

Parking Lot v1.4.2

Nature of the Game

We want to understand how you think like a programmer, and the [level of craft](#) you bring to bear when building software.

Of course, the ideal would be a real-world problem, with real scale, but that isn't practical as it would take too much time. So instead we have a dead simple, high school level problem that we want you to solve as though it was a real-world problem.

Please note that not following the instructions below will result in an automated rejection. Taking longer than the time allocated will negatively affect our evaluation of your submission.

Rules of the Game

1. You have a week to implement a solution.
2. We are really, really interested in your object-oriented or functional design skills, so please craft the most beautiful code you can.
3. We're also interested in understanding how you make assumptions when building software. If a particular workflow or boundary condition is not defined in the problem statement below, what you do is your choice.
4. You have to solve the problem in any object-oriented or functional language without using any external libraries to the core language except for a testing library for TDD. Your solution must build+run on Linux. If you don't have access to a Linux dev machine, you can easily set one up using Docker.
5. Please use Git for version control. We expect you to send us a standard zip or tarball of your source code when you're done that includes Git metadata (the .git folder) in the tarball so we can look at your commit logs and understand how your solution evolved. Frequent commits are a huge plus.
6. Please do not check in binaries, class files, jars, libraries or output from the build process.
7. Please write comprehensive unit tests/specs. For object-oriented solutions, it's a huge plus if you test drive your code. Submitting your solution with only a functional suite will result in a rejection.
8. Please create your solution inside the parking_lot directory. Your codebase should have the same level of structure and organization as any mature open source project including coding conventions, directory structure and build approach (make, gradle etc) and a README.md with clear instructions.
9. For your submission to pass the automated tests, please update the Unix executable scripts bin/setup and bin/parking_lot in the bin directory of the project root. bin/setup should install dependencies and/or compile the code and then run

your unit test suite. `bin/parking_lot` runs the program itself. It takes an input file as an argument and prints the output on `STDOUT`. Please see the examples below. Please note that these files are Unix executable files and should run on Unix.

10. Please ensure that you follow the syntax and formatting of both the input and output samples. The zip file you have been sent includes the same automated functional test suite we use. This is to help you validate the correctness of your program. You can run it by invoking `bin/run_functional_tests`. IMPORTANT: To make the functional specs work correctly, some setup is needed. Instructions to set up the functional suite can be found under `functional_spec/README.md`.
11. Please do not make either your solution or this problem statement publicly available by, for example, using github or bitbucket or by posting this problem to a blog or forum.

Problem Statement

I own a parking lot that can hold up to 'n' cars at any given point in time. Each slot is given a number starting at 1 increasing with increasing distance from the entry point in steps of one. I want to create an automated ticketing system that allows my customers to use my parking lot without human intervention.

When a car enters my parking lot, I want to have a ticket issued to the driver. The ticket issuing process includes us documenting the registration number (number plate) and the colour of the car and allocating an available parking slot to the car before actually handing over a ticket to the driver (we assume that our customers are nice enough to always park in the slots allocated to them). The customer should be allocated a parking slot which is nearest to the entry. At the exit the customer returns the ticket which then marks the slot they were using as being available.

Due to government regulation, the system should provide me with the ability to find out:

- Registration numbers of all cars of a particular colour
- Slot number in which a car with a given registration number is parked.
- Slot numbers of all slots where a car of a particular colour is parked.

We interact with the system via a simple set of commands which produce a specific output. Please take a look at the example below, which includes all the commands you need to support - they're self explanatory. The system should allow input in two ways. Just to clarify, the same codebase should support both modes of input - we don't want two distinct submissions.

1. It should provide us with an interactive command prompt based shell where commands can be typed in
2. It should accept a filename as a parameter at the command prompt and read the commands from that file

Example: File

To install all dependencies, compile and run tests: \$ bin/setup

To run the code so it accepts input from a file: \$ bin/parking_lot file_inputs.txt

Input (contents of file):

```
create_parking_lot 6 park KA-01-HH-1234 White park KA-01-HH-9999 White park
KA-01-BB-0001 Black park KA-01-HH-7777 Red park KA-01-HH-2701 Blue park
KA-01-HH-3141 Black leave 4 status park KA-01-P-333 White park DL-12-AA-9999 White
registration_numbers_for_cars_with_colour White slot_numbers_for_cars_with_colour
White slot_number_for_registration_number KA-01-HH-3141
slot_number_for_registration_number MH-04-AY-1111
```

Output (to STDOUT):

```
Created a parking lot with 6 slots Allocated slot number: 1 Allocated slot number: 2
Allocated slot number: 3 Allocated slot number: 4 Allocated slot number: 5 Allocated slot
number: 6 Slot number 4 is free Slot No. Registration No Colour 1    KA-01-HH-1234
White 2    KA-01-HH-9999    White 3    KA-01-BB-0001    Black 5    KA-01-HH-2701    Blue 6
KA-01-HH-3141    Black Allocated slot number: 4 Sorry, parking lot is full KA-01-HH-1234,
KA-01-HH-9999, KA-01-P-333 1, 2, 4 6 Not found
```

Example: Interactive

To install all dependencies, compile and run tests: \$ bin/setup

To run the program and launch the shell: \$ bin/parking_lot

Assuming a parking lot with 6 slots, the following commands should be run in sequence by typing them in at a prompt and should produce output as described below the command.

Note that exit terminates the process and returns control to the shell.

```
$ create_parking_lot 6 Created a parking lot with 6 slots
```

```
$ park KA-01-HH-1234 White Allocated slot number: 1
```

```
$ park KA-01-HH-9999 White Allocated slot number: 2
```

```
$ park KA-01-BB-0001 Black Allocated slot number: 3
```

```
$ park KA-01-HH-7777 Red Allocated slot number: 4
```

```
$ park KA-01-HH-2701 Blue Allocated slot number: 5
```

```
$ park KA-01-HH-3141 Black Allocated slot number: 6
```

```
$ leave 4 Slot number 4 is free
```

```
$ status Slot No. Registration No Colour 1    KA-01-HH-1234    White 2    KA-01-HH-9999
White 3    KA-01-BB-0001    Black 5    KA-01-HH-2701    Blue 6    KA-01-HH-3141    Black
```

```
$ park KA-01-P-333 White Allocated slot number: 4
```

```
$ park DL-12-AA-9999 White Sorry, parking lot is full
$ registration_numbers_for_cars_with_colour White KA-01-HH-1234, KA-01-HH-9999,
KA-01-P-333
$ slot_numbers_for_cars_with_colour White 1, 2, 4
$ slot_number_for_registration_number KA-01-HH-3141 6
$ slot_number_for_registration_number MH-04-AY-1111 Not found
$ exit
```