

TQS Lab activities

v2020-03-05

Lab 1: Unit testing with JUnit 5	1
Learning objectives	1
Preparatory readings	1
Key points	1
Lab activities	2
Explore	4
Lab 2: Mocking dependencies in unit testing	5
Learning objectives	5
Lab activities	5
Explore	7
Lab 3: Functional testing with web automation	7
Prepare	7
Key Points	7
Lab	7
Explore	8
Lab 4: Behavior-driven development (Cucumber in Java)	8
Prepare	8
Key Points	8
Lab	9

Lab 1: Unit testing with JUnit 5

Learning objectives

- Identify relevant unit tests to verify the contract of a module.
- Write and execute unit tests using the JUnit framework.
- Link the unit tests results with further analysis tools (e.g.: code coverage)

Preparatory readings

- The [test pyramid concept](#).
- Optional: [TDD & Unit testing in IntelliJ](#) tutorial.

Key points

- Unit testing is when you (as a programmer) write test code to verify units of (production) code. A unit represents some small subset of a much larger end-to-end-behavior. A true “unit” does not have dependencies on other (external) components.
- Unit tests help the developers to (i) understand the module contract (what to construct); (ii) document the intended use of a component; (iii) prevent regression errors; (iv) increase confidence on the code.
- When following a TDD approach, typically you go through a cycle of [Red-Green-Refactor](#). You'll run a

test, see it fail (go red), implement the simplest code to make the test pass (go green), and then refactor the code so your test stays green and your code is sufficiently clean.

- JUnit and TestNG are popular frameworks for unit testing in Java.

JUnit best practices: unit test one object at a time

A vital aspect of unit tests is that they're finely grained. A unit test independently examines each object you create, so that you can isolate problems as soon as they occur. If you put more than one object under test, you can't predict how the objects will interact when changes occur to one or the other. When an object interacts with other complex objects, you can surround the object under test with predictable test objects. Another form of software test, integration testing, examines how working objects interact with each other. See chapter 4 for more about other types of tests.

Lab activities

Be sure that your developer environment meets the following requirements:

- Java development environment ([JDK](#)), v8 or v11. Note that you should install it into a path without spaces or special characters (e.g.: avoid \Users\José Conceição\Java).
- [Maven configured](#) to run in the command line.
- Java capable IDE, such as [IntelliJ IDEA](#).

Implement a stack data structure (TqsStack) with appropriate unit tests. Be sure to adopt a **write-the-tests-first** workflow:

- a) Create a new **maven-based**, Java standard application.

Note: use the IDE features. If you are not sure if the IDE can generate a maven-compatible structure, consider using this [starter project](#).

- b) Create the required classes definition (**just the “skeleton”**, not the methods body; you may need to add dummy return values). The code should compile, though the implementation is incomplete.
- c) Write the unit tests that will verify the TqsStack contract.

You may use the IDE features to generate the testing class; note that the [IDE support will vary](#). Be sure to use [JUnit 5.x](#).

Important: for [maven based projects](#), check the proper POM.xml [dependencies for JUnit 5](#). If you get an error saying that the 1.5 version is no longer supported, specify the [target compiler version](#) in the POM.

Your tests will verify several [assertions that should evaluate to true](#) for the test to pass. See [some examples](#).

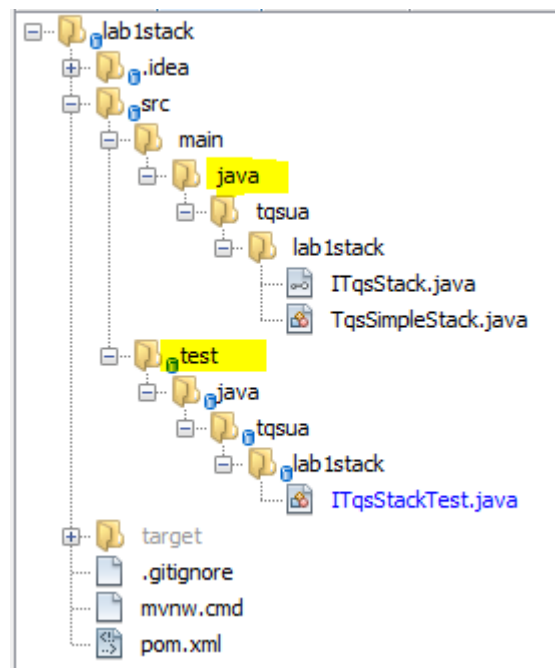
- d) Run the tests and prove that TqsStack implementation is not valid yet (the tests should fail for now, the first step in [Red-Green-Refactor](#)).
- e) Correct/add the missing implementation to the TqsStack;
- f) Run the unit tests.
- g) Iterate from steps d) to f) and confirm that all tests pass.

Stack operations:

- push(x): add an item on the top
- pop: remove the item at the top
- peek: return the item at the top (without removing it)
- size: return the number of items in the stack
- isEmpty: return whether the stack has no items

What to test¹:

- a) A stack is empty on construction.
- b) A stack has size 0 on construction
- c) After n pushes to an empty stack, $n > 0$, the stack is not empty and its size is n
- d) If one pushes x then pops, the value popped is x.
- e) If one pushes x then peeks, the value returned is x, but the size stays the same
- f) If the size is n, then after n pops, the stack is empty and has a size 0
- g) Popping from an empty stack does throw a NoSuchElementException [[You should test for the Exception occurrence](#)]
- h) Peeking into an empty stack does throw a NoSuchElementException
- i) For bounded stacks only, pushing onto a full stack does throw an IllegalStateException

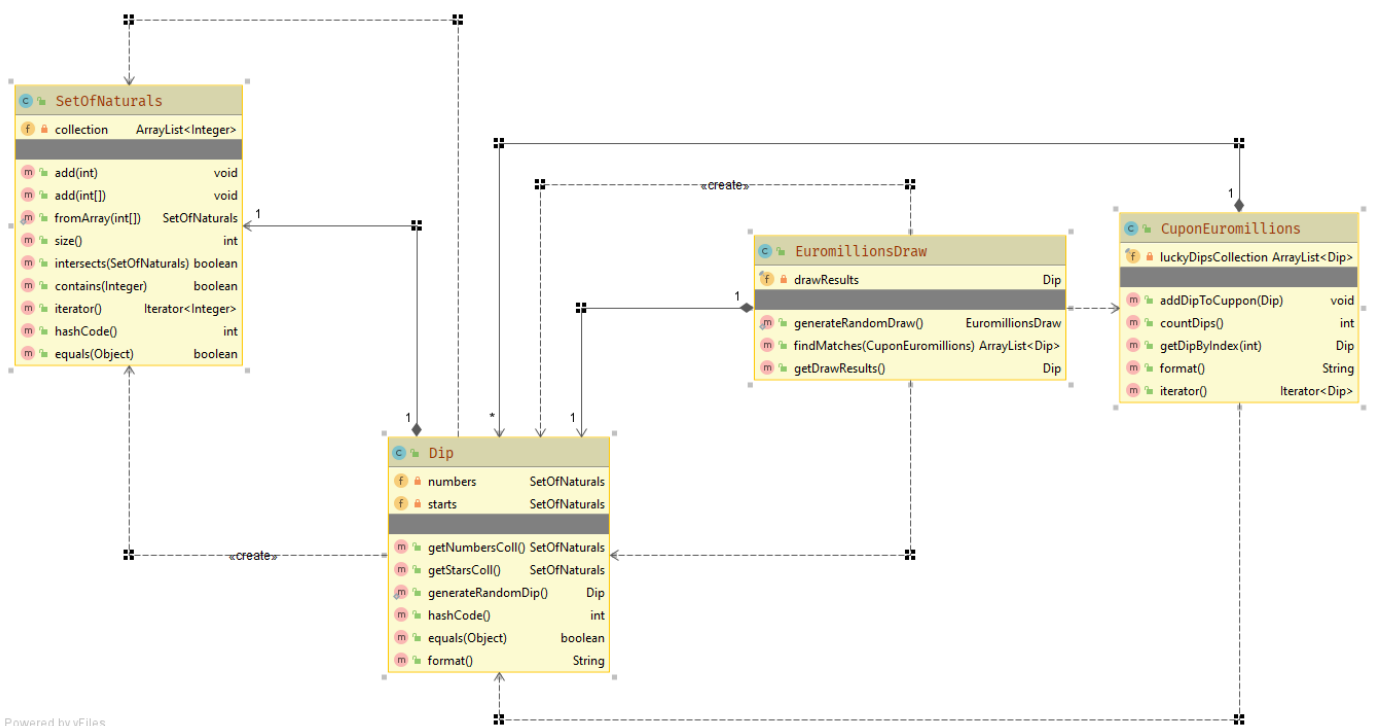


2a/ Pull the [“euromillions-play” project](#) and correct the code (or the tests themselves, if needed) to have the existing unit tests passing.

For test:	You should:
testFormat	Correct the <u>implementation</u> of Dip#format so the tests pass.
testConstructorFromBadArrays	Implement new <u>test</u> logic to confirm that an exception will be raised if the arrays have invalid numbers (wrong count of numbers of starts)

Note: you may suspend temporary a test with the [@Disable](#) tag (useful while debugging the tests themselves).

¹ Adapted from <http://cs.lmu.edu/~ray/notes/stacks/>



2b/ The class `SetOfNaturals` represents a set (no duplicates should be allowed) of integers, in the range $[1, +\infty]$. Some basic operations are available (add element, find the intersection...). What kind of unit test are worth writing for the entity `SetOfNaturals`? Complete the project, adding the new tests you identified.

2c/ Note that the provided code includes “magic numbers” (2 for the number of stars, 50 for the max range,...). Refactor the code to extract constants and eliminate the “magic numbers” bad-smell.

2d/ Assess the coverage level in project “Euromillions-play”.

[Configure the maven project to run Jacoco analysis.](#)

Run the maven “test” goal and then “jacoco:report” goal. You should get an HTML report under `target/jacoco`.

Interpret the results accordingly. Which classes/methods offer less coverage? Are all possible decision branches being covered?

Note: IntelliJ has an integrated option to run the tests with the coverage checks (without setting the Jacoco plugin in POM). But if you do it at maven level, you can use this feature in multiple tools.

Explore

- Book: [JUnit in Action](#).
- Vogel's [tutorial on JUnit](#). Useful to compare between JUnit 4 and JUnit 5.
- [Working effectively with unit testing](#) (podcast).

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).

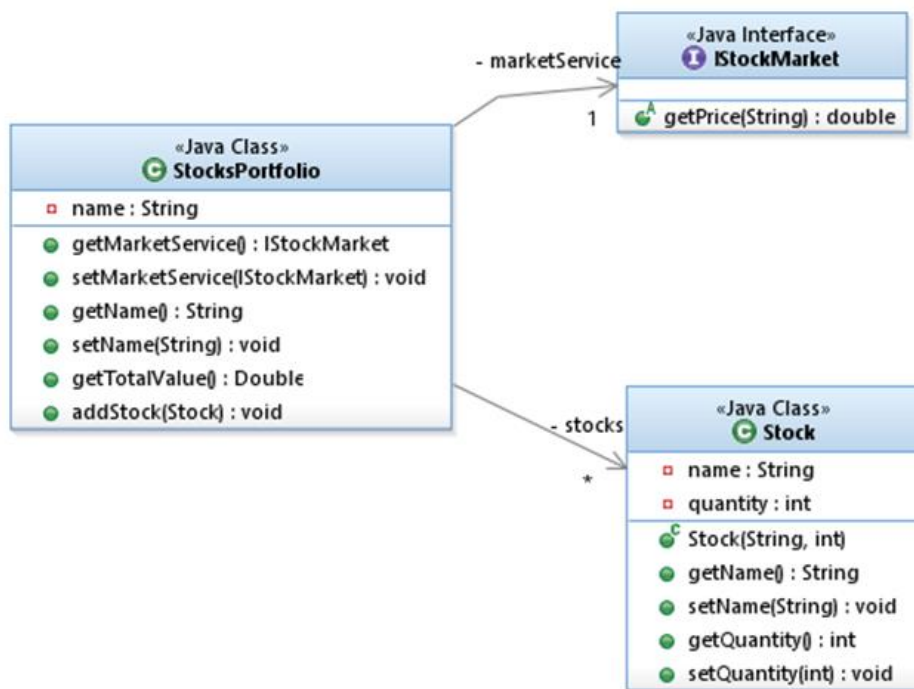


Figure 1: Classes for the StocksPortfolio use case.

- Create the classes. You may write the implementation of the services before or after the tests.
- Create the test for the `getTotalValue()`. As a guideline, you may adopt this outline:
 - Prepare a mock to substitute the remote service (`@Mock` annotation)
 - 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`)
 - Load the mock with the proper expectations (`when...thenReturn`)
 - Execute the test (use the service in the SuT)
 - 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](#)).

- 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.
- Consider that we want to verify the `AddressResolver#findAddressForLocation`, which invokes a remote geocoding service, available in a REST interface, passing the site coordinates. Which is the service to fake?
- 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](#)).
- Implement a test for `AddressResolver#findAddressForLocation` using a mock.
- Besides de “success” case, consider also testing for alternatives (e.g.: invalid coordinates should raise an exception).

This [getting started project](#) can be used in your implementation.

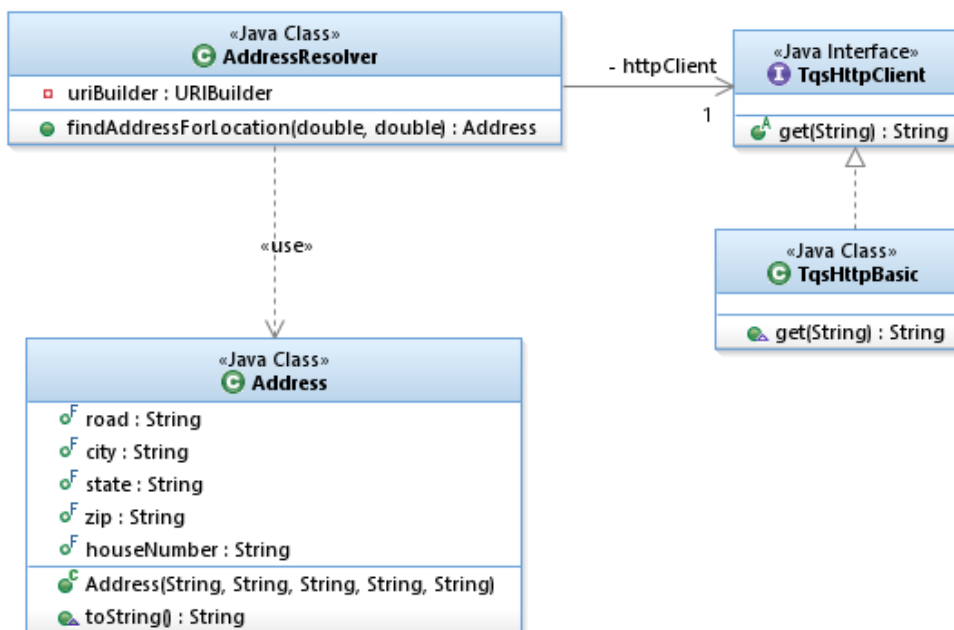


Figure 2: Classes for the geocoding use case.

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](#).

Lab 3: Functional testing with web automation

Prepare

- [Selenium and separation of concerns](#)

Key Points

- Acceptance tests (or functional test) exercise the user interface of the system, as if a real user was using the application. The system is treated as a black box.
- Browser automation (control the browser interaction from a script) is an essential step to implement acceptance tests on web applications. There are several frameworks for browser automation (e.g.: Puppeteer); for Java, the most used framework is the WebDriver API, provided by Selenium (that can be used with JUnit or TestNG engines).
- The test script can easily get “messy” and hard to read. To improve the code (and its maintainability) we could apply the Page Objects patterns.
- Web browser automation is also very handy to implement “smoke tests”.

Lab

Selenium works with multiple browsers but, for sake of simplicity, the samples will be discussed with respect to Chrome/Chromium; you may adapt for Firefox.

Suggested setup:

Install Chrome/Chromium in your system (if needed), using the default installation paths.

Download the [ChromeDriver](#) and make sure it is available available in the PATH. ([GeckoDriver](#) for Firefox)

Install the “Selenium IDE” browser plugin. (or, alternatively, the Katalon Recorder).

In this lab:

1. Create a web automation with Selenium IDE recorder
2. Run the test as a Java project (JUnit 5 + Selenium)
3. Use the Web Page Object pattern

1/ Create a web automation with Selenium IDE recorder

Access the Redmine demo site and create a temporary account (<http://demo.redmine.org>)

Be sure to logout before recording the test.

Create (record) an automation macro with the Selenium IDE recorder tool to test login (a [quick start for Katalon](#) is available, which is similar to Selenium IDE):

- a) Open <http://demo.redmine.org>
- b) Sign-in with your credentials
- c) Assert that you have successfully logged in (by verifying the presence of the username)
- d) Logout

... and Stop recording. Test your macro (replay).

Add a new step, at the end, to confirm that, after logout, the home shows the “Sign in” option present. Enter this assertion “manually” (in the editor, but not recording).

2a/ Run the test as a Java project (JUnit 5, Selenium)

Prepare a (new) project to run JUnit tests and Selenium ([sample POM.xml](#) available; [alternative site](#)).

Take note of the information [in this page](#) under “quick reference”; then, in the section “Local browsers”, pick the example that suites your setup and run the test (as you usually do with JUnit).

You will have to deploy the WebDriver implementation (binary) for you browser [→ [download browser driver](#)]. Be sure to include in the system PATH.

2b/ Export and run the test (Webdriver)

Export the test from Selenium IDE into a Java test class and include it in the previous project.

Refactor the code that was generated to be compliant with JUnit 5 and the [Selenium-Jupiter extension](#).

Run the test (programmatically).

3/ Use the Web Page Object pattern

Consider the [example discussed here](#).

Implement the “Page object pattern” for a cleaner and more readable test, as suggested.

Notes:

Execute the suggested steps to interactively record the test case. Export it to Java and run the test with JUnit automation, refactoring for JUnit 5.

The text in the tutorial is somewhat old. You may **need to adapt to the current implementation** of the site under test (e.g.: IDs of page elements,...).

You should **stop** at “Increased readability” with Cucumber.

Explore

- [Puppeteer](#) - a Node library which provides a high-level API to control headless Chrome/Chromium.
- Another, more recent, [Page Object Model example](#).

Lab 4: Behavior-driven development (Cucumber in Java)

Prepare

Key Points

- The [Cucumber framework](#) enables the concept of “executable specifications”: with Cucumber we use concrete examples to specify what we want the software to do. **Scenarios are written before production code.**
- Cucumber executes features (test scenarios) written with the [Gherkin language](#) (readable by [non-programmers too](#)).
- The steps included in the feature description (scenario) must be mapped into Java test code by annotating test methods with matching “expressions”. [Expressions](#) can be (traditional) regular expressions or the (new) Cucumber expressions.

Lab

Bear in mind that the integration of JUnit 5 and Cumcumber is still very recent and most of the samples available in the internet are still based on JUnit 4.

In this lab you will:

- 1/ Create a simple cumcumber-enabled project
- 2/ Book search example
- 3/ Integrate Cucumber with Selenium Webdriver

1/ Create a simple cucumber-enabled project

This example assumes that you have Maven available in the path and you can [invoke mvn from the command line](#). Check with:

Go through the [Cucumber getting started tutorial](#) for Java.

The example uses the command line and the IntelliJ IDEA (you may use other IDE, but you will need to adapt the instructions).

Note that:

- Start the project from the suggested Maven archetype `io.cucumber:cucumber-archetype`
- You need a test class that activates the Cucumber runner; that is the purpose of the “RunCucumberTest” file automatically included in this archetype.
- the features are stored under `src/test/resources/` and the folder structure (under `resources`) must mirror the package hierarchy under `src/test/java/`. In this sample: `src/test/resources/hellocucumber/` and `src/test/java/hellocumcumber/`
- IntelliJ recognizes the `.feature` file type. You can even select “Run all features” to run the Cucumber tests.

2/ Book search example

To get into the “spirit” of BDD, partner with a colleague, and jointly write a couple of features to verify a book search user story. Consider a few search options (by year, etc).

Take the approach discussed in [this example](#), and write your own tests. Feel free to add different scenarios/features.

The sample uses Cucumber v2, but you should write the test steps using Cucumber 3x. Some changes are required:

In the “book search” story:

- You will need to change the parameters placeholder in the steps definition. Prefer the “cucumber expressions” (instead of regular expressions). [→ [partial snippet](#)]
- migrate the date formatter option, by creating a new datatype in a “Configurer” class, which should be placed in the same package as the test steps [→ [possible solution snippet](#)]. This defines a new custom parameter type (“`date_iso_local_date_time`”)
- The dates in the feature description need also to match the date mask used (`aaaa-mm-dd`).

In the “salary” story:

- Adapt the data table definition for the Salary use case [→ [possible solution snippet](#)]. This allows to extract a `List<Employee>` from the feature definition and use it as a parameter.

```
public void
the_salary_management_system_is_initialized_with_the_following_data(final
List<Employee> employees)
```

Notes:

→ you may build on the previous project or start with a new one, but stick with the base archetype (io.cucumber:cucumber-archetype).

3/ Integrate Cucumber with Selenium Webdriver

Implement the brief sample using [cucumber and selenium webdriver](#) to declare expressive web automation tests.

You may now extend this strategy to revisit the example (Weather forecast) of lab 3.