Summary

This is documentation for a Parking Lot project, which has the following properties:

- N entries, M exits.
- C capacity.
- Can handle 2 kinds of vehicles, General and handicapped.

Description

The properties of the parking lot are number of entries, exits and capacity. These are configured in a file `conf.properties` which can be found in `ParkingLot/src/main/resources`.

- There are two threads that run under the hood, parking and unparking threads. As the name suggests, they handle parking and unparking queries respectively. The number of these threads are also configurable, also in the **config.properties** file, which makes the complete setup of the parking lot configurable.
- An interface, **IParkingLot** expects all implementors to implement two methods, **Park(Vehicle v)** and **unPark(Vehicle v)**. This makes the code extensible because regardless of the kind of parking lot being implemented, the methods will remain the same.
- A vehicle is of two types. This is also extensible because both **GeneralVehicle** and **HandicappedVehicle** extend the abstract class Vehicle.
- The core logic uses reEntrantLocks and condition variables. Condition variables are used because their functionality of **await** and **signal** mimic the wait, notify, notifyAll and synchronized aspects of multi-threaded programming. We have condition variables for number of entries, exits, and capacity. We also differentiate between general slots and handicapped slots. Thus, there can be a scenario where the parking lot hasn't used up it's complete capacity but is out of handicapped slots. This is handled by the respective condition variable.

Classes: Features

- ParkingLot --> implements the core functionality. Has reEntrantLocks, condition variables, and multiple threads handling all aspects. Implements the interface **IParkingLot**.
- ParkingLotClient --> client that generates requests based on the property in `config.properties` file. Can be modified to run more or less requests.
- Vehicle --> abstract class that should be extended to implement different vehicle types. Uses enum to differentiate different types of vehicles e.g General, Handicapped etc. Also uses `parkOrUnpark` to denote whether to park or unpark the vehicle.
- Custom exception classes (**VehicleNotParkedException**, **VehicleAlreadyParkedException**) to handle the not parked and already parked vehicle scenarios.
- ParkingThread and UnParkingThread --> these threads handle the parking and unparking functionality. They get one request every few milliseconds and go into wait if the parking lot class

is out of requests or if the method hits a condition variable (e.g all entries are full, the parking lot is full).
- PropertyValues --> this class is used to configure the features of the parking lot. e.g number of entries, number of exits, etc.

Evaluation & Results

- **ParkingLotClient.java** creates a request list comprising "numRequests" requests, which is also configurable in the `config.properties` file. The requests are randomly generated, where 0 stands for park and 1 stands for unpark.
- There are two JUnit test cases in com/parkinglot/test.
- If number of threads is large, then the chances of maxing out number of entries or exits is higher. One can simulate this by changing the config.properties file.

Dependencies

- junit-4.12
- hamcrest-core.jar

How to run the application

- You can build the JAR file by doing "mvn package" in the top-level directory.
- You can then run the class by doing java -jar target/parkinglotapplication-1.0-SNAPSHOT.jar.

Conclusion

This application implements a parking lot, making it configurable and extensible. It also has JUnit tests. It uses Object-Oriented concepts to generalize the parking lot.