Lab #2: Mocking dependencies in unit testing

Learning objectives

- Prepare a project to run unit tests (JUnit 5) and mocks (Mockito 3.x), with mocks injection (@Mock).
- Write and execute unit tests with mocked dependencies.
- Play with mock behaviors: strict/lenient verifications, advanced verifications, etc.

Lab activities

1a/ Implement the test case illustrated with the following classes, with respect to the **StockPortfolio#getTotalValue()** method. The method is expected to calculate the value of the portfolio by summing the current value (looked up in the stock market) of the owned stocks.
Be sure to use:

- Maven-based Java application project;
- Mockito framework (mind the maven dependencies).

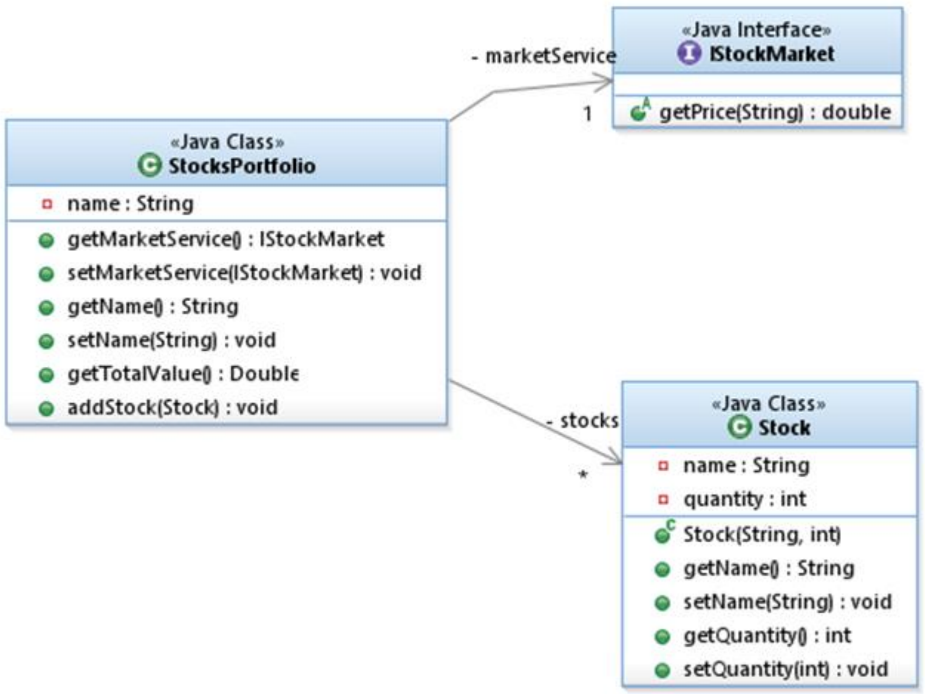

Figure 1: Classes for the StocksPortfolio use case.

a) Create the classes. You may write the implementation of the services before or after the tests.

b) Create the test for the getTotalValue(). As a guideline, you may adopt this outline:

```
1. Prepare a mock to substitute the remote service (@Mock annotation)
2. Create an instance of the subject under test (SuT) and use the mock to set the
   (remote) service instance (you may prefer to use @InjectMocks)
3. Load the mock with the proper expectations (when...thenReturn)
4. Execute the test (use the service in the SuT)
5. Verify the result (assert) and the use of the mock (verify)
```

Notes:

- Mind the JUnit version. For JUnit 5, you should use the @ExtendWith annotation to integrate the Mockito framework.
- Some IDE may not support JUnit 5 integration; you may need to further configure the POM.
- See a quick reference of Mockito syntax and operations.

1b/ Instead of the JUnit core asserts, you may use the Hamcrest library to create more human-readable assertions. Consider using this library in the previous example, in particular, assertThat(), is().

2/ Consider an application that needs to perform reverse geocoding to find a zip code for a given set of GPS coordinates. This service can be obtained in the Internet (e.g.: using the MapQuest API).

a) Create the objects represented in Figure 1. TqsHttpClient represents a service to initiate HTTP requests to remote servers. **You don't need to implement TqsHttpBasic**; in fact, you should provide a substitute for it.

b) Consider that we want to verify the AddressResolver#findAddressForLocation, which invokes a remote geocoding service, available in a REST interface, passing the site coordidates. Which is the service to fake?

c) To create a test for findAddressForLocation, you will need to know the exact response of the geocoding service for a sample request. Assume that we will use the MapQuest API. Use the browser or an HTTP client to try some samples so you know what to test for (example 1).

d) Implement a test for AddressResolver#findAddressForLocation using a mock.

e) Besides de "success" case, consider also testing for alternatives (e.g.: invalid coordinates should raise an exception).

This getting started project can used in your implementation.

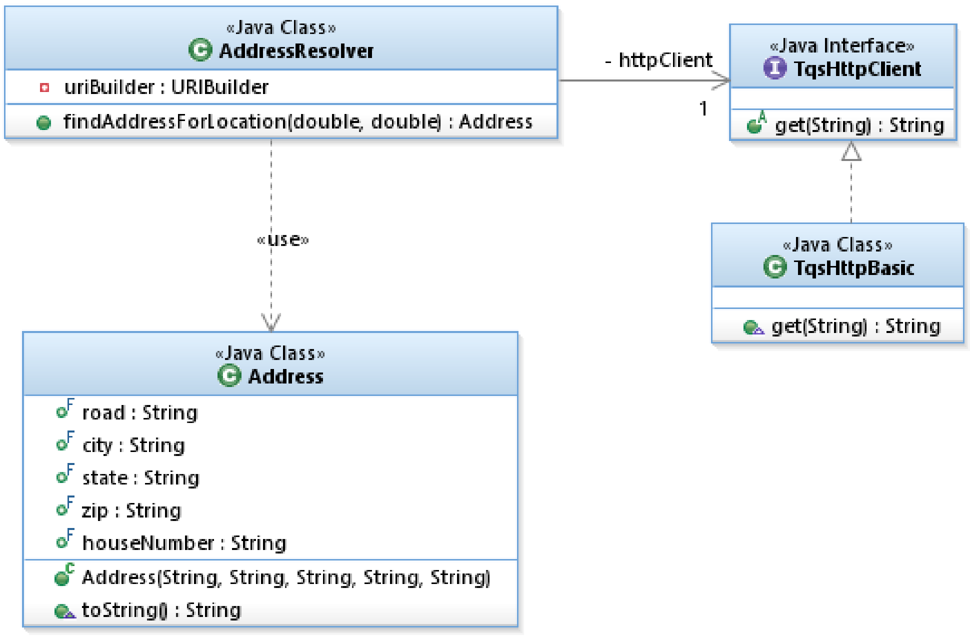

Figure 2: Classes for the geocoding use case.

3/ Consider you are implementing an integration test, and, in this case, you would use the real implementation of the module, not the mocks, in the test.
Create new test class and be sure its name end with "IT" (e.g.: GeocodeTestIT).
Copy the tests from the previous exercise into this new test class.
Remove all support for mocking (no dependencies on Mockito imports).
Correct the test implementation, so it used the real module.

If the “failsafe” maven plugin is configured, you should get different results with:

```
$ mvn test
$ mvn package failsafe:integration-test
```

Explore

JUnit 5 cheat sheet ;