




TQS Lab activities

v2022-03-10

Work submission

You should create a [new] personal (git) repository for your **individual portfolio** with respect to the work proposed for the labs (e.g.: tqs_123567 , in which the number is your student number). Keep a **clean organization** that maps the exercise structure, e.g.:  lab1/lab1_1;  lab1/lab1_2;  lab2/lab2_1

You are expected to keep your repo (portfolio) up to date and complete. Teachers will select a few exercises later for assessment [not all, but representative samples].

Lab activities

Be sure that your developer environment meets the following requirements:

- Java development environment ([JDK](#); v11 suggested). 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. Check with:

```
$ mvn --version
```
- Java capable IDE, such as [IntelliJ IDEA](#) (version “Ultimate” suggested) or [VS Code](#).

1 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)

Key points

- Unit testing is when you (as a programmer) write test code to verify units of (production) code. A unit is a small, coherent subset of a much larger solution. A true “unit” should not depend of the behavior of other (collaborating) modules.
- 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 in the code.
- 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.

1.1

In this exercise, you will implement a stack data structure (TqsStack) with appropriate unit tests. Be sure to adopt a **write-the-tests-first** workflow:

- a) Create a new project (**maven-based**, Java standard application). You may need to update the Java version in the POM.xml:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>${maven.compiler.source}</maven.compiler.target>
</properties>
```
- b) Add the required dependencies to run JUnit 5 tests¹. Here are some sources:
 - Adapt from the [quick start project](#) for Maven².
 - Adapt from [this tutorial](#).
- c) Create the required class definition (**just the “skeleton”**, do not implement the methods body yet; you may need to add dummy return values). The **code should compile**, but the **implementation is yet incomplete**.
- d) 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](#).

Your tests will verify several [assertions that should evaluate to true](#) for the test to pass.
- e) 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](#)).
- f) Correct/add the missing implementation to the TqsStack;
- g) Run the unit tests.
- h) Iterate from steps d) to f) and confirm that all tests pass.

Suggested stack contract:

- 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 test³:

- 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

¹ If using IntelliJ: you may skip this step and ask, later, the IDE to fix JUnit imports.

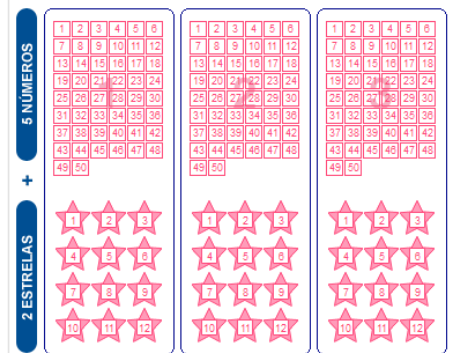
² Delete the “pom-SNAPSHOT.xml”, if you are cloning the project to use as a quick starter.

³ Adapted from <http://cs.lmu.edu/~ray/notes/stacks/>

1.2

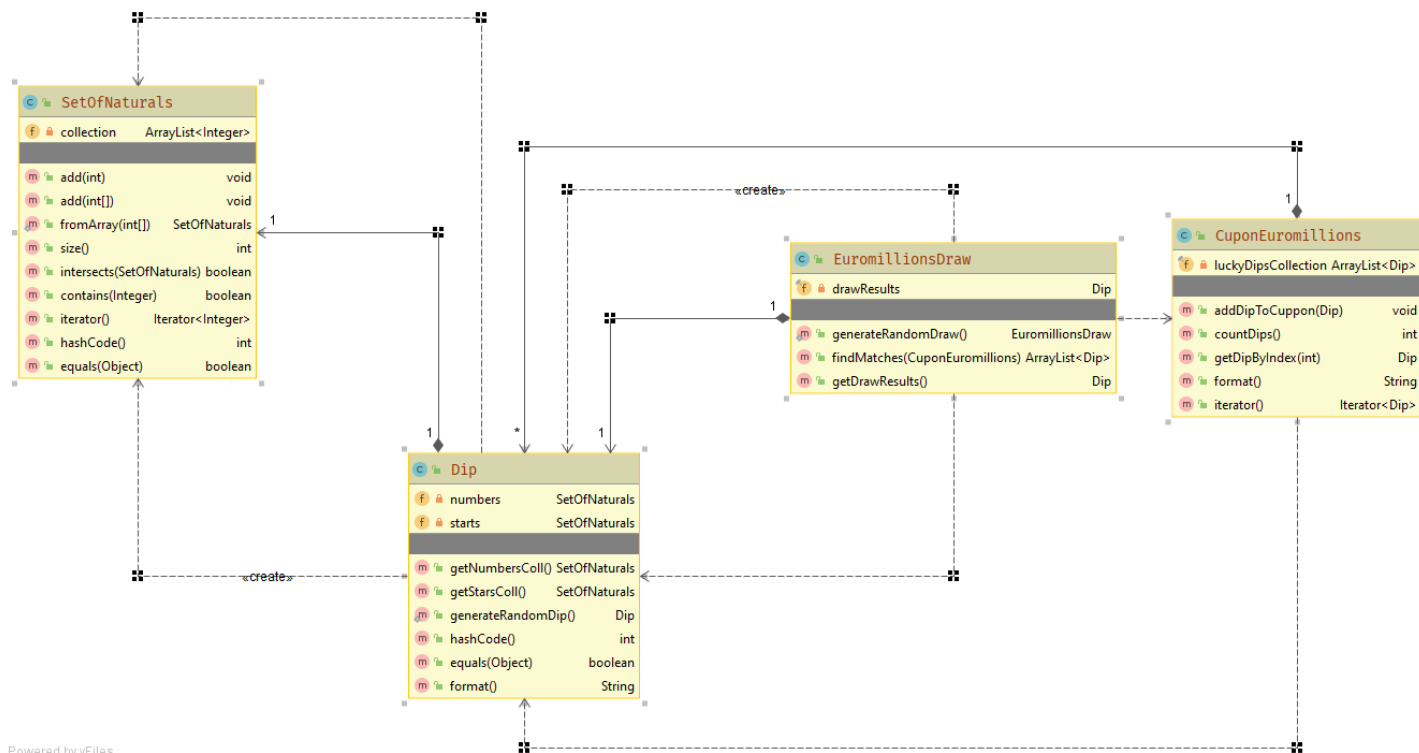
Let us consider the “[Euromilhões](#)” use case.

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



For the (failing) 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 or stars)

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



Powered by yFiles

2b/ The sample project was prepared for the scenario in which the range for the “stars” was 1..10. However, the rules have changed, and the range is 1..12. Be sure to include a test to verify the [new] ranges.

2c/ Note that the code provided includes “magic numbers” (2 for the number of stars, 50 for the max range in numbers, [was] 10 for the max range in starts...). Refactor the code to extract constants and eliminate the “magic numbers”.

2d/ 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.

2e/ 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`.

```
$ mvn clean test jacoco:report
```

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.

Troubleshooting some frequent errors

➔ “Test are run from the IDE but not from command line.”

Be sure to configure the Surefire plug-in in Maven ([example](#)).

Explore

- JetBrains Blog on [Writing JUnit 5 tests](#) (with video).
- Book: [JUnit in Action](#). Note that you can access it from the [OReilly on-line library](#).
- JUnit 5 [cheat sheet](#).
- Vogel's [tutorial on JUnit](#). Useful to compare between JUnit 4 and JUnit 5.