Software Testing

Imane Fouad, UM6P

Warming up: Your First Unit Test

Objective

This step-by-step practical guide helps you write and run your first JUnit 5 unit test in Java using IntelliJ IDEA. While following the steps you will:

- Create a simple `Calculator` class.
- Write a unit test for the `add` (sum) method.
- Learn key assertions and test lifecycle annotations.
- Run the test in IntelliJ and interpret results.

Prerequisites

- IntelliJ IDEA (Community or Ultimate) installed.
- JDK 11 or newer configured in IntelliJ.

Step 1 — Create project

Open IntelliJ and create a new Java project with Maven as a build system (Figure 1).

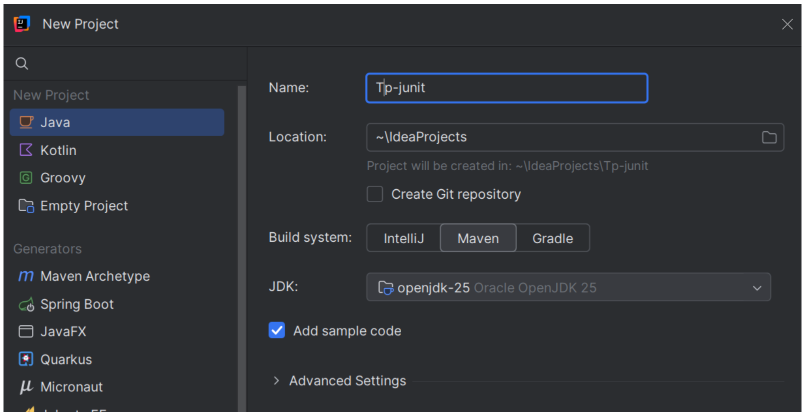

Figure 1: Step 1 — Create project

Ensure you have this structure:

```
project-root/
+-- src/
|   +-- main/
|   |   +-- java/
|   +-- test/
|       +-- java/
```

Step 2 — Create the Calculator class

1. Create a new Java Class.
2. Name it `Calculator`.

Paste this code in `Calculator.java`:

```
public class Calculator {
/**
* Returns the sum of two integers.
*/
public int add(int a, int b) {
return a + b;
}
}
```

Step 3 — Create the test class

1. Open `Calculator.java` in the editor.
2. Right-click inside the editor -> `Generate` (or press `Alt+Insert`) -> `Test...`.
3. In the dialog choose `JUnit5 (Jupiter)` and select the `add` method to generate a test stub.
4. If IntelliJ asks to create a test directory, click `Yes`. IntelliJ will create `src/test/java` and place the generated test there.

Step 4 — Explore the generated test and add assertions

The generated test class contains a skeleton test method. Now, let's add an assertion to verify that `add` works correctly. Replace or edit the test method as follows:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;


class CalculatorTest {


@Test
void add_twoPositiveNumbers_shouldReturnSum() {
// Arrange
Calculator calc = new Calculator();
// Act
int result = calc.add(2, 3);
// Assert
assertEquals(5, result, "2 + 3 should equal 5");
}
}
```

Explanation of the test code:

- `@Test`: marks the method as a test that JUnit will execute.

- Arrange: preparing objects or inputs needed for the test.
- Act: invoking the method under test.
- Assert: checking the actual result against the expected result.

Common Assertions:

Method	Role
`assertEquals(Object a, Object b)`	Verifies that `a` and `b` are equal.
`assertSame(Object a, Object b)`	Verifies that `a` and `b` refer to the same object.
`assertNotSame(Object a, Object b)`	Verifies that `a` and `b` do not refer to the same object.
`assertNull(Object o)`	Verifies that `o` is null.
`assertNotNull(Object o)`	Verifies that `o` is not null.
`assertTrue(boolean e)`	Verifies that `e` is true.
`assertFalse(boolean e)`	Verifies that `e` is false.
`assertThrows(Exception.class, () -> ...)`	Verifies that an exception is thrown.

Table 1: Common JUnit Assertion Methods

Step 5 — Run the test in IntelliJ

1. Right-click `CalculatorTest.java` or the test method.
2. Select `Run 'CalculatorTest'`.
3. Observe the Run tool window: green = passed, red = failed.

Step 6 — Small variations to try

- Temporarily modify the implementation of the `add` method by replacing:

  ```
  return a + b;
  ```

  with:

  ```
  return a - b;
  ```

- Run the test again and observe how it now fails, showing how unit tests detect regressions.

Step 7 — Extend the Calculator

Extend the class `Calculator` with the following methods:

- `subtract(a, b)`
- `multiply(a, b)`
- `divide(a, b)`

Write a unit test for each method using the Arrange–Act–Assert structure.

Step 8 — Exception Handling (Divide by Zero)

Modify the `divide` method so that it throws an exception when dividing by zero (You can refer to Table **??** in the Appendix) .

Write a test to verify this behavior (Check Table 1).

Exercise 2:

In this exercise, you will test the behavior of a small module used in industrial temperature control systems. Your goal is to apply **Boundary Value Analysis (BVA)** to detect a subtle defect in the implementation.

Context

A regulator receives two parameters: **currentTemperature** (in °C), and **targetTemperature** (in °C).

It must decide whether the system should: `HEAT`, `COOL`, or `STANDBY`

Specification

The regulator should behave as follows:

- If `current < target - 0.5` then the action is **HEAT**.
- If `current > target + 0.5` then the action is **COOL**.
- Otherwise, the action is **STANDBY**.

This defines a tolerance zone of $\pm 0.5°C$ around the target temperature.

The following implementation is used by the regulator.

```java
public class TemperatureRegulator {

    public enum Action { HEAT, COOL, STANDBY }

    public Action compute(double current, double target) {
        final double TOL = 0.5;

        double diff = current - target;

        if (diff < -TOL) {
            return Action.HEAT;
        } else if (diff > TOL) {
            return Action.COOL;
        } else {
            return Action.STANDBY;
        }
    }
}
```

> **Task 1** : Build the test cases.
>
> **Task 2** : Implement the test using JUnit and analyze the results to determine whether the class behaves according to the specification.

Exercise 3

A quadratic polynomial is a function of the form:

$$P(X) = aX^2 + bX + c$$

where a, b, and c are constants and X is the variable.

For example, the polynomial $X^2 + X - 2$ has $a = 1$, $b = 1$, and $c = -2$.

A root of the polynomial $P(X)$ is a real number x_1 such that:

$$ax_1^2 + bx_1 + c = 0$$

For instance, 1 is a root of $X^2 + X - 2$ because:

$$1^2 + 1 - 2 = 0$$

> **Task 1:** Write a Java class `PolynomeSecondDegre` to solve a quadratic equation and calculate its real roots.
>
> $$P(X) = aX^2 + bX + c$$

Hint: *Mathematically, the roots are given by the discriminant:*

$$\Delta = b^2 - 4ac$$

- *If* $\Delta > 0$*, there are 2 real roots:* $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$
- *If* $\Delta = 0$*, there is 1 real root:* $x = \frac{-b}{2a}$
- *If* $\Delta < 0$*, there are no real roots.*

> **Task 2:** Identify test cases for the polynomial class
>
> **Task 3:** Write the Test Code