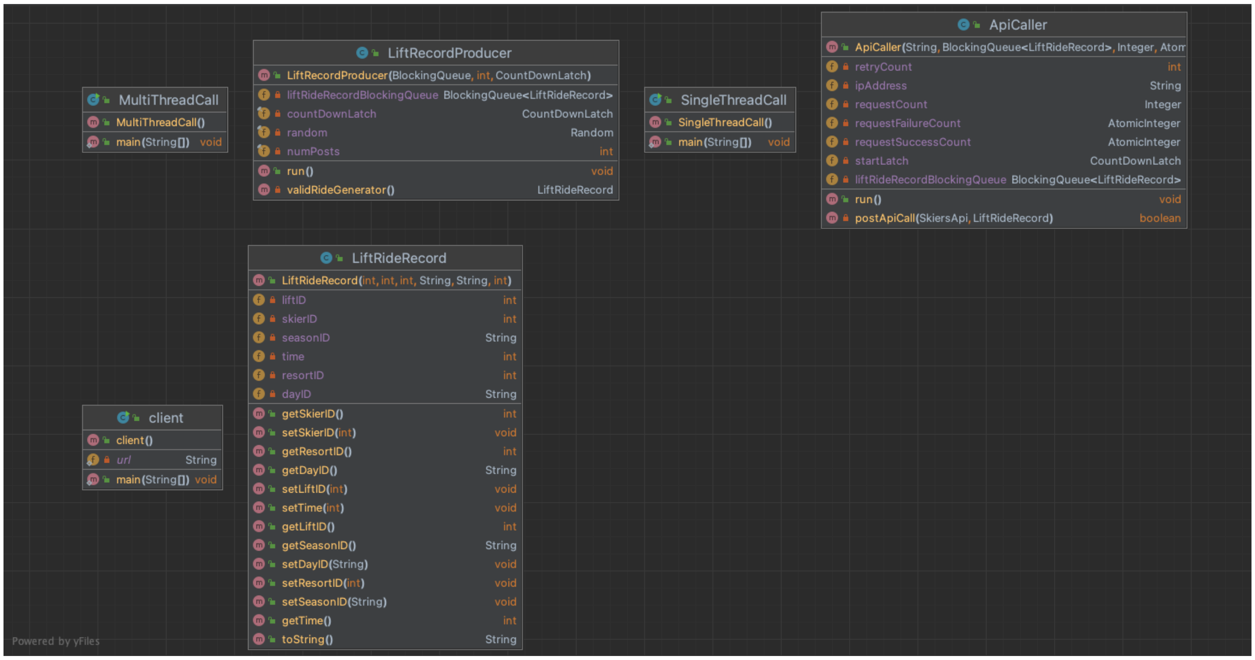
Assignment 1 Zegui Jiang

Client 1



1 - The ApiCaller class is designed to manage API calls to a ski Server, encapsulating the logic for posting lift ride data. Implementing the Runnable interface, this class is intended for use in concurrent execution environments, allowing it to be run on separate threads to facilitate parallel API requests.

run () method: Overriding the Runnable interface's run method, it contains the logic to sequentially take LiftRideRecord objects from the queue and post them to the ski resort service API. It utilizes a retry mechanism for handling API call failures. Once the designated number of requests is processed, it updates the success and failure counts and decrements the start latch, signaling completion.

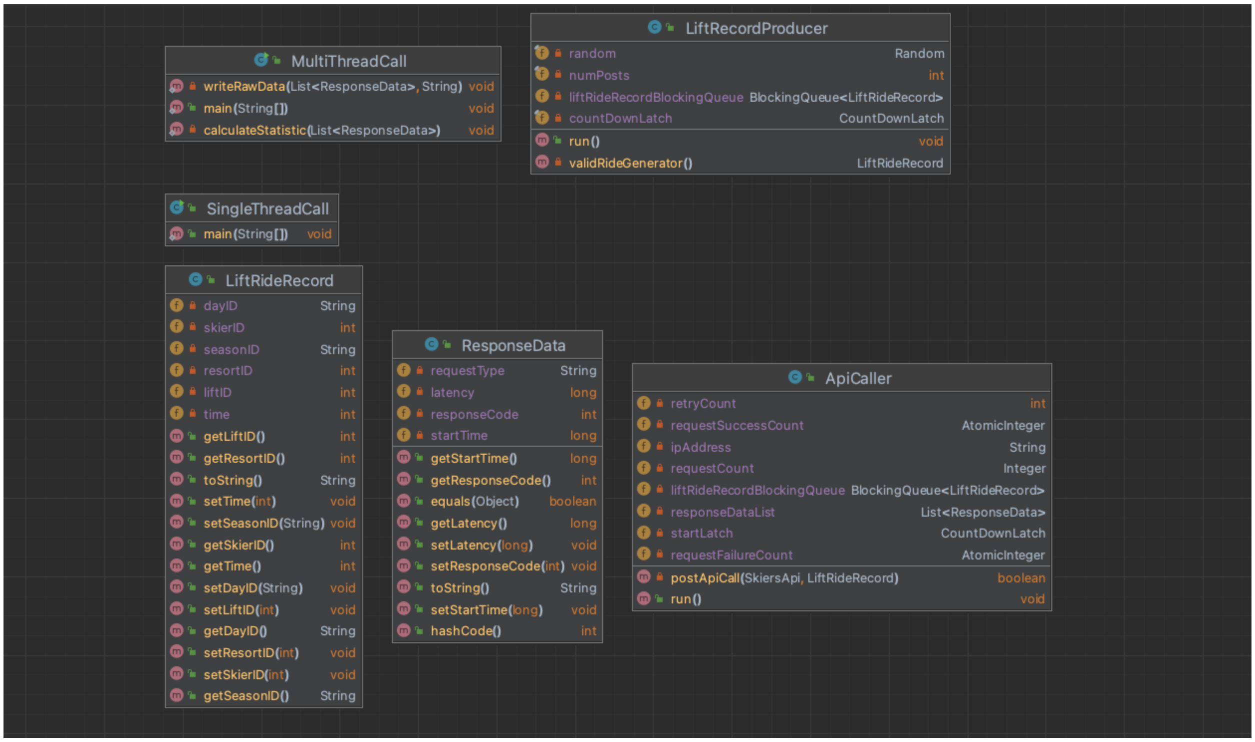
postApiCall method: A private helper method that executes the actual API call to post a LiftRide object. It implements a simple retry strategy based on the predefined retryCount for handling request failures due to network issues or server errors.

2 - The LiftRideRecord class, contained within the model package, serves as a data model representing a record of a ski lift ride within a skiing resort application.

3 – LiftRecordProducer simulate the production of ski lift ride records and enqueue them into a BlockingQueue for further processing. This class implements the Runnable interface, allowing it to be executed by a thread.

4 - MultiThreadCall class is designed to simulate a multi-threaded environment for making API calls to a server, specifically targeting the scenario of logging ski lift rides at a ski resort. This class efficiently manage and execute multiple threads for API calls and data production.

Client 2



Client 2 extends the functionality of client 1 by enhancing the MultiThreadCall class to include capabilities for writing raw data to a CSV file and calculating statistical metrics. To fulfill these additional requirements, a new class named ResponseData is introduced to encapsulate the response records.

The MultiThreadCall class in client 2 would have these new responsibilities:

Writing Response Records: After each API call, the response details are captured in an instance of the ResponseData class. This object includes all relevant information that needs to be logged, such as timestamps, response status, and any payload data returned by the API.

Calculating Statistics: The class is responsible for computing various performance metrics based on the response data collected. These statistics might include the total number of requests, success rate, failure rate, average response time, and other relevant performance indicators.

Throughput Overtime:

```
/Users/monarch/Library/Java/JavaVirtualMachines/corretto-11.0.22/Contents/Home/bin/java ...
requestSuccessCount 0
API Finished
Summary:
Number of thread:1
Number of successful requests: 10000
Number of fail requests: 0
Total run time: 336157
Response Time: 33.6157 ms/request
RPS:     29 requests/second
```

```
/Users/monarch/Library/Java/JavaVirtualMachines/corretto-11.0.22/Contents/Home/bin/java ...
Statistic Metrics
Mean Response Time: 37.585765 ms
Median Response Time: 31.0 ms
P99 Response Time: 123 ms
Min Response Time: 15 ms
Max Response Time: 337 ms
Summary:
Number of thread: 32
Number of successful requests: 200000
Number of fail requests: 0
Total run time: 232905
Response Time: 1.164525 ms/request
RPS: 858 requests/second
```

```
/Users/monarch/Library/Java/JavaVirtualMachines/corretto-11.0.22/Contents/Home/bin/java ...
Statistic Metrics
Mean Response Time: 39.5409 ms
Median Response Time: 33.0 ms
P99 Response Time: 126 ms
Min Response Time: 13 ms
Max Response Time: 374 ms
Summary:
Number of thread: 50
Number of successful requests: 200000
Number of fail requests: 0
Total run time: 157401
Response Time: 0.787005 ms/request
RPS: 1270 requests/second

Process finished with exit code 0
```

```
/Users/monarch/Library/Java/JavaVirtualMachines/corretto-11.0.22/Contents/Home/bin/java ...
Statistic Metrics
Mean Response Time: 40.0843 ms
Median Response Time: 33.0 ms
P99 Response Time: 128 ms
Min Response Time: 14 ms
Max Response Time: 297 ms
Summary:
Number of thread: 100
Number of successful requests: 200000
Number of fail requests: 0
Total run time: 79239
Response Time: 0.396195 ms/request
RPS: 2524 requests/second

Process finished with exit code 0
```

Littles law:
Base on little's law, N = λW.
if we predict 100 threads. Throughput = 100 / 35 * 1000 = 2857
the actual performance is 2524 is close to 2857.

Here is Summary:

1. With **1 thread**, the system processed **10,000** successful requests at a rate of **29 RPS**.
2. With **32 threads**, the system processed **200,000** successful requests at a rate of **858 RPS**.

3. With **50 threads**, the system processed **200,000** successful requests at a rate of **1270 RPS**.
4. With **100 threads**, the system processed **200,000** successful requests at a rate of **2524 RPS**.

As we analyze these data points, we notice a clear trend: as the number of threads increases, the throughput of the system also increases. This is expected behavior in multithreaded processing up to a certain point, as parallel execution allows more tasks to be processed simultaneously.

- **Single Thread**: The throughput is the lowest with a single thread, as requests are processed sequentially.
- **Increasing Threads**: Increasing the number of threads to 32 results in a significant increase in throughput due to parallel processing.
- **Optimal Throughput**: As we move to 50 threads, we see an increase in throughput, suggesting that the system can efficiently manage and utilize these threads without significant