

TQS Lab activities

v2023-02-16

TQS Lab activities	1
Introductory notes and setup	1
Lab 1 Unit testing (with JUnit 5)	1
1.1 Stack contract	2
1.2 EuroMillions	3

Introductory notes and setup

Work submission

You should create a personal (git) repository for your TQS **individual portfolio** in which you will be including your solutions for the labs (e.g.: **tqs_123567**, the number being your student number). Keep a **clean organization** that maps the exercise structure, e.g.: `lab1/lab1_1`; `lab1/lab1_2`; `lab2/lab2_1`; `lab2/lab2_2`...

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 or v17 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](#).

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 on 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 Stack contract

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:

- Create a new project (**maven project** for a Java standard application). You may need to update the Java version in the POM.xml and other dependencies. You may “clone” from a sample project:
 - Adapt from the [quick start project](#) for Maven¹.

- Add the required dependencies to run JUnit 5 tests².
Example:

- [sample content for POM.xml](#) (note the elements: **junit-jupiter** and **maven-surefire-plugin**)

- Create the required class definition (**just the “skeleton”**, do not implement the methods body yet!). The **code should compile**, but the **implementation is yet incomplete** (you may need to add dummy return values).

- Write 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](#). [Mixing JUnit 4 and JUnit 5 dependencies will prevent the test methods to run as expected!]

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

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

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

```
<!-- ... -->
<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
  </dependency>
<!-- ... -->
</dependencies>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M7</version>
    </plugin>
    <plugin>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>3.0.0-M7</version>
    </plugin>
  </plugins>
</build>
<!-- ... -->
```

TqsStack<T>		
m	TqsStack()	
f	collection	LinkedList<T>
m	pop()	T
m	size()	int
m	peek()	T
m	push(T)	void
m	isEmpty()	boolean

¹ Delete the “pom-SNAPSHOT.xml”, if you are cloning the project to use as a quick starter.

² If using IntelliJ: you may skip this step and ask, later, the IDE to fix JUnit imports.

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

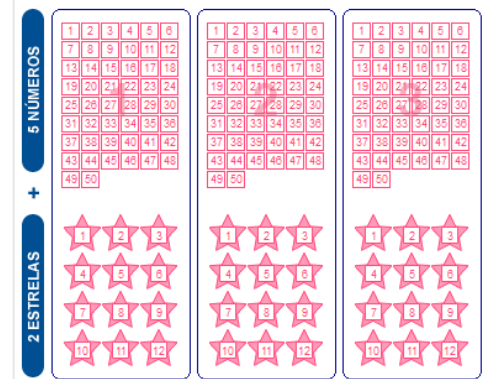
1.2 EuroMillions

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

2a/ Pull the “[euromillions-play](#)” project

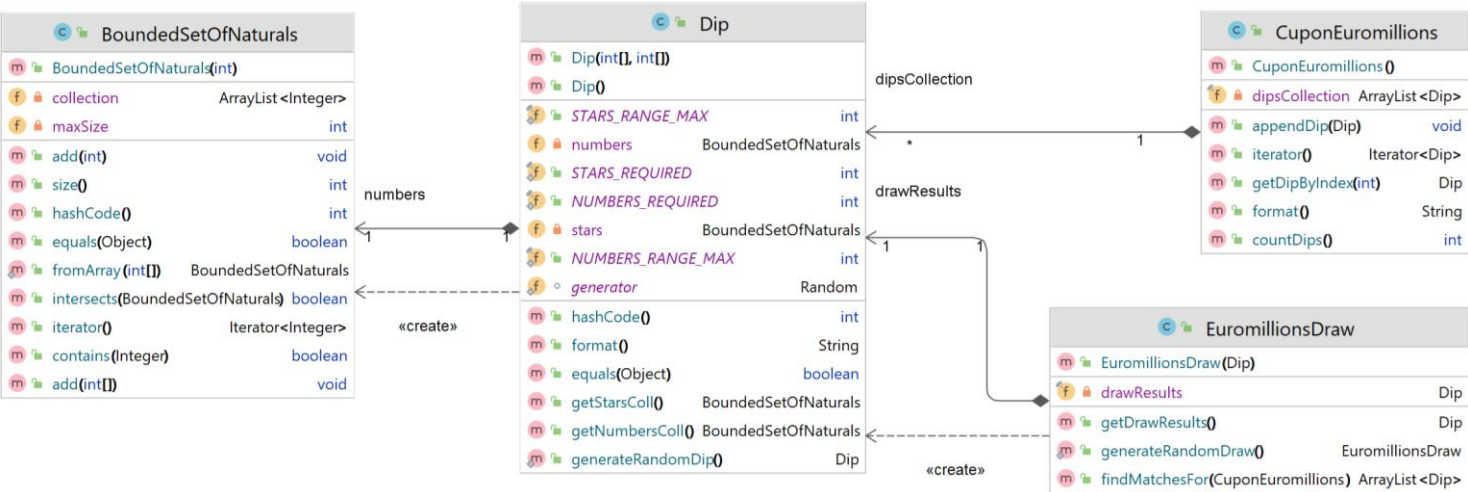
The supporting implementations is visualized in the class diagram that follows.

Get familiar with the solution and existing tests.



Class	Purpose
BoundedSetOfNaturals	Reusable set data structure no duplicates allowed (it is a Set) only natural numbers in the range $[1, +\infty]$. the max size of the set (count of elements) is bounded to a limit allows set operations (contains element?, append element, calculate intersection with another set,...)
Dip	A collection of 5 “numbers” and 2 “stars” (a “column” in the Euromillions playing coupon)
CouponEuromillion	One or more Dips, representing a bet from a player.
EuromillionsDraw	Holds the winning dip and can find matched for a given player coupon.

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



2b/ Make the necessary changed for the existing (non-disabled) unit tests pass.

For the (failing) test:	You should:
testConstructorFromBadRanges	Change Dip implementation. Be sure to raise the expected exception if the arrays have invalid numbers (out of range numbers)

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

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

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

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

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

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

Collect evidence of the coverage for “BoundedSetOfNaturals”.

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.

2c/

Consider the class `BoundedSetOfNaturals` and its expected contract.

What kind of unit test are worth writing for proper validation of `BoundedSetOfNaturals`?

Complete the project, adding the tests you have identified. (You may also enhance the implementation of `BoundedSetOfNaturals`, if necessary.)

2d/

Run Jacoco coverage analysis and compare with previous results. In particular, compare the “before” and “after” for the `BoundedSetOfNaturals` class.

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](#).
- Book: “[Mastering Software Testing with JUnit 5](#)” and associated [GitHub repository](#) with examples
- JUnit 5 [cheat sheet](#).