
# Testing Java applications using JUnit framework

- Niket Nagwekar

# Agenda

1)Why do testing?

2)Different types of testing.

3)Why use a testing framework?

4)What is JUnit5 Framework & Architecture ?

5)Creating a Unit Test

6)Assertions

7)JUnit conventions

8)JUnit API

9)Annotation Types

10)Testing Exceptions

11)Performance Concerns: Using JUnit for performance evaluation

12)Parameterized Tests

13)Grouping tests using Suites

14)Good Practices in Unit Testing

15)Other testing frameworks which go hand in hand with Junit.

# Why do testing?

.Steps:

1. Design
2. Write code
3. Test

Testing Process:

1. Design
2. Write Code
3. Write Tests
4. Run Tests.

# Why write tests?

## Tag Line:

The point of writing automated tests is not so much to verify that the code works now.

The point is to verify that on a ongoing basis that the code continues to work in the future as well.

# Different types of testing

1. Alpha Testing - End of developement but before Beta Testing
2. Acceptance Testing - Client Testing or UAT
3. Beta Testing
4. Accessibility Testing
5. Ad-hoc Testing
6. Boundary Value Testing
7. Compatibility Testing
8. End-to-End Testing
9. Functional Testing
10. Gorilla Testing
11. Integration Testing
12. Monkey Testing
13. Unit Testing-Testing of an individual software component or module
14. Component Testing
15. Snapshot Testing - Jest

# Why use a testing framework?

```
public class Calculator{
    public int add(int a , int b){
        return a+b;
    }
}
```

```
Calculator calc = new Calculator();
int sum = calc.add(0,1);

if (sum!=1){
System.out.println("Test failed");
}
```

```
Calculator calc = new Calculator();
int sum = calc.add(-1,3);

if (sum!=2){
System.out.println("Test failed");
}
```

# Test Runs without a framework

Preparation

Provide test inputs

Run the tests

Provide expected output

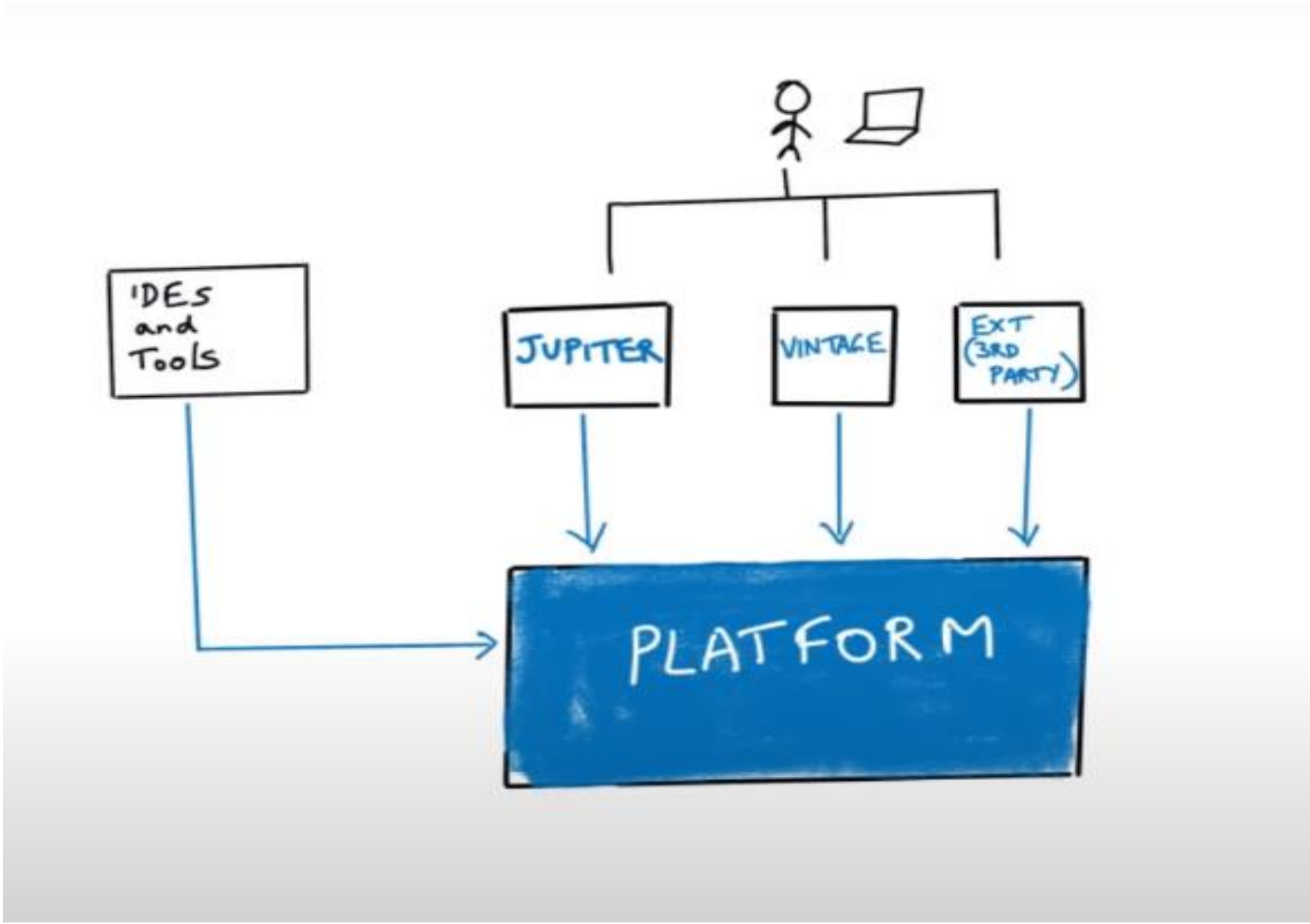
Verify Result

Do something to alert the developer that the test failed

# Why JUnit5?

1. JUnit4 is > 10 years old
2. Not up to date with the new testing patters.
3. Not up to date with the new Java features.
4. Monolithic architecture.
5. Bugs & features piled up.
6. Crowdfunding campaign took place :"JUnit Lambda"
7. Many companies and indivituals contributed to it .
8. Started the path to JUnit 5.
9. JUnit% != JUnit4 +1

# JUnit 5 Architecture






MORNINGSTAR®

# Assert: Used to verify conditions

Definition of assert :

state a fact or belief confidently and forcefully..

assert Keyword in Java:
assert is a Java keyword used to define an assert statement. An assert statement is used to declare an expected boolean condition in a program. If the program is running with assertions enabled, then the condition is checked at runtime. If the condition is false, the Java runtime system throws an AssertionError .

Assertions are done in JUnit 5 with help of
import static org.junit.jupiter.api.Assertions class which has many static methods to do so.

# How JUnit internally works?

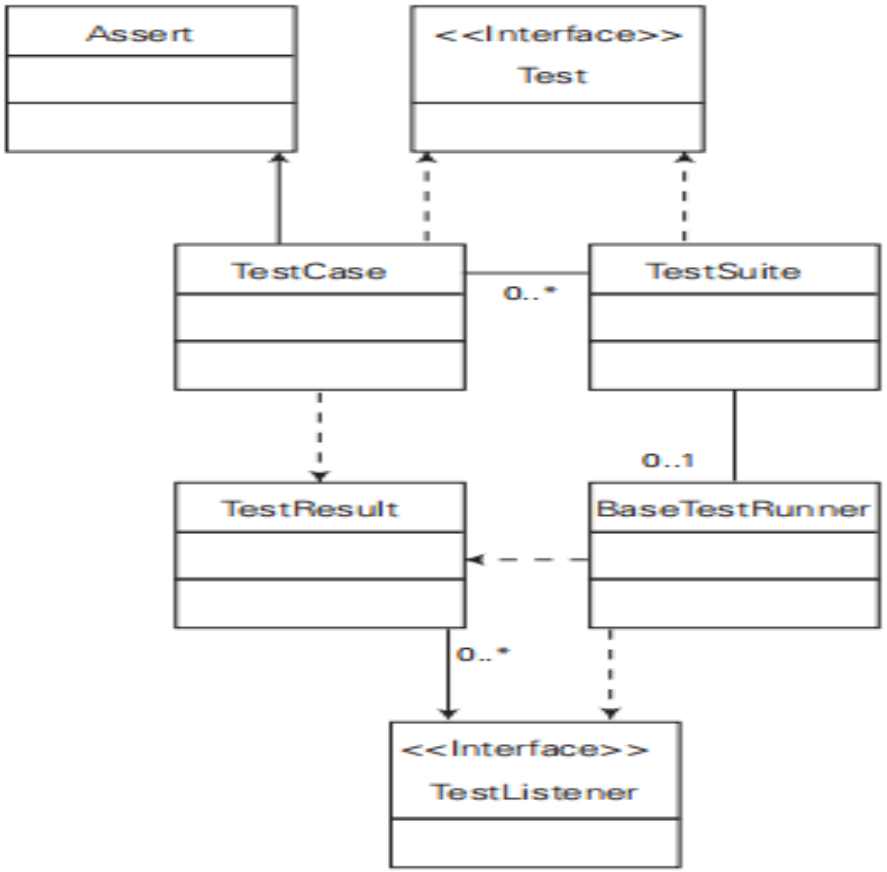



**Figure 2.2 The core JUnit classes used to run any JUnit test program**

7 core clasess & interfaces:

1. TestCase
2. TestSuite
3. BaseTestRunner - launcher of test suites
4. Assert
5. TestResult
6. Test
7. TestListener


**Figure 2.1 The members of the JUnit trio work together to render a test result.**



MORNINGSTAR®

# How JUnit internally works?(Contd)

Keep the bar green to keep the code clean is the JUnit motto.

TestRunner:

Unlike other elements of the JUnit framework, there is no TestRunner interface.

Instead, the various test runners bundled with JUnit all extend BaseTestRunner.

If you needed to write your own test runner for any reason, you could also extend this class yourself. For example, the Cactus framework that extends BaseTestRunner to create a ServletTestRunner that can run JUnit tests from a browser.

# JUnit API: Available test methods

| Test method | Description |
|---|---|
| **assertArrayEquals**(Object[] expecteds, Object[] actuals) | Asserts that two object arrays are equal. |
| **assertEquals**(double expected, double actual, double delta) | Asserts that two doubles or floats are equal to within a positive delta. The “delta” can be omitted. |
| **assertFalse**(String message, boolean condition) | Asserts that a condition is false. The “message” can be omitted. |
| **assertNotNull**(Object object) | Asserts that an object isn't null. |
| **assertNotSame**(String message, Object unexpected, Object actual) | Asserts that two objects do not refer to the same object. The String “message” can be omitted. |
| **assertSame**(String message, Object expected, Object actual) | Asserts that two objects refer to the same object. String “message” can be omitted. |
| **fail**(String message) | Fails a test with the given message. |

# Test Suite:

Between the TestCase and the TestRunner, it would seem that you need sometype of container that can collect several tests together and run them as a set. But, by making it easier to run multiple cases, you don't want to make it harder to run a single test case.

JUnit's answer to this puzzle is the TestSuite. The TestSuite is designed to run one or more test cases. The test runner launches the TestSuite; which test casesto run is up to the TestSuite.

A project can have a huge number of test case classes.

Some of the tests can be logically grouped.

Test Suites are a way of grouping tests that follows a common logic:

- Divide tests taking in account their specificity. Allows a logic separation.
- Detach heavier tests from others. Allows the execution of the test environment restricted to lightest tests, or heaviest ones.

Instead of executing all tests, is possible to execute only the tests included in a Test Suite.

If a group of tests are not so important to execute among others, is possible to create a Test Suite for them and exclude them from test execution environment.

# Test-Driven Development

Unit Tests may be written very early. In fact, they may even be written before any production code exists:

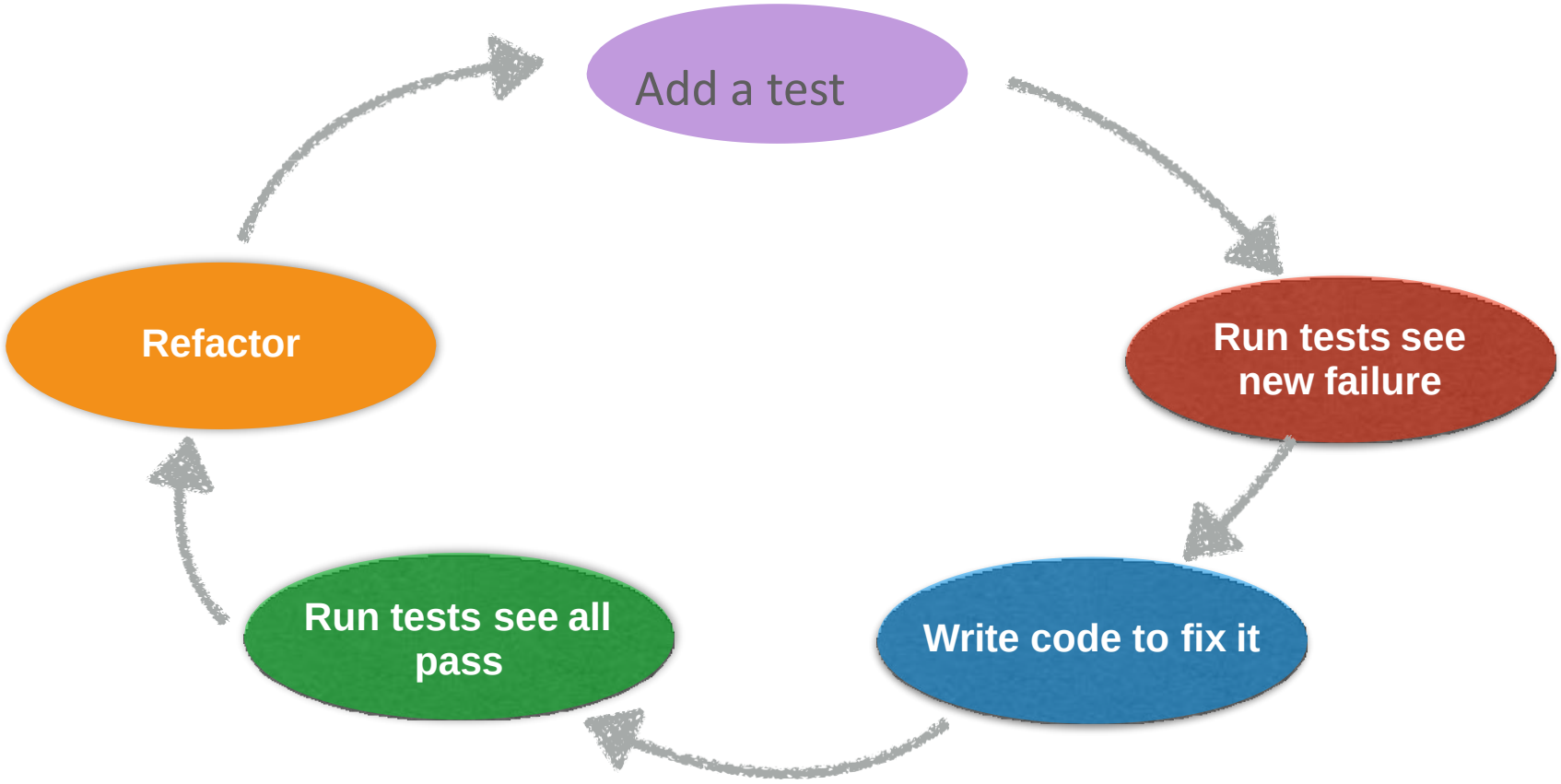
- Write a test that specifies a tiny bit of functionality
- Ensure the test fails (you haven't built the functionality yet!)
- Write the code necessary to make the test pass
- Refactoring the code to remove redundancy

There is a certain rhythm to it: Design a little – test a little – code a little – design a little – test a little – code a little – .

# TDD process

1. **Think** about **what** you want to do.
2. **Think** about how to **test** it.
3. Write a small test. **Think** about the desired **API**.
4. **Write** just enough code to fail the test.
5. **Run** and watch the test **fail** (and you'll get the "**Red Bar**").
6. Write just enough code to **pass the test** (and pass all your previous tests).
7. **Run** and watch all of the tests pass (and you'll get the "**Green Bar**").
8. If you have any duplicate logic, or inexpressive code, **refactor** to remove duplication and increase expressiveness.
9. **Run** the tests **again** (you should still have the "Green Bar").
10. Repeat the steps above until you can't find any more tests that drive writing new code.

# TDD process


Add a test
Run tests see new failure
Write code to fix it
Run tests see all pass
Refactor

# JUnit conventions

- The annotation *@Test* specifies a test method. The name of test methods should have the same name of original tested method preceded by “test” and followed by an “_” to specify the condition that is being made within a test.
- The source code is not mixed with test code:
  - In **/src/main/java** you maintain your code to be tested
  - In **/src/test/java** you maintain the created tests
  - This will allow to compile the JAR with source code only, without unnecessary code used for testing purposes.
- Use inline expressions in test methods arguments (assert methods of JUnit API for example) instead of passing variables.
- Each test method should be *public void.*
- Do not use multiple *assert* or other test methods within a test.

# Using JUnit for Performance evaluation

1. In most different scenarios, is important to know if certain methods runs in less than a certain period of time.

2. In JUnit this can be done using ***timeout*** *as a performance metric* parameter for @Test.

3. The tests created to evaluate the performance of certain methods do not evaluate other conditions simultaneously.

# Parameterized Tests: The Big Picture

```java
package com.myJunitTutorial;

import static org.junit.Assert.*;

public class wordOpTest {

    WordOp word = new WordOp();


    /* The following tests should be Parameterized!! */
    @Test
    public void testWordCountInAString_NullString() {
        assertEquals("0",word.wordCountInAString("null"));
    }

    @Test
    public void testWordCountInAString_Space() {
        assertEquals("0",word.wordCountInAString(" "));
    }

    @Test
    public void testWordCountInAString_Numbers() {
        assertEquals("1",word.wordCountInAString("42343"));
    }

    @Test
    public void testWordCountInAString_TwoWords() {
        assertEquals("2",word.wordCountInAString(" Hello World"));
    }

    // (....)
```

```java
package com.myJunitTutorial;

import static org.junit.Assert.*;

import java.util.Arrays;
import org.junit.Test;

// Step 1
import org.junit.runner.RunWith;
import java.util.Collection;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class wordOpParameterizedTest {

    WordOp word = new WordOp();

    // Step 3
    private String inputs;
    private String outputs;

    public wordOpParameterizedTest(String inputs, String outputs) {
        this.inputs = inputs;
        this.outputs = outputs;
    }

    //Step 2 - Defining the parameters.
    @Parameters
    public static Collection<String[]> parametersForAssertEqualsTest() {
        String arrayOfInputOutput[][] = { {"null","0"},
                                          {" ","0"},
                                          {"42343","1"},
                                          {" Hello World","2"}
                                        };
    return Arrays.asList(arrayOfInputOutput);
    }

    // Step 4
    // Now we have only a single test with all necessary parameters for testing
    @Test
    public void testWordCountInAString() {
        assertEquals(outputs,word.wordCountInAString(inputs));
    }
```

# How to correctly test Exceptions

- When an exception test is made, is supposed that the execution of the test actually throws the expected Exception so the test can ***succeed***.
  Otherwise, it will ***fail***.

```java
@Test(expected=NullPointerException.class)
public void testSortingNames_ArrayNull() {
    String[] names = {};
    Arrays.sort(names);
}
```

**This fails**

```java
@Test(expected=NullPointerException.class)
public void testSortingNames_ArrayNull() {
    String[] names = null;
    Arrays.sort(names);
}
```

**This succeeds**

# Good Practices in Unit Testing

1. One *@Test* should have only one purpose (one condition per test). Do not test multiple aspects in the same test.
2. Follow the conventions about test class naming;
3. Always separate your source code from testing code.
4. Make sure that a test only fails when there is a failure. Abstract the dependencies from external sources. If a test fails because of an external source changed his behavior, the test becames not useful in fact.
5. Cover all possible scenarios in each test to assure the correctness of the methods.
6. When using JUnit for performance evaluation, take in consideration the platforms where the tests will run for choosing ***timeout*** wisely.

# What is Test Coverage?

Test coverage is defined as a metric in Software Testing that measures the amount of testing performed by a set of test. It will include gathering information about which parts of a program are executed when running the test suite to determine which branches of conditional statements have been taken.

***"What are we testing and How much are we testing?"***

Test coverage helps monitor the quality of testing, and assists testers to create tests that cover areas that are missing or not validated.

It is an important metrics to maintain a standard quality of QA efforts.

Enlisted below is the list of the most popular Code Coverage Tools that are available in the market:

1. Parasoft JTest
2. Cobertura
3. Eclemma
4. JaCoCo
5. CodeCover
6. JCov

# JUnit & Mockito

1. Mockito (or any other mocking tool) is a framework that you specifically use to efficiently write certain kind of tests.

2. One core aspect in unit testing is the fact that you want to isolate your "class under test" from anything else in the world. In order to do that, you very often have to create "test doubles" that you provide to an object of your "class under test". You could create all those "test doubles" manually; or you use a mocking framework that generates object of a certain class for you using reflection techniques. Interestingly enough, some people advocate to never use mocking frameworks; but honestly: I can't imagine doing that.

3. In other words: you can definitely use JUnit without using a mocking framework. Same is true for the reverse direction; but in reality, there are not many good reasons why you would want to use Mockito for anything else but unit testing.

# References


1. Jeff Langr, Andy Hunt, and Dave Thomas. 2015. *Pragmatic Unit Testing in Java 8 with Junit* (1st ed.). Pragmatic Bookshelf.
2. Andrew Patterson, Michael Kölling, and John Rosenberg. 2003. Introducing Unit testing with BlueJ. *8th annual conference on Innovation and technology in computer science education* (ITiCSE '03)
3. http://junit.org/junit4/faq.html
4. https://junit.org/junit5/docs/current/api/index.html
5. https://junit.org/junit5/docs/current/user-guide/

MORNINGSTAR®

# Thank you ☺