Final Project Assignment

The objectives of this final project assignment are:

- To verify and confirm your proficiency with Java syntax.
- To verify your ability to properly document and test your own software system.
- To verify your knowledge of object-oriented principles, including inheritance, abstraction, information hiding, encapsulation, polymorphism, and abstraction.
- To verify your knowledge of the SOLID principles.
- To verify your knowledge and expertise of Java Collections Framework.
- To verify your knowledge of software design principles, and the MVC design architecture.
- To verify your knowledge of functional programming.
- To verify your ability to design and implement an intermediate-size software system.
- To confirm your ability to analyze and talk about your own code.

General Requirements

Please create a new Gradle project for this assignment in your individual course GitHub repo.
To submit your work, push it to GitHub and create a release.

Please choose one of the two possible options for your final project:

Option 1 – Rideshare Dispatch Simulator

Your ride-sharing business is going through a growth phase, and you have been tasked with developing a rideshare dispatch simulator, to get a better insight into your customers riding experience. Among other things, your simulator should provide insight into:

- How long do your customers have to wait for their ride?
- What is the average ride time?
- What would be the optimal number of drivers on the roads, to balance your business operating costs, with the customer convenience?

Specification:

You are going to design an **event-driven simulation** that models the customers requesting a ride through a ride-sharing app. The number of available drivers on the road will be entered by the user of your simulation.

When a new ride request arrives, your system should keep track of:

- customer's name (ID),
- customer's starting location
- customer's desired location
- the anticipated distance of the ride (the distance is calculated based on the origin and the destination, but this information has already been calculated for you),
- the time ride was requested, and
- the type of the ride, which can be one of the following:
    - Express pick-up (highest priority),
    - Standard pick-up,
    - Wait-and-save pick-up,
    - Environmentally conscious pick-up (lowest priority).

Based on the type of the ride, and the anticipated ride distance, the system should assign a priority to your customer. ***You will want to come up with the rules for how the priority is assigned.***

If there are currently drivers available to take on a new ride, you can assign the newly arrived ride request to a driver (you can assume that a driver cannot reject a ride for the purpose of this simulator).

However, if no drivers are available, then a customer needs to wait for one of the current rides to finish. Customers are being assigned to the freed-up drivers based on their priority. Customers with the same priority are served in order of arrival.

All the relevant data about customers, and rides, including:

1. Time of arrival,
2. The time ride was requested,
3. The length of a ride

should be simulated by your software system.

Events:

Your software system should support two different kinds of events:

- **Ride requested event** – this event happens when a new ride is being requested.
- **Ride finished event** – this event happens when one of the existing rides has reached its destination point.

To support these two types of events, we recommend using a **priority queue**. You are welcome to use any built-in data collection of your choice as a priority queue.

In your simulation, you will likely want to use one queue for ride requests, and one queue for currently active rides. We recommend organizing currently active rides by departure time (you can simplify the design, and assume that each ride has an average speed of 60mph. That would mean that the anticipated ride distance also provides information about the time duration of a ride).

Simulation:

Write a class `RideshareDispatchSimulator` that will run your simulation. The user input to your simulator will be the number of drivers.

For each of the simulation scenarios, output on the command line:

- What was the average wait time for a ride?
- What was the average number of rides a driver has handled?

Your system should work for at least the following simulation scenarios:

- 50 drivers and 25 rides
- 50 drivers and 100 rides
- 50 drivers and 250 rides

Option 2 – Rideshare System – Prospective Drivers Validator and Simulator

You are a part of a ride-sharing company, building a new **driver-side application**. In this phase of the development, you are developing a part of a system focusing on **prospective drivers**.

A prospective driver should be able to register with your application. In doing so, they should have a way to provide the following information:

- **Driver's name:** information about a prospective driver's first and last name
- **Driver's birthdate:** information about a prospective driver's day, month, and year of birth.
- **Driver license information:** information about a prospective driver's driver license, including information about the license unique number, a driver's name, a driver's address, a driver's birthdate, country and state of issuance, and issuance and expiration date.
- **Vehicle information:** information about make, model and year of the vehicle, as well as about the vehicle's official owner.
- **Vehicle's insurance information:** information about the vehicle's official owner, other people covered by insurance as drivers, and the expiration date of the insurance.
- **Driver's history: information** about all traffic violations committed by the prospective driver, including the date of violation, and a type of violation. A violation type can be a moving and non-moving violation. A moving violation can be one of the following:
    - Distracted driving
    - Reckless driving
    - Speeding
    - Driving under influence
    - Failure to respect traffic signs
    - Driving without a valid license and/or insurance

A non-moving violation can be one of the following:

- Parking violation
- Paperwork issues
- Problems with the vehicle

- **Vehicle history:** information about all crashes and all traffic violations committed with the prospective vehicle, including the date of violation, a type of violation, and the name of the offending driver. A crash can be one of the following:
    - A fender-bender
    - A crash without bodily injuries
    - A crash involving bodily injuries.

Registration Validator

Once a prospective driver has registered with the system, another part of the system, referred to as the **registration validator**, should check if the prospective driver is suitable to be added to the pool of registered and accepted drivers. Some things that the registration validator should check for:

- **Prospective driver's age:** if the prospective driver is underage (younger than 21), they should not be accepted as a driver.
- **Prospective driver's license information:** there are several things that the system should check on a driver's license:
    - **Name differences:** are there any differences between the name provided on the application and the name on the license? If yes, the prospective driver should not be accepted as a driver.
    - **Birthdate differences:** are there any differences between the birthdate provided on the application and the date on the license? If yes, the prospective driver should not be accepted as a driver.
    - **Country of issuance:** is the license issued in the US or Canada? If not, for now, the prospective driver should not be accepted as a driver.

    - **Date of issuance:** was the driver license issued less than six months ago? If yes, then the prospective driver should not be accepted as a driver.

    - **Expiration date:** has the driver license expired? If yes, then the prospective driver should not be accepted as a driver.

- **Vehicle information:**

    - Is the vehicle older than 15 years? If yes, then prospective driver should not be accepted as a driver.

- **Vehicle insurance information:**

    - Is the prospective driver the official owner of the vehicle?

    - If not, is the prospective driver listed as an insured driver? If not, then the prospective driver should not be accepted as a driver.

    - Has the insurance expired? If yes, then the prospective driver should not be accepted as a driver.

- **Driver's history:**

    - Does the prospective driver have any moving violations?

    - If yes, do those include reckless driving, speeding, DUI, or driving without a valid license/insurance? If yes, then prospective driver should not be accepted as a driver.

- **Vehicle history:**

    - Are there any crashes or moving violations committed with this vehicle in the last six months? If yes, then prospective driver should not be accepted as a driver.

Pool of Existing Accepted Drivers

Based upon the check performed by the registration validator, a prospective driver should either rejected as a driver, or they should be added to the pool of existing accepted drivers. The pool of existing accepted drivers is a data collection with the following additional requirements:

- **Pool uniqueness:** the pool of existing accepted drivers should be unique, in that it should not contain multiple instances of the same driver with the same vehicle.

- **Driver with multiple vehicles:** it should be possible for the same driver to register with multiple vehicles, but every driver can drive only one vehicle at the time (additional requirements provided in the next subsection).

- **Shared vehicle:** it should be possible for the same vehicle to be registered with different drivers, but the laws of physics should hold, such that multiple drivers cannot drive the same vehicle at the same time, in possibly different directions.

Additionally, the pool of existing accepted drivers can be queried, to present information about a specific driver and/or drivers. In this version of the project, querying can be done by calling method:

- `provideDriverInfo(String lastName)` – method displays on the console the full name of a driver whose last name is the same as the queried name. It then displays the list of all the vehicles registered with that driver. Finally, if the driver has any driving violations, those are displayed too.

If there are multiple drivers with the last name same as the queried name, information about all of them is displayed alphabetically by first name.

Finally, if there doesn't exist a driver with the queried last name, an appropriate message is displayed on the screen.

Example: let's assume there exist two accepted drivers with the last name Smith, Anne and Brandon. Anne has two vehicles registered, 2019 Blue Honda Accord, BAY01FG and 2018 Silver Toyota Prius, AZ034W1. Brandon has only one vehicle registered, 2017 Red Mazda 6, AAB1234. No drivers have any driving violations.

When method call `provideDriverInformation("Smith")` gets executed, we should expect to see the following on the screen:

Smith, Anne

2014 Blue Honda Accord, BAY01FG

2015 Silver Toyota Prius, AZ034W1

Smith, Brandon

2015 Red Mazda 6, AAB1234

Similarly, let's assume there exists only one driver with the last name Fuller – Patricia Fuller, who has only one vehicle registered – 2021 Blue Ford Explorer, ZZW-123. Patricia has two driving violations – distracted driving and parking violation.

When method call `provideDriverInformation("Fuller")` gets executed, we should expect to see the following on the screen:

Fuller, Patricia

2012 Blue Ford Explorer, ZZW-123

Driving violations:

Distracted driving

Parking violation

Finally, let's assume there doesn't exist an accepted driver with the last name Jefferson. When method call `provideDriverInformation("Jefferson")` gets executed, we should expect to see the following on the screen:

No registered driver found

Simulation:

Write a class `RideshareDriverValidator` that will run your simulation. The user input to your simulator will be a file containing information about prospective drivers. **You will want to define your own file format, and include one example file of prospective drivers with your submission.**

If the provided file is correctly read, and processed, the user should be able to interact with your program by searching for potential prospective drivers based on their last name.

Questions About Your Software System

Once you have designed, implemented, documented, and tested your software system**, please answer the following questions about your code in your write-up document, which you will push you're your individual repo along with your code.**

1. Please include a code snippet showing how have you used **inheritance** and **composition** in your code.
2. Please include a code snippet showing how have you used an **interface** or an **abstract class** in your code.
3. Please include code example of a **method overriding** and **method overloading** from your code, or explain why you have not used any overloading or overriding.
4. Please include a code example showing how have you used **encapsulation**, or explain why you did not need encapsulation in your code.
5. Please include a code example of **subtype polymorphism** from your code, or explain why you did not need subtype polymorphism.
6. Please include a code snippet of **generics** from your code.
7. Please include a code snippet showing how have you used some of the **built-in data collections** from the Java Collections Framework, or explain why you had no need for any data collections.
8. Please include a code snippet showing how have you used interfaces `Iterable` and `Iterator`, or explain why you had no need for these two interfaces.
9. Please include a code snippet showing how have you used interfaces `Comparable` and `Comparator`, or explain why you had no need for these two interfaces.
10. Please include a code snippet showing how have you used **regular expressions**, or explain why you had no need for it.
11. Please include a code snippet showing how have you used **nested classes**, or justify why you had no need for nested classes.
12. Please include code example showing how have you used **components of functional programming, such as lambdas and streams,** or explain why you had no need for it in your code.
13. Please include code snippet(s) showing how have you used **creational, structural and/or behavioral design patterns**. Please list which design patterns have you used, or explain why you had no need for design patterns in your solution.
14. Please include code snippets showing examples of **MVC architecture**, or justify why you had no need for **MVC architecture** in your design.
15. Please include code examples showing **data** and **stamp coupling** in your code.

Grading Rubric

5 points – Project compiles.

5 points – Project is properly documented.

5 points – Project is properly tested (test coverage of at least 70%).

5 points – Project write-up includes a section, describing how to run the program.

25 points – Project runs without crushing (occasional crushes will not result in points reduction).
10 points – Project contains substantial behavior, in accordance with the chosen problem option.

5 points – Project write-up includes an overview of the chosen problem, and high-level overview of the solution.

10 points – Project write-up includes a description of key challenges encountered during design and implementation, and how those challenges were addressed.

5 points – Project write-up acknowledges the sources and resources used in the project.

25 points – Project write-up contains meaningful answers to all the questions about the software system.