
    int id;
    bool isFree;
    Vehicle vehicle;

  public:
    bool isFree();
    virtual bool assignVehicle(Vehicle vehicle) = 0;
    bool removeVehicle();
};

class Handicapped : public ParkingSpot {
  public:
    bool assignVehicle(Vehicle vehicle) {
        // definition
    }
};

class Compact : public ParkingSpot {
  public:
    bool assignVehicle(Vehicle vehicle) {
        // definition
    }
};

class Large : public ParkingSpot {
  public:
    bool assignVehicle(Vehicle vehicle) {
```

ParkingSpot and its derived classes

## Vehicle

`Vehicle` will be another abstract class, which serves as a parent for four different types of vehicles: car, truck, van, and motor cycle. The definition of the `Vehicle` and its child classes are given below:

```
// Vehicle is an abstract class
class Vehicle {
  private:
    string licenseNo;

  public:
    void virtual assignTicket(ParkingTicket ticket) = 0;
};

class Car : public Vehicle {
  public:
    void assignTicket(ParkingTicket ticket) {
        // definition
    }
};

class Van : public Vehicle {
  public:
    void assignTicket(ParkingTicket ticket) {
        // definition
    }
};

class Truck : public Vehicle {
  public:
    void assignTicket(ParkingTicket ticket) {
```

```
    }
};

class MotorCycle : public Vehicle {
```

Vehicle and its child classes

## Account

The `Account` class will be an abstract class, which will have the actors, `Admin` and `ParkingAttendant`, as child classes. The definition of these classes is given below:

```
class Account {
  // Data members
  private:
    string userName;
    string password;
    Person person; // Refers to an instance of the Person class
    AccountStatus status; // Refers to the AccountStatus enum

  public:
    virtual bool resetPassword() = 0;
}

class Admin : public Account {
  public:
    // spot here refers to an instance of the ParkingSpot class
    bool addParkingSpot(ParkingSpot spot);
    // displayBoard here refers to an instance of the DisplayBoard class
    bool addDisplayBoard(DisplayBoard displayBoard);
    // entrance here refers to an instance of the Entrance class
    bool addEntrance(Entrance entrance);
    // exit here refers to an instance of the Exit class
    bool addExit(Exit exit;

    // Will implement the functionality in this class
    bool resetPassword() {
      // definition
    }
}

class ParkingAttendant : public Account {
  public:
```

Account and its child classes

## Display board and parking rate

This section contains the `DisplayBoard` and `ParkingRate` classes that only have the composition class with the `ParkingLot` class. This relationship is highlighted in the `ParkingLot` class. The definition of these classes is given below:

```
class DisplayBoard {
// Data members
  private:
    int id;
    vector<Handicapped> handicappedSpot;
    vector<Compact> compactSpot;
    vector<Large> largeSpot;
    vector<MotorCycle> motorCycleSpot;

```

```
  // Member functions
  public:
    void showFreeSlot();
};

class ParkingRate {
  // Data members
  private:
    double hours;
    double rate;

// Member function
  public:
    void calculate();
};
```

The DisplayBoard and ParkingRate classes

## Entrance and exit

This section contains the `Entrance` and `Exit` classes, both of which are associated with the `ParkingTicket` class. The definition of the `Entrance` and `Exit` classes is given below:

```
class Entrance {
  // Data members
  private:
    int id;

  // Member function
  public:
    ParkingTicket getTicket();
};

class Exit {
  // Data members
  private:
    int id;

  // Member function
  public:
    void validateTicket(ParkingTicket ticket);{
      // Perform validation logic for the parking ticket
      // Calculate parking charges, if necessary
      // Handle the exit process
    }
};
```

The Entrance and Exit classes

## Parking ticket

The definition of the `ParkingTicket` class can be found below. This contains instancesof the `Vehicle`, `Payment`, `Entrance` and `Exit` classes:

```
class ParkingTicket {
  private:
    int ticketNo;
    time_t timestamp;
    time_t exit;
    double amount;
    bool status;

    // Following are the instances of their respective classes
    Vehicle vehicle;
    Payment payment;
    Entrance entrance;
    Exit exitIns;
};
```

The ParkingTicket class

## Payment

The `Payment` class is another abstract class, with the `Cash` and `CreditCard` classes as its child. This takes the `PaymentStatus` enumeration and the `dateTime` data type to keep track of the `payment status` and time. The definition of this class is given below

```
// Payment is an abstract class
class Payment {
    private:
        double amount;
        PaymentStatus status;
        time_t timestamp;

    public virtual bool initiateTransaction() = 0;
};

class Cash : public Payment {
    public bool initiateTransaction() {
        // definition
    }
};

class CreditCard : public Payment {
    public bool initiateTransaction() {
        // definition
    }
};
```

Payment and its derived classes

## Parking lot

The final class of the parking lot system is the `ParkingLot` class which will be a Singleton class, meaning the entire system will only have one instance of this class. The definition of this class is given below:

```
class ParkingLot {
    private:
        int id;
        string name;
        string address;
        ParkingRate parkingRate;

        map<string, Entrance> entrance;
        map<string, Exit> exit;

        // Create a hashmap that identifies all currently generated tickets using their ticket number
        map<string, ParkingTicket> tickets;

        // The ParkingLot is a singleton class that ensures it will have only one active instance at a t
        // Both the Entrance and Exit classes use this class to create and close parking tickets
        static ParkingLot parkingLot = NULL;

        // Created a private constructor to add a restriction (due to Singleton)
        ParkingLot() {
            // Call the name, address, parking_rate elements of the customer in the parking lot from the
            // Create initial entrance and exit hashmaps respectively
        }

    // Created a static method to access the singleton instance of ParkingLot
    public:
        static ParkingLot getInstance() {
            if (parkingLot == NULL) {
                parkingLot = new ParkingLot();
            }
            return parkingLot;
```

The ParkingLot class

## Wrapping up

We've explored the complete design of a parking lot system in this chapter. We've looked at how a basic parking lot system can be visualized using various UML diagrams and designed using object-oriented principles and design patterns.

Back

Activity Diagram for th...

Next

Getting Ready: Elevat...

Mark as Completed