1 Lab: Unit tests

v2025-02-13

1.0 Introduction

Learning objectives

- Identify relevant unit tests to verify the contract of a module.
- Write and execute unit tests using the JUnit 5 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-scoped, 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.

Useful resources

- 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
- JetBrains Blog on Writing JUnit 5 tests (with vídeo).

1.1 Simple Stack contract

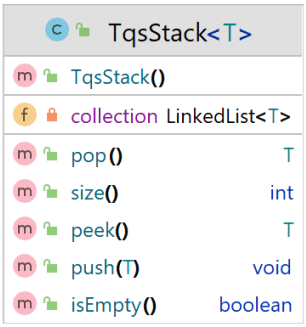
In this exercise, you will implement a stack data structure (TqsStack) with appropriate unit tests.

Be sure to force a **write-the-tests-first** workflow as described ahead. Don't use AI to generate the first tests.

a) Create a new project (**maven project** for a Java standard application). Add the required dependencies to run JUnit 5 tests (check POM.xml in linked sample).

   Note the elements: **junit-jupiter** (version 5.x) and **maven-surefire-plugin** (>2.22 required)

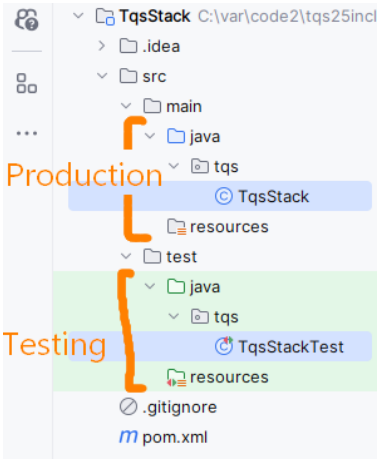
b) Create the required class definition (**just the "skeleton"**, do not implement the methods body yet!).



The **code should compile** (you may need to add dummy return values).
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

c) Write unit tests that will verify the TqsStack contract.



Consider testing the following cases[1]:

- A stack is empty on construction
- A stack has size 0 on construction
- After n > 0 pushes to an empty stack, the stack is not empty and its size is n
- If one pushes x then pops, the value popped is x.
- If one pushes x then peeks, the value returned is x, but the size stays the same
- If the size is n, then after n pops, the stack is empty and has a size 0
- Popping from an empty stack throws a NoSuchElementException
- Peeking into an empty stack throws a NoSuchElementException
- For bounded stacks only, pushing onto a full stack throws an IllegalStateException

Check this page for sample unit test syntax.

Keep in mind that maven expects test code to be placed in a specific folder. You may use the IDE features to generate the initial (empty) 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 from running as expected!

Test code should include several assertions that should evaluate to true for the test to pass.

d) 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). You can run from the IDE options or using maven support:

```
$ mvn test
```

[1] Adapted from http://cs.lmu.edu/~ray/notes/stacks/

e) Correct/add the missing implementation to the TqsStack.

f) Run the unit tests and confirm that all tests are passing.

g) Your IDE should be able to provide information on coverage.

Look for it and note it down. Experiment disabling some tests (with @Disabled annotation) and verify the impact on coverage metrics.

h) Rename the testing class you have implemented so it's not executed (e.g., rename it from *xxxTest*.java to *_xxxTest*.java__). Generate a new suite of unit tests, but now using AI (such as Github Copilot,...) and compare the tests with your own previous implementation. Run the tests, check the coverage information, and compare with the previous one.

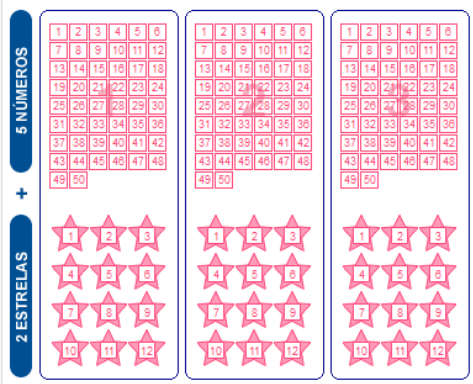
i) Add a method **popTopN(int n)** to the TqsStack that returns the n^{th} item, discarding the n-1 top elements, inspired in the implementation snippet provided. Add also one test to cover it, so that coverage level stays the same (aim at 100% method and statement coverage).

```
public T popTopN(int n) {  1 usage   new *
    T top = null;
    for (int i = 0; i < n; i++) {
        top = collection.removeFirst();
    }
    return top;
}
```

j) Considering the previous point, can you think of a scenario where the TqsStack will fail despite the high coverage level? To which extent can one rely on code coverage to assess quality of your code?

1.2 EuroMillions

Let us consider the "Euromilhões" use case.

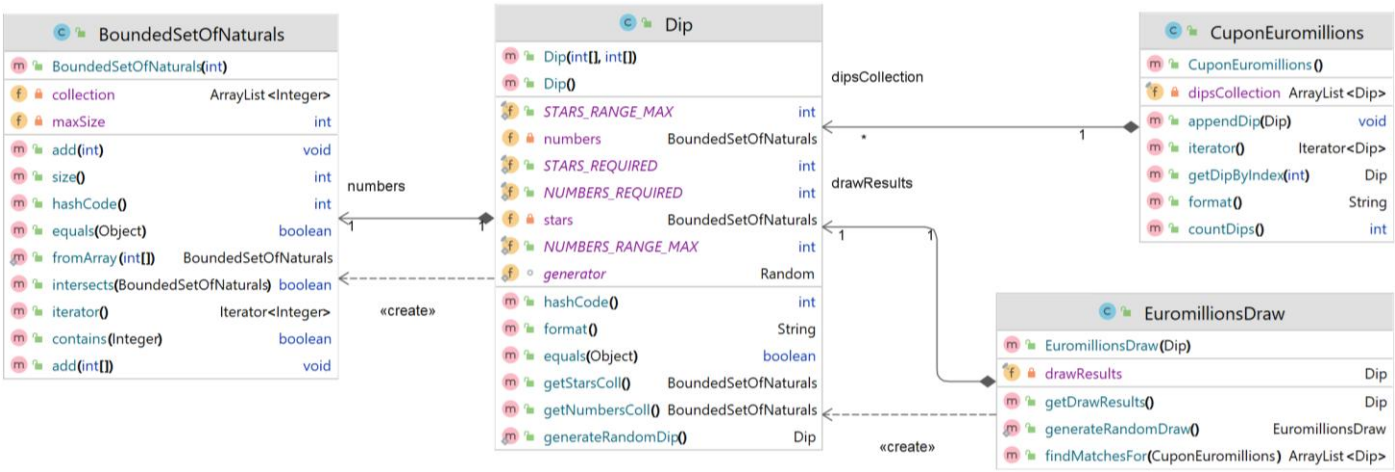


a) Pull the "gs-euromillions" project

The supporting implementation is visualized in the class. Get familiar with the solution and existing tests.

Class	**Purpose**
BoundedSetOfNaturals	Reusable set data structure A set has not duplicates Only natural numbers in the range [1, +∞]. 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 matches for a given player coupon.



b) 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 a test with the @Disabled tag

c) **Assess the coverage** level in project "Euromillions-play" using Jacoco

Configure the maven project to run Jacoco analysis. Use the up to date version of the plugin.

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 and VS Code have 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.

d) Consider the class BoundedSetOfNaturals and its expected contract.

Are there more 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.)

e)

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

f)

Add a rule in Jacoco (i.e., in your pom.xml file) to assure a minimum of 90% line coverage for every class, except test classes. Run the maven “test” goal and then “jacoco:check” goal to make sure it fails (if you need to set a higher line coverage goal, please set accordingly; also, you may disable some tests for the purpose of making the rule fail the build, for now).

```
$ mvn clean test jacoco:check
```

g)

Improve the existing coverage levels using the coverage information (e.g., from your IDE and from Jacoco), to meet the previous line coverage goal.

Troubleshooting some frequent errors

Problem/Symptom	**Solution**
Tests run from the IDE but not from command line	Be sure to configure the Surefire plug-in in Maven to a recent version (example).
My project’s pom.xml is a mess! It is too long and uses old artifacts…	Check this POM and adapt (loggers are optional). Delete everything else.
My project build gives these warnings: [WARNING] Classes in bundle 'euromillions-play' do not match with execution data. For report generation the same class files must be used as at runtime. [WARNING] Execution data for class tqs/euromillions/EuromillionsDraw does not match.	Clean the project, compile and run the tests once again (e.g., “mvn clean test”)