PROJECT 4
Parking Management
(optional) 100 points

Distribute on November 20, 2023
Due before December 4,, 2023 (Sunday) at 12:00 midnight

Learning Outcomes ((CLO) vs (SO) Mapping)

- Recognize the software and hardware components of a computer system (1)vs(6)
- Recognize and apply the software development phases (2)vs(6)
- Utilize Java syntax in fundamental programming algorithms (3)vs(1)
- Recognize and apply the various input and output devices in programming (4)vs(2)
- Recognize and apply the various control structures (5)vs(1)
- Recognize and apply the basic debugging strategies in programming(8)vs(2)
- Design and implement elementary multi-class solutions to programming problems (6)vs(2,6)
- Recognize the need for arrays in the solutions of programming problems, and manipulate data in one-dimensional arrays (7) vs(2,6)

Objectives

Completing this project, you will gain experience and/or continue practice with the following JAVA features:

- Multi-class design
- Class cooperation
- arrays
- Defining methods
- Using constructors with and without parameters
- Selection structures
- Loops
- Internal timing control

The problem and Requirements

This program is going to simulate the operations of a parking garage over an undetermined period of time.

- The parking bays in the garage are numbered (index).
- The capacity of the garage is a predetermined input number.
- The garage has an initial state of occupancy created by random selections, each bay is occupied with a probability of 0.5
- Cars arriving to park take the first available bay (that is, the bay that has the smallest index).
- The index of the bay occupied by the new car must be displayed by the program
- Cars leaving the garage are picked randomly from the set of parked cars

- When a car is leaving, the index of the vacated bay and the fee collected must be displayed by the program
- Parking operations are performed continually; it is decided randomly (with equal probability) if the next car is for parking or removal.
- Before parking operations start, the welcoming message of Figure 1 is displayed, followed by the display of the initial state of occupancy on the console (see Figure 2, C is for a car in bay E that marks an empty bay)
- Parking events are followed by displays as shown in Figures 3 and 4. Also, after every parking operation, the new state of occupancy like in Figure 2 must be displayed on the console
- The exact time of arrival for a parking car is to be registered, and upon leaving the garage the elapsed time the car spent in the garage is computed and a parking fee is collected accordingly
- The fee is based upon an hourly rate (an input) and fractional hours charged pro-rate.
- As mentioned above, the process is started with a garage set up randomly: each bay is occupied with a probability of **0.5**
- Before the program terminates the total value of the collected parking fees has to be reported
- The process is terminated in either of the following cases:
  - The next car arrives for parking, but the garage is full
  - The next car is supposed to leave, but the garage is empty
  - A series of ***N*** parking operations completed without interruption, where ***N*** is a predetermined limit (an input).

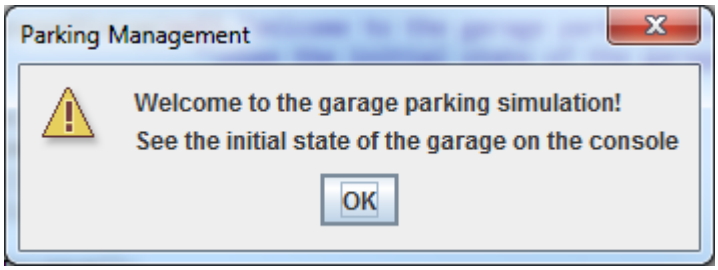

Figure 1

```
0    1    2    3    4    5    6    7    8    9    10   11
C    C    C    E    E    C    C    E    C    E    C    E
```

Figure 2

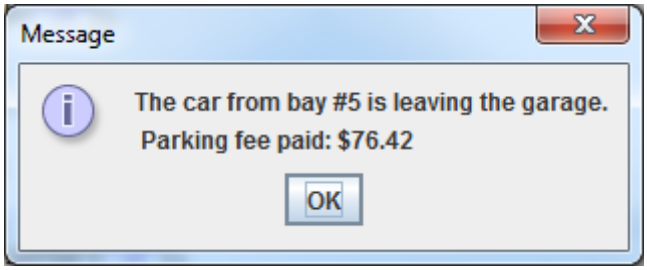

Figure 3

Figure 4

Analysis

Input

As described in the Problem section, the program needs

- the capacity of the garage (too large capacity is not practical for displaying the garage occupancy)
- the initial state of the garage
- a limit for the number of operations in the process (the too-large limit number is not practical for the potentially very long running time)
- a value for the parking fee per hour

Output

- messages showing the state of occupancy on the console
- messages reporting the current parking operations (JOptionPane windows)
- message reporting the total of parking fees collected (JOptionPane window, or console)

Tools and Formulas

- the method call

  ```
  long time = System.currentTimeMillis( );
  ```

  saves the return value of the method currentTimeMillis( ) in a variable of type **long** and named **time**. The method returns the actual time in **milliseconds** elapsed since 00:00 hour of January 1, 1970 (Greenwich mean-time). The value can be stored in variables of type **long**
- the parking time can be measured by the difference of two subsequent return values of currentTimeMillis( ), if it is called first at the arrival, and next when the car is leaving the garage
- one **second** of real-time (1000 millis) will represent a one-hour duration in the simulation model
- if **hourlyRate** is the dollar value charged for one-hour parking, the fee for a car leaving the garage will be

  double fee = hourlyRate*(departureTime – arrivalTime)/1000.00;

- the boolean value **Math.random( ) < 0.5** is **true** with probability **0.5**. As such it can be used as a coin toss to decide if the next garage operation is an arrival or a departure; in the same way, it can also be used to decide at the initial setup if a bay is occupied or vacant

Design and **Implementation**

The following responsibilities are naturally represented by different objects:

A car

- The **`Car`** class shall describe car objects. A car object must know the arrival time for occupying a parking bay

A garage

- The **`Garage`** class shall represent the garage
- the garage knows a Car type array to store Car objects
- can use accessor and mutator methods of the array
- can decide if a bay of a given index is empty
- can display the state of occupancy of the garage
- can park a car at the first available index (adding a car to the array), and notifies the caller (manager) of the index
- can remove a car from the array at a given index, and notifies the caller (manager) of the elapsed parking time measured in seconds
- can find out the bay index of a randomly selected car currently in the garage

A manager

- The **`Manager`** class represents the manager object, who runs the parking operations
- the manager knows the garage (the class has a Garage type reference variable)
- knows the number of cars currently in the garage
- knows the hourly rate (a named constant)
- knows the running total of collected fees
- aggregates a Random object
- can ask the garage to park a car, and manages the related messages
- can generate a random choice of a parked car to remove from the garage and can ask the garage to find and remove the selected car; computes and accumulates the parking fee of the leaving car
- can run the parking operations as many times as allowed by the limit

The **`Application`** class has the main method.

- The main method assigns values to capacity and limit
- declares and instantiates a Garage object to the capacity
- Adds a Car object to the array with 0.5 probability at each bay
- Instantiates a manager and the manager starts the process

The details of the above-described classes are shown in the UML diagrams below. The diagrams have attached columns for comments and method functionalities.

Car	
-timeIn: long	the field stores the arrival time; the value assigned in the constructor
+getTime():long	accessor associated with the field
+Car()	no-arg constructor; calls **currentTimeMillis( )** to initialize the field **timeIn**

Garage	
-cars: Car[]	an array to store Car objects; the array is instantiated in the constructor
+getCars( ): Car[]	accessor to the field
+setCars(auto: Car, index:int):void	mutator method; adds auto to the array at the index
+isEmpty(k: int): boolean	returns the boolean value **cars[k] = = null**
+displayState( ):void	first the method prints all the indices of the **cars** array to the console in a single line; second, in the next line and below each index prints a letter E (empty) for a **null**, and a letter C (car) for a non-**null** array entry
+park(auto: Car):int	calls isEmpty( ) for each bay and counts the number of non-empty bays; if the garage is full (the counting is car.length), returns **-1**; the first time an empty bay is found at an **index**, adds (assigns) the parameter **auto** to the array at index; returns the **index**
+remove(index: int): double	computes the elapsed parking time of the element **cars[index]**; assigns **null** to the bay of **index;** returns the elapsed time
+findBayOfCar(carNumber:int): int	Finds and returns the index of the bay, where a car with a randomly selected serial number is parked (see Hints)
+Garage(capacity: int)	non-default constructor instantiates the **cars** array to length **capacity**

Manager	
- garage: Garage	a Garage type reference variable
-FEE_PER_HOUR: double	named constant for hourly parking rate; assign 1.50
-feeTotal: double	stores the running total of collected fees
-manyCars	stores the number of cars currently parked in the garage
+parkACar():void	calls the park method of the Garage class, the parameter is new Car( ), stores the return value in the local variable **index**; if the index is not -1, Figure 4 is displayed, manyCars is updated and the displayState( ) of Garage is called; if index = -1, Figure 5 and Figure 7 are displayed, and the program terminates
+chooseACarToLeave():void	**if** the garage is empty (manyCars == 0), Figures 6 and 8 are displayed and the program terminates; **else** a parked car to leave is selected randomly; for instance, if manyCars is 7, use rd.nextInt to choose one of the numbers 1,2,3,4,5,6,7 Suppose the result is 4, then a call **findBayOfCar(4)** returns the **index**, where the fourth car is stored in the array. Next, **remove(index)** is called the method that receives the elapsed time; the method computes the fee, updates feeTotal, and displays the message in Figure 3; displayState is called and manyCars is updated
+processParking(limit:int):void	displays the welcoming message Figure1; calls displayState( ); runs a for loop to limit, at each iteration either the parkACar( ) or the chooseACarToLeave( ) method is called, each selected with probability 0.5 both messages of Figures 7 and 8 are displayed
+Manager(gar:Garage, many:int)	The constructor initializes the garage and

	manyCars

Application	
+main	stores garage capacity and the limit of iterations in local variables; for testing purposes both variables are assigned 15; declares and instantiates a Garage object with the given capacity; sets up a counter variable ; runs a for loop to visit all the parking bays; at each bay, a new Car object is added to the bay with probability 0.5 and the counter is updated for every added car; instantiates a Manager object with parameters garage and counter, the object calls processParking( ) with parameter limit

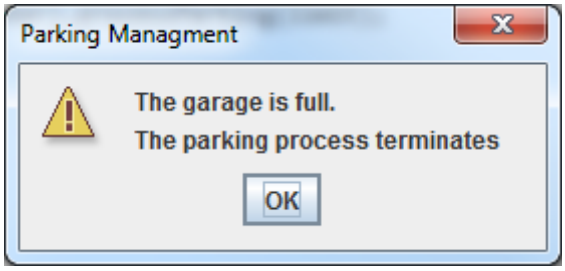

Figure 5

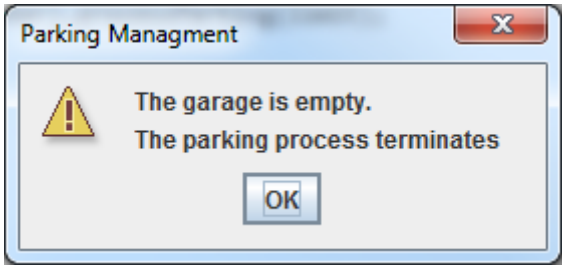

Figure 6

```
The total parking fee collected is $39.33
```

Figure 7

```
After 15 parking operations, the process is terminated.
```

Figure 8

Hints

1. Measuring time

The application of the **`currentTimeMillis( )`** method is explained in the Analysis section above

2. `findBayOfCar (int carNumber)`

This method in the Garage class is simple (just a few, say 6 lines of code), and there are many options to implement the logic, but it must be built and tested carefully. Try to do it first on your own without reading this hint.
The option recommended here might be the simplest, though it uses nested loops. In the method

- declare an int variable index (this is a counter), initialize index to -1 (!)
- run a for loop to the parameter carNumber;
- in the for loop
- update index
- run a while loop controlled by the boolean expression isEmpty(index);
- in the while loop
- update index
- after the for loop
- return index

3. Note that the Garage class does not maintain a manyCars variable to have the number of cars in the garage (the Manager class has it). Therefore the only way to know if the garage is full, a search for an empty bay must be run, and the search must be unsuccessful.

Testing

Codes with random choices should be tested with nonrandom input first. Make temporary printings including the random input values to check correctness. Methods should be tested independently of each other as much as possible.
You should use a temporary application class that calls and tests all the methods of a given class. Temporary console displays of partial results also help to see if the code works correctly. For testing purposes choose a small number for the garage capacity like 5 and a small number for limiting the iterations.
Also, for testing purposes, you may replace the random steps by applying temporary non-random choices to see if the methods of the manager class work properly.
Eliminate the temporary elements if testing is satisfactory.

Hand-In

You must have a comment block preceding each Java class header as usual:

```
/*
 * <your name>
 * CS 16000-01 – 02/03, Fall Semester 2023
 * Project 4: Parking Management
 * Description. <Summarize the purpose of the class here.>
 *
 */
```

What to Submit

- Submit your zipped project containing all the source codes at the designated link on Brightspace/Blackboard.

Evaluation

Your implementation must conform to the design above as follows:

- you must implement all classes with the responsibilities and collaboration as described above; use the class names given
- you cannot add any more data fields to either class
- you can add more methods to further divide the tasks (not recommended)
- you cannot eliminate the functionalities represented by the given methods
- you cannot unify tasks separated in the described classes and methods
- you may not change the identifiers given in the UML diagrams
- you may declare and use local variables within the methods at will

Documentation and Style (20 points)

Your program must conform to the Computer Science Department's Java Documentation and Style Requirements. Emphasis will be placed on having the required banner and internal comments, indentation, and overall professional appearance. Comments must be clearly written with correct grammar and spelling. **Note that the requirements of documentation are heavily represented in the score!**

Correctness (80 points)

1. 10 points for the correct class collaborations
2. 30 points for correct Garage methods (6 for each)
3. 30 points for the correct implementation of the Manager methods (10 for each)
4. 5 points for the correct main method
5. 5 points for the correct output as required

Total **100 points**