EECS2030F: LAB 2 Due: Sept. 30, 2025 - 11:59 pm

The purpose of this lab is to ensure that you practice

A) implementing a simple class with complex mathematical operations,

B) encapsulating your data,

C) and writing unit test to test your code.

1. Lab Policies

a) **Academic Integrity**: Submit your own work. Do not copy code from classmates or online sources. All violations will be reported as academic misconduct.

b) **Submission Format**: Submit only the file: **ParkingSpot.java**. Do not upload ZIP files or project folders.

c) **Work Environment**: Complete the methods and test your submission by using tester file**.** Your code should not contain a main method and Scanner Class objects and its methods. Ensure your code compiles without errors.

d) **Deadline:** Submit your **ParkingSpot.java** file to the eClass course page by **Tuesday, September 30 (11:59pm).** No late submissions are accepted. Email submissions are not accepted.

e) Don't change the name of the file as we will not be able to test your code with our tester**.** It needs to be **ParkingSpot.java** (no case change)

f) Your lab assignment is not graded during the weekly lab sessions scheduled. The lab sessions are meant to get your questions answered from TAs.

2. Downloading and importing the Starter Project

You have already done this step in previous labs. Follow the below listed steps to set up your working environment by:

a) Download the Eclipse Java project archive file from eClass: EECS2030_Lab2.zip

b) Launch Eclipse and browse to EECS2030-workspace (for instance or your own created workspace).

c) In Eclipse:

- Choose File->Import
- Under General, choose Existing Projects into workspace
- Choose Select archive file. Browse your compressed zip folder and attach it.

- Make sure that the EECS2030_Lab2 box is checked under Projects and you don't have the same project already in the workspace. Then Finish.
- You should see two files, one is called **ParkingSpot.java** and one **Test_ ParkingSpot.java**.

3. Important Notes:

To practice testing, we have only provided a set of incomplete test cases. Please make sure to add enough test cases to the tester that thoroughly tests your code. Look at the tester code, **Test_ ParkingSpot.java**, and add more test cases to ensure comprehensive testing of your code. To do this, you can copy one of the methods, change the name of the method to avoid having a duplicate method name, and modify the body of the method to test your code with your selected input.

4. Javadoc generation

The Javadoc has been written for you. All you need to do is generate it as an HTML file to make navigation easier. To do this, click on the lab2 package, select Project -> Generate Javadoc. It will ask you for the location where you want to store the documentation. You can choose the default location or select your own folder. Enter the path, and then click on Finish. If you look at the location where you stored the documentation, you'll see a file called index.html. Clicking on this file will display the project documentation in your browser.
If you have any doubts on the Javadoc generation, please review lecture slides of week2.

5. Programming Tasks for this Lab

This lab focuses on creating a smart parking system with advanced mathematical operations. You will implement classes that represent parking spots with coordinate-based positioning and a parking garage that manages complex pricing algorithms and distance calculations.

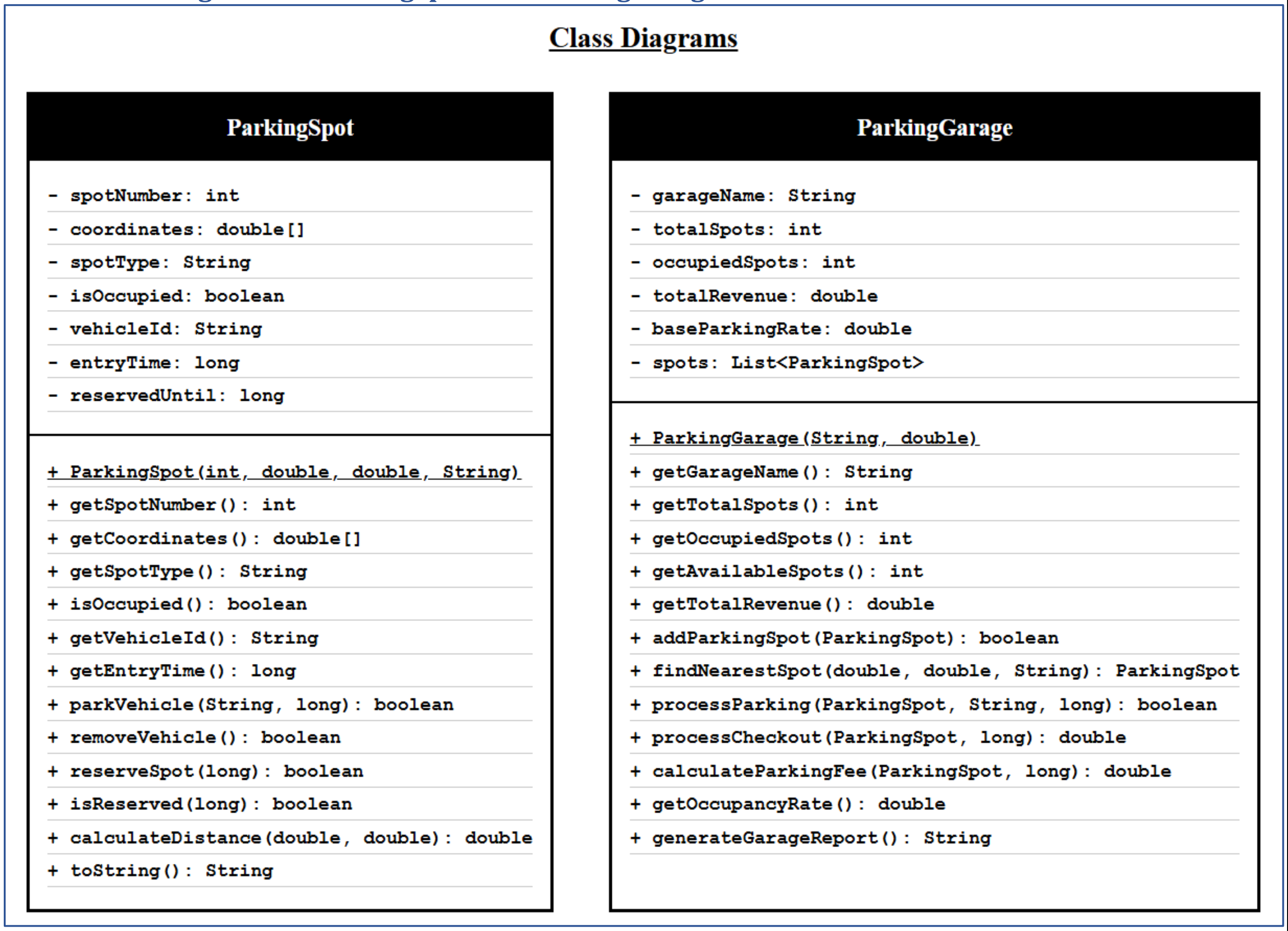
Imagine yourself as a software engineer working for a smart city initiative. Your task is to create an intelligent parking management system that can automatically find the nearest available parking spots, calculate dynamic pricing based on demand, and manage vehicle reservations. The system uses coordinate geometry and advanced algorithms to optimize parking efficiency. Let me guide you on how to solve the problem:

i. In the class ParkingSpot, define the components of a parking spot (i.e., spotNumber, coordinates, spotType, occupancy status, vehicle information, timing data). It is necessary to encapsulate your data since a client program, ParkingGarage, will use this class later.

ii. Implement the methods provided in the UML diagram below for the ParkingSpot class. These methods involve coordinate geometry, time calculations, and state management.

iii. Now you should implement the ParkingGarage class from scratch based on the provided UML diagram.

UML Diagram for ParkingSpot and ParkingGarage Class:



Class Diagrams

ParkingSpot

```
- spotNumber: int
- coordinates: double[]
- spotType: String
- isOccupied: boolean
- vehicleId: String
- entryTime: long
- reservedUntil: long

+ ParkingSpot(int, double, double, String)
+ getSpotNumber(): int
+ getCoordinates(): double[]
+ getSpotType(): String
+ isOccupied(): boolean
+ getVehicleId(): String
+ getEntryTime(): long
+ parkVehicle(String, long): boolean
+ removeVehicle(): boolean
+ reserveSpot(long): boolean
+ isReserved(long): boolean
+ calculateDistance(double, double): double
+ toString(): String
```

ParkingGarage

```
- garageName: String
- totalSpots: int
- occupiedSpots: int
- totalRevenue: double
- baseParkingRate: double
- spots: List<ParkingSpot>

+ ParkingGarage(String, double)
+ getGarageName(): String
+ getTotalSpots(): int
+ getOccupiedSpots(): int
+ getAvailableSpots(): int
+ getTotalRevenue(): double
+ addParkingSpot(ParkingSpot): boolean
+ findNearestSpot(double, double, String): ParkingSpot
+ processParking(ParkingSpot, String, long): boolean
+ processCheckout(ParkingSpot, long): double
+ calculateParkingFee(ParkingSpot, long): double
+ getOccupancyRate(): double
+ generateGarageReport(): String
```

The instance variables of this class represent the garage's name, spot statistics, revenue tracking, and base pricing rate. Now, you need to implement complex parking operations involving mathematical calculations:

Mathematical Operations Required:

i. **Distance Calculation (Euclidean Distance):** Implement calculateDistance(double, double) method in ParkingSpot using the formula: distance = $\sqrt{[(x_2-x_1)^2 + (y_2-y_1)^2]}$
where (x_1,y_1) are the spot coordinates and (x_2,y_2) are the target coordinates.

ii. **Nearest Spot Algorithm:** Implement findNearestSpot(double, double, String) method in ParkingGarage that:
- Filters available spots by type (Regular, Compact, Electric, Handicap)
- Calculates distance from given coordinates to each valid spot
- Returns the spot with minimum distance
- Handles tie-breaking by choosing the spot with smaller spot number

iii. **Dynamic Pricing Algorithm:** Implement calculateParkingFee(ParkingSpot, long) method using complex pricing formula: fee = baseRate × hours × typeMultiplier × demandFactor where:
- demandFactor = 1.0 + (occupancyRate × 0.5)
- occupancyRate = (occupiedSpots / totalSpots)
- hours = (checkoutTime - entryTime) / 3600000
- typeMultiplier = {

    Regular: 1.0

    Compact: 0.8

    Electric: 1.2

    Handicap: 0.5

  }

iv. **Time-based Reservation System:** Implement reservation functionality with time validation:
- reserveSpot(long): Reserve until specified timestamp
- isReserved(long): Check if spot is reserved at given time
- Handle overlapping reservations and expiry

v. **Occupancy Rate Calculation:** Implement getOccupancyRate() method:
occupancyRate = (occupiedSpots / totalSpots) × 100.0

Special Mathematical Requirements:

- **Coordinate System**: Use 2D Cartesian coordinates with double precision
- **Distance Precision**: Round distances to 2 decimal places using Math.round()
- **Time Calculations**: Use System.currentTimeMillis() for timestamp operations
- **Fee Calculations**: Round final fees to nearest cent (2 decimal places)
- **Percentage Calculations**: Handle division by zero for empty garages

6. Method Implementation Guidelines:

ParkingSpot Class:

- **Constructor**: Initialize with spot number, x-coordinate, y-coordinate, and spot type
- **parkVehicle()**: Mark spot as occupied with vehicle ID and entry time
- **removeVehicle()**: Clear spot and return true if successful
- **reserveSpot()**: Set reservation until specified time
- **calculateDistance()**: Use Euclidean distance formula with Math.sqrt()

ParkingGarage Class:

- **addParkingSpot()**: Register spot and update total count
- **findNearestSpot()**: Complex algorithm using distance calculations and filtering
- **processParking()**: Handle parking with validation and statistics update
- **processCheckout()**: Calculate fees, update revenue, and free the spot
- **calculateParkingFee()**: Apply dynamic pricing with multiple factors

Complex Algorithm:

Nearest Spot Algorithm Pseudocode: **findNearestSpot()**:

```
1. Initialize minDistance = Double.MAX_VALUE, nearestSpot = null
2. For each spot in garage:
  a. If spot.isOccupied() OR spot.isReserved() OR !spot.type.equals(requiredType)
     continue
        b. distance = spot.calculateDistance(targetX, targetY)
```

```
    c. If distance < minDistance OR (distance == minDistance AND spot.number <
       nearestSpot.number):
       nearestSpot = spot, minDistance = distance
3. Return nearestSpot
```

Dynamic Fee Calculation Pseudocode: calculateParkingFee()

```
1. duration = (checkoutTime - spot.entryTime) / 3600000.0
2. occupancyRate = occupiedSpots / totalSpots
3. demandFactor = 1.0 + (occupancyRate * 0.5)
4. typeMultiplier = getMultiplierForType(spot.type)
5. fee = baseRate * duration * typeMultiplier * demandFactor
6. Return Math.round(fee * 100.0) / 100.0  // Round to cent
```

Please note that, as usual, the signature of the method should not change if it is given.

7. Testing your code

A test case file has been provided to evaluate your methods. Your implementation should pass all the test cases if it correctly handles both normal and edge cases. **Do not modify or update the provided test cases in any way. You should create your own test cases too to thoroughly test your code.**

8. Submit

Submit only one file named " **ParkingSpot.java** " via eClass by clicking on the lab link.

9. Marking Schema

You are graded based on the correctness of your code. For each method, there will be comprehensive test cases to examine the correctness of the code. All test cases carry the same weight.

Please note that in all the labs, even if the test cases are provided, we will test your code with a different set of test cases to ensure that you have tested your code completely.

Note:

- The program that does not compile will get zero marks.
- Don't change the method headers. The test cases will fail if you change the method headers.

- Your submission should strictly have the name as **ParkingSpot.java**, otherwise it can not be used to test your assignment. It will have zero grades.
- If any method has iterative solution, the method would be graded as zero and its corresponding test cases will be considered as failed which will further reduce the grades for test cases.
- No resubmissions are allowed. Therefore, please make sure you submit the correct file within due date.

****