A picture containing tableware, dishware

Description automatically generated



JUnit and Mockito

Brandon Rocha Díaz.

Software developer.

Shape, rectangle

Description automatically generated

**JUnit**

The JUnit Platform serves as a foundation for launching testing frameworks on the JVM. It also defines the TestEngine API for developing a testing framework that runs on the platform. Furthermore, the platform provides a Console Launcher to launch the platform from the command line and a JUnit 4 based Runner for running any TestEngine on the platform in a JUnit 4 based environment.

JUnit Jupiter is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.

JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.

All core annotations are located in the [org.junit.jupiter.api](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/package-summary.html) package in the junit-jupiter-api module. You can consult the main annotations in here:

[JUnit 5 User Guide](https://junit.org/junit5/docs/current/user-guide/#overview-getting-started)

*DIFFERENCE BETWEEN JUNIT-JUPITER-API AND JUNIT-JUPITER-ENGINE*

* junit-jupiter-api

JUnit Jupiter API for writing tests and extensions

* junit-jupiter-engine

JUnit Jupiter test engine implementation, only required at runtime

* junit-vintage-engine

JUnit Vintage test engine implementation that allows to run vintage JUnit tests, i.e. tests written in the JUnit 3 or JUnit 4 style, on the new JUnit Platform.

You need both junit-jupiter-api and junit-jupiter-engine to write and run JUnit5 tests.

You only need junit-vintage-engine if (a) you are running with JUnit5 **and** (b) your test cases use JUnit4 constructs/annotations/rules etc.

*CONFIGURATION PARAMETERS*

In addition to instructing the platform which test classes and test engines to include, which packages to scan, etc., it is sometimes necessary to provide additional custom configuration parameters that are specific to a particular test engine, listener, or registered extension. For example, the JUnit Jupiter TestEngine supports configuration parameters for the following use cases.

* [Changing the Default Test Instance Lifecycle](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle-changing-default)
* [Enabling Automatic Extension Detection](https://junit.org/junit5/docs/current/user-guide/#extensions-registration-automatic-enabling)
* [Deactivating Conditions](https://junit.org/junit5/docs/current/user-guide/#extensions-conditions-deactivation)
* [Setting the Default Display Name Generator](https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-name-generator-default)

Configuration Parameters are text-based key-value pairs that can be supplied to test engines running on the JUnit Platform via one of the following mechanisms.

* The configurationParameter() and configurationParameters() methods in the LauncherDiscoveryRequestBuilder which is used to build a request supplied to the [Launcher API](https://junit.org/junit5/docs/current/user-guide/#launcher-api). When running tests via one of the tools provided by the JUnit Platform you can specify configuration parameters as follows:
* [Console Launcher](https://junit.org/junit5/docs/current/user-guide/#running-tests-console-launcher): use the --config command-line option.
* [Gradle](https://junit.org/junit5/docs/current/user-guide/#running-tests-build-gradle-config-params): use the systemProperty or systemProperties DSL.
* [Maven Surefire provider](https://junit.org/junit5/docs/current/user-guide/#running-tests-build-maven-config-params): use the configurationParameters property.
* JVM system properties.
* The JUnit Platform configuration file: a file named junit-platform.properties in the root of the class path that follows the syntax rules for a Java Properties file.

Configuration parameters are looked up in the exact order defined above. Consequently, configuration parameters supplied directly to the Launcher take precedence over those supplied via system properties and the configuration file. Similarly, configuration parameters supplied via system properties take precedence over those supplied via the configuration file.

*ANNOTATIONS*

JUnit Jupiter annotations can be used as meta-annotations. That means that you can define your own composed annotation that will automatically inherit the semantics of its meta-annotations.

For example, instead of copying and pasting @Tag("fast") throughout your code base (see [Tagging and Filtering](https://junit.org/junit5/docs/current/user-guide/#writing-tests-tagging-and-filtering)), you can create a custom composed annotation named @Fast as follows. @Fast can then be used as a drop-in replacement for @Tag("fast").

import java.lang.annotation.ElementType; import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy; import java.lang.annotation.Target;

import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD }) @Retention(RetentionPolicy.RUNTIME) @Tag("fast") public @interface Fast { }

The following @Test method demonstrates usage of the @Fast annotation.

@Fast

@Test

void myFastTest() { // ... }

*TEST CLASSES AND METHODS*

**Test Class**: any top-level class, static member class, or [@Nested class](https://junit.org/junit5/docs/current/user-guide/#writing-tests-nested) that contains at least one test method.

Test classes must not be abstract and must have a single constructor.

**Test Method**: any instance method that is directly annotated or meta-annotated with @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, or @TestTemplate.

**Lifecycle Method**: any method that is directly annotated or meta-annotated with @BeforeAll, @AfterAll, @BeforeEach, or @AfterEach.

Test methods and lifecycle methods may be declared locally within the current test class, inherited from superclasses, or inherited from interfaces (see [Test Interfaces and Default Methods](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-interfaces-and-default-methods)). In addition, test methods and lifecycle methods must not be abstract and must not return a value (except @TestFactory methods which are required to return a value).

*CLASS AND METHOD VISIBILITY*

Test classes, test methods, and lifecycle methods are not required to be public, but they must not be private.

It is generally recommended to omit the public modifier for test classes, test methods, and lifecycle methods unless there is a technical reason for doing so – for example, when a test class is extended by a test class in another package. Another technical reason for making classes and methods public is to simplify testing on the module path when using the Java Module System.

*ASSERTIONS*

JUnit Jupiter comes with many of the assertion methods that JUnit 4 has and adds a few that lend themselves well to being used with Java 8 lambdas. All JUnit Jupiter assertions are static methods in the [org.junit.jupiter.api.Assertions](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html) class.

Even though the assertion facilities provided by JUnit Jupiter are sufficient for many testing scenarios, there are times when more power and additional functionality such as matchers are desired or required. In such cases, the JUnit team recommends the use of third-party assertion libraries such as [AssertJ](https://joel-costigliola.github.io/assertj/), [Hamcrest](https://hamcrest.org/JavaHamcrest/), [Truth](https://truth.dev/), etc. Developers are therefore free to use the assertion library of their choice.

For example, the combination of matchers and a fluent API can be used to make assertions more descriptive and readable. However, JUnit Jupiter’s [org.junit.jupiter.api.Assertions](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html) class does not provide an [assertThat()](https://junit.org/junit4/javadoc/latest/org/junit/Assert.html#assertThat) method like the one found in JUnit 4’s org.junit.Assert class which accepts a Hamcrest [Matcher](https://junit.org/junit4/javadoc/latest/org/hamcrest/Matcher.html). Instead, developers are encouraged to use the built-in support for matchers provided by third-party assertion libraries.

You can use the assertThat() support from Hamcrest in a JUnit Jupiter test. As long as the Hamcrest library has been added to the classpath, you can statically import methods such as assertThat(), is(), and equalTo() and then use them in tests like in the assertWithHamcrestMatcher() method.

*TEST EXECUTION ORDER*

By default, test classes and methods will be ordered using an algorithm that is deterministic but intentionally nonobvious. This ensures that subsequent runs of a test suite execute test classes and test methods in the same order, thereby allowing for repeatable builds.

*METHOD ORDER*

Although true *unit tests* typically should not rely on the order in which they are executed, there are times when it is necessary to enforce a specific test method execution order — for example, when writing *integration tests* or *functional tests* where the sequence of the tests is important, especially in conjunction with @TestInstance(Lifecycle.PER\_CLASS).

To control the order in which test methods are executed, annotate your test class or test interface with [@TestMethodOrder](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/TestMethodOrder.html) and specify the desired [MethodOrderer](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.html) implementation. You can implement your own custom MethodOrderer or use one of the following built-in MethodOrderer implementations.

* [MethodOrderer.DisplayName](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.DisplayName.html): sorts test methods *alphanumerically* based on their display names (see [display name generation precedence rules](https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-name-generator-precedence-rules))
* [MethodOrderer.MethodName](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.MethodName.html): sorts test methods *alphanumerically* based on their names and formal parameter lists
* [MethodOrderer.OrderAnnotation](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.OrderAnnotation.html): sorts test methods *numerically* based on values specified via the [@Order](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Order.html) annotation
* [MethodOrderer.Random](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.Random.html): orders test methods *pseudo-randomly* and supports configuration of a custom *seed*
* [MethodOrderer.Alphanumeric](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.Alphanumeric.html): sorts test methods *alphanumerically* based on their names and formal parameter lists; deprecated in favor of [MethodOrderer.MethodName](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.MethodName.html), to be removed

You can use the junit.jupiter.testmethod.order.default [configuration parameter](https://junit.org/junit5/docs/current/user-guide/#running-tests-config-params) to specify the fully qualified class name of the [MethodOrderer](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.html) you would like to use by default. Just like for the orderer configured via the [@TestMethodOrder](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/TestMethodOrder.html) annotation, the supplied class has to implement the MethodOrderer interface.

The default orderer will be used for all tests unless the @TestMethodOrder annotation is present on an enclosing test class or test interface.

For example, to use the [MethodOrderer.OrderAnnotation](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.OrderAnnotation.html) method orderer by default, you should set the configuration parameter to the corresponding fully qualified class name (e.g., in src/test/resources/junit-platform.properties)

*CLASS ORDER*

Although test classes typically should not rely on the order in which they are executed, there are times when it is desirable to enforce a specific test class execution order. You may wish to execute test classes in a random order to ensure there are no accidental dependencies between test classes, or you may wish to order test classes to optimize build time as outlined in the following scenarios.

* Run previously failing tests and faster tests first: "fail fast" mode
* With parallel execution enabled, run longer tests first: "shortest test plan execution duration" mode
* Various other use cases

To configure test class execution order globally for the entire test suite, use the junit.jupiter.testclass.order.default [configuration parameter](https://junit.org/junit5/docs/current/user-guide/#running-tests-config-params) to specify the fully qualified class name of the [ClassOrderer](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/ClassOrderer.html) you would like to use. The supplied class must implement the ClassOrderer interface.

You can implement your own custom ClassOrderer or use one of the following built-in ClassOrderer implementations.

* [ClassOrderer.ClassName](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/ClassOrderer.ClassName.html): sorts test classes alphanumerically based on their fully qualified class names
* [ClassOrderer.DisplayName](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/ClassOrderer.DisplayName.html): sorts test classes alphanumerically based on their display names (see [display name generation precedence rules](https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-name-generator-precedence-rules))
* [ClassOrderer.OrderAnnotation](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/ClassOrderer.OrderAnnotation.html): sorts test classes numerically based on values specified via the [@Order](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Order.html) annotation
* [ClassOrderer.Random](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/ClassOrderer.Random.html): orders test classes pseudo-randomly and supports configuration of a custom seed

For example, for the @Order annotation to be honored on test classes, you should configure the [ClassOrderer.OrderAnnotation](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/ClassOrderer.OrderAnnotation.html) class orderer using the configuration parameter with the corresponding fully qualified class name (e.g., in src/test/resources/junit-platform.properties)

The configured ClassOrderer will be applied to all top-level test classes (including static nested test classes) and @Nested test classes.

Top-level test classes will be ordered relative to each other; whereas, @Nested test classes will be ordered relative to other @Nested test classes sharing the same enclosing class.

To configure test class execution order locally for @Nested test classes, declare the [@TestClassOrder](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/TestClassOrder.html) annotation on the enclosing class for the @Nested test classes you want to order, and supply a class reference to the ClassOrderer implementation you would like to use directly in the @TestClassOrder annotation. The configured ClassOrderer will be applied recursively to @Nested test classes and their @Nested test classes. Note that a local @TestClassOrder declaration always overrides an inherited @TestClassOrder declaration or a ClassOrderer configured globally via the junit.jupiter.testclass.order.default configuration parameter.

The following example demonstrates how to guarantee that @Nested test classes are executed in the order specified via the @Order annotation.

*TEST INSTANCE LIFECYCLE*

In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each test method (see [Test Classes and Methods](https://junit.org/junit5/docs/current/user-guide/#writing-tests-classes-and-methods)). This "per-method" test instance lifecycle is the default behavior in JUnit Jupiter and is analogous to all previous versions of JUnit.

Please note that the test class will still be instantiated if a given test method is disabled via a [condition](https://junit.org/junit5/docs/current/user-guide/#writing-tests-conditional-execution) (e.g., @Disabled, @DisabledOnOs, etc.) even when the "per-method" test instance lifecycle mode is active.

If you would prefer that JUnit Jupiter execute all test methods on the same test instance, annotate your test class with @TestInstance(Lifecycle.PER\_CLASS). When using this mode, a new test instance will be created once per test class. Thus, if your test methods rely on state stored in instance variables, you may need to reset that state in @BeforeEach or @AfterEach methods.

The "per-class" mode has some additional benefits over the default "per-method" mode. Specifically, with the "per-class" mode it becomes possible to declare @BeforeAll and @AfterAll on non-static methods as well as on interface default methods. The "per-class" mode therefore also makes it possible to use @BeforeAll and @AfterAll methods in @Nested test classes.

If you are authoring tests using the Kotlin programming language, you may also find it easier to implement @BeforeAll and @AfterAll methods by switching to the "per-class" test instance lifecycle mode.

*DYNAMIC TESTS*

Assumptions provide a basic form of dynamic behavior but are intentionally rather limited in their expressiveness.

In addition to these standard tests a completely new kind of test programming model has been introduced in JUnit Jupiter. This new kind of test is a dynamic test which is generated at runtime by a factory method that is annotated with @TestFactory.

In contrast to @Test methods, a @TestFactory method is not itself a test case but rather a factory for test cases. Thus, a dynamic test is the product of a factory. Technically speaking, a @TestFactory method must return a single DynamicNode or a Stream, Collection, Iterable, Iterator, or array of DynamicNode instances. Instantiable subclases of DynamicNode are DynamicContainer and DynamicTest. DynamicContainer instances are composed of a display name and a list of dynamic child nodes, enabling the creation of arbitrarily nested hierarchies of dynamic nodes. DynamicTest instances will be executed lazily, enabling dynamic and even non-deterministic generation of test cases.

Any Stream returned by a @TestFactory will be properly closed by calling stream.close(), making it safe to use a resource such as Files.lines().

As with @Test methods, @TestFactory methods must not be private or static and may optionally declare parameters to be resolved by ParameterResolvers.

A DynamicTest is a test case generated at runtime. It is composed of a display name and an Executable. Executable is a @FunctionalInterface which means that the implementations of dynamic tests can be provided as lambda expressions or method references.

The execution lifecycle of a dynamic test is quite different than it is for a standard @Test case. Specifically, there are no lifecycle callbacks for individual dynamic tests. This means that @BeforeEach and @AfterEach methods and their corresponding extension callbacks are executed for the @TestFactory method but not for each dynamic test. In other words, if you access fields from the test instance within a lambda expression for a dynamic test, those fields will not be reset by callback methods or extensions between the execution of individual dynamic tests generated by the same @TestFactory method.

*PARALLEL EXECUTION*

By default, JUnit Jupiter tests are run sequentially in a single thread. Running tests in parallel — for example, to speed up execution — is available as an opt-in feature since version 5.3.

To enable parallel execution, set the junit.jupiter.execution.parallel.enabled configuration parameter to true — for example, in junit-platform.properties (see [Configuration Parameters](https://junit.org/junit5/docs/current/user-guide/#running-tests-config-params) for other options).

Please note that enabling this property is only the first step required to execute tests in parallel. If enabled, test classes and methods will still be executed sequentially by default. Whether or not a node in the test tree is executed concurrently is controlled by its execution mode. The following two modes are available.

SAME\_THREAD

Force execution in the same thread used by the parent. For example, when used on a test method, the test method will be executed in the same thread as any @BeforeAll or @AfterAll methods of the containing test class.

CONCURRENT

Execute concurrently unless a resource lock forces execution in the same thread.

By default, nodes in the test tree use the SAME\_THREAD execution mode. You can change the default by setting the junit.jupiter.execution.parallel.mode.default configuration parameter. Alternatively, you can use the [@Execution](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/parallel/Execution.html) annotation to change the execution mode for the annotated element and its subelements (if any) which allows you to activate parallel execution for individual test classes, one by one.

This is the configuration parameters that you’ll need to execute all tests in parallel.

junit.jupiter.execution.parallel.enabled = true

junit.jupiter.execution.parallel.mode.default = concurrent

The default execution mode is applied to all nodes of the test tree with a few notable exceptions, namely test classes that use the Lifecycle.PER\_CLASS mode or a [MethodOrderer](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.html) (except for [MethodOrderer.Random](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/MethodOrderer.Random.html)). In the former case, test authors must ensure that the test class is thread-safe; in the latter, concurrent execution might conflict with the configured execution order. Thus, in both cases, test methods in such test classes are only executed concurrently if the @Execution(CONCURRENT) annotation is present on the test class or method.

All nodes of the test tree that are configured with the CONCURRENT execution mode will be executed fully in parallel according to the provided [configuration](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parallel-execution-config) while observing the declarative [synchronization](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parallel-execution-synchronization) mechanism. Please note that [Capturing Standard Output/Error](https://junit.org/junit5/docs/current/user-guide/#running-tests-capturing-output) needs to be enabled separately.

In addition, you can configure the default execution mode for top-level classes by setting the junit.jupiter.execution.parallel.mode.classes.default configuration parameter. By combining both configuration parameters, you can configure classes to run in parallel but their methods in the same thread:

Configuration parameters to execute top-level classes in parallel but methods in same thread

junit.jupiter.execution.parallel.enabled = true

junit.jupiter.execution.parallel.mode.default = same\_thread

junit.jupiter.execution.parallel.mode.classes.default = concurrent

The opposite combination will run all methods within one class in parallel, but top-level classes will run sequentially:

Configuration parameters to execute top-level classes sequentially but their methods in parallel.

junit.jupiter.execution.parallel.enabled = true junit.jupiter.execution.parallel.mode.default = concurrent junit.jupiter.execution.parallel.mode.classes.default = same\_thread

If the junit.jupiter.execution.parallel.mode.classes.default configuration parameter is not explicitly set, the value for junit.jupiter.execution.parallel.mode.default will be used instead.

**JUNIT BEST PRACTICES**

* Source Code

It's a good idea to keep the test classes separate from the main source code. So, they are developed, executed and maintained separately from the production code.

Also, it avoids any possibility of running test code in the production environment.

We can follow the steps of the build tools such as Maven and Gradle that look for src/main/test directory for test implementations.

* Package naming convention

We should create a similar package structure in the src/main/test directory for test classes, this way improving the readability and maintainability of the test code.

Simply put, the package of the test class should match the package of the source class whose unit of source code it'll test.

* Test naming convention

The test names should be insightful, and users should understand the behavior and expectation of the test by just glancing at the name itself.

For example, the name of our unit test was testCalculateArea, which is vague on any meaningful information about the test scenario and expectation.

Therefore, we should name a test with the action and expectation such as testCalculateAreaWithGeneralDoubleValueRadiusThatReturnsAreaInDouble, testCalculateAreaWithLargeDoubleValueRadiusThatReturnsAreaAsInfinity.

However, we can still improve the names for better readability.

It's often helpful to name the test cases in given\_when\_then to elaborate on the purpose of a unit test:

public class CircleTest {

//... @Test

public void givenRadius\_whenCalculateArea\_thenReturnArea() {

//... }

@Test

public void givenDoubleMaxValueAsRadius\_whenCalculateArea\_thenReturnAreaAsInfinity() { //... } }

We should also describe code blocks in the [Given, When and Then](https://www.baeldung.com/cs/bdd-guide) format. In addition, it helps to differentiate the test into three parts: input, action and output.

First, the code block corresponding to the *given* section creates the test objects, mocks the data and arranges input.

Next, the code block for the *when*section represents a specific action or test scenario.

Likewise, the *then* section points out the output of the code, which is verified against the expected result using assertions.

* Expected vs actual

A test case should have an assertion between expected and actual values.

To corroborate the idea of the expected vs actual values, we can look at the definition of the [assertEquals method of JUnit's Assert class](https://www.baeldung.com/junit-assertions#junit4-assertequals):

public static void assertEquals(Object expected, Object actual)

Let's use the assertion in one of our test cases:

@Test

public void givenRadius\_whenCalculateArea\_thenReturnArea() {

double actualArea = Circle.calculateArea(1d);

double expectedArea = 3.141592653589793;

Assert.assertEquals(expectedArea, actualArea);

}

It's suggested to prefix the variable names with the actual and expected keyword to improve the readability of the test code.

* Prefer simple test case

In the previous test case, we can see that the expected value was hard-coded. This is done to avoid rewriting or reusing actual code implementation in the test case to get the expected value.

It's not encouraged to calculate the area of the circle to match against the return value of the calculateArea method:

@Test

public void givenRadius\_whenCalculateArea\_thenReturnArea() {

double actualArea = Circle.calculateArea(2d);

double expectedArea = 3.141592653589793 \* 2 \* 2;

Assert.assertEquals(expectedArea, actualArea);

}

In this assertion, we're calculating both expected and actual values using similar logic, resulting in similar results forever. So, our test case won't have any value added to the unit testing of code.

Therefore, we should create a simple test case that asserts hard-coded expected value against the actual one.

Although it's sometimes required to write the logic in the test case, we shouldn't overdo it. Also, as commonly seen, we should never implement production logic in a test case to pass the assertions.

* Appropiate assertions

Always use proper assertions to verify the expected vs. actual results. We should use various methods available in the Assert class of [JUnit](https://www.baeldung.com/junit) or similar frameworks such as [AssertJ](https://www.baeldung.com/introduction-to-assertj).

For instance, we've already used the Assert.assertEquals method for value assertion. Similarly, we can use assertNotEquals to check if the expected and actual values are not equal.

Other methods such as assertNotNull, assertTrue and assertNotSame are beneficial in distinct assertions.

* Specific Unit Tests

Instead of adding multiple assertions to the same unit test, we should create separate test cases.

Of course, it's sometimes tempting to verify multiple scenarios in the same test, but it's a good idea to keep them separate. Then, in the case of test failures, it'll be easier to determine which specific scenario failed and, likewise, simpler to fix the code.

Therefore, always write a unit test to test a single specific scenario.

A unit test won't get overly complicated to understand. Moreover, it'll be easier to debug and maintain unit tests later.

* Test Production Scenarios

Unit testing is more rewarding when we write tests considering real scenarios in mind.

Principally, it helps to make unit tests more relatable. Also, it proves essential in understanding the behavior of the code in certain production cases.

* Mock External Services

Although unit tests concentrate on specific and smaller pieces of code, there is a chance that the code is dependent on external services for some logic.

Therefore, we should mock the external services and merely test the logic and execution of our code for varying scenarios.

We can use various frameworks such as [Mockito](https://www.baeldung.com/mockito-series), [EasyMock](https://www.baeldung.com/easymock) and [JMockit](https://www.baeldung.com/jmockit-101) for mocking external services.

* Avoid Code Redundancy

Create more and more helper functions to generate the commonly used objects and mock the data or external services for similar unit tests.

As with other recommendations, this enhances the readability and maintainability of the test code.

* Annotations

Often, testing frameworks provide annotations for various purposes, for example, performing setup, executing code before and tearing down after running a test.

Various annotations such as JUnit's [@Before, @BeforeClass and @After](https://www.baeldung.com/junit-before-beforeclass-beforeeach-beforeall) and from other test frameworks such as [TestNG](https://www.baeldung.com/testng) are at our disposal.

We should leverage annotations to prepare the system for tests by creating data, arranging objects and dropping all of it after every test to keep test cases isolated from each other.

* 80% Test coverage

More [test coverage for the source code](https://www.baeldung.com/cs/code-coverage) is always beneficial. However, it's not the only goal to achieve. We should make a well-informed decision and choose a better trade-off that works for our implementation, deadlines and the team.

As a rule of thumb, we should try to cover 80% of the code by unit tests.

Additionally, we can use tools such as [JaCoCo](https://www.baeldung.com/jacoco) and [Cobertura](https://www.baeldung.com/cobertura) along with Maven or Gradle to generate code coverage reports.

* TDD Approach

We can improve the reliability of the code by automating the execution of the entire test suite while creating new builds.

Primarily, this helps to avoid unfortunate regressions in various release environments. It also ensures rapid feedback before a broken code is released.

Therefore, unit test execution should be part of [CI-CD pipelines](https://www.baeldung.com/ops/jenkins-pipelines) and alert the stakeholders in case of malfunctions.

**MOCKITO**

Mockito is a mocking framework that tastes really good. It lets you write beautiful tests with a clean & simple API. Mockito doesn’t give you hangover because the tests are very readable and they produce clean verification errors

Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

Mockito facilitates creating mock objects seamlessly. It uses Java Reflection to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.

Mockito is a [unit testing framework](https://www.lambdatest.com/blog/9-of-the-best-java-testing-frameworks-for-2021/) for Java that simplifies the task of automation testing. It makes unit testing highly effective with clean tests, thanks to dependency injection and compile-time checks. In addition, Mockito helps improve the test independence of components being developed by helping you create objects that have no dependencies on a test-specific configuration. The most popular way of using Mockito is within the JUnit framework to help you write better tests.

For starters, mocking in unit testing is used for isolating the AUT (Application Under Test) from external dependencies. Mocking must be leveraged when implementation of external dependencies is not yet completed.

We should know different types of Test objects that can be created and used in Mockito.

* Dummy: This is an object that is used only for the code to compile and has no business logic associated with it like a parameter passed to a function.
* Fake: This is an object that has an implementation but not ready for production, like H2 in-memory database.
* Stub:  This is an object that has predefined answers to method executions made during the test.
* Mock:  This is an object that has predefined answers to method executions made during the test and has recorded expectations of these executions.
* Spy: This is an object that is similar to stubs, but they additionally record how they were executed.

Consider a case of Stock Service which returns the price details of a stock. During development, the actual stock service cannot be used to get real-time data. So we need a dummy implementation of the stock service. Mockito can do the same very easily, as its name suggests.

Benefits of Mockito

* No Handwriting − No need to write mock objects on your own.
* Refactoring Safe − Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
* Return value support − Supports return values.
* Exception support − Supports exceptions.
* Order check support − Supports check on order of method calls.
* Annotation support − Supports creating mocks using annotation.

Most of the classes we come across have dependencies, and often methods delegates some of the work to other methods in other classes, and we call these classes dependencies. When unit testing such methods, if we only used JUnit, our tests would also depend on those methods as well. We want the unit tests to be independent of all other dependencies.

* eg: we want to test the method addCustomer in CustomerService class, and within this addCustomer method, the save method of the CustomerDao class is invoked. We don’t want to call the real implementation of the CustomerDao save() method for a few reasons:
  + We only want to test the logic inside the addCustomer() in isolation.
  + We may not yet have implemented it.
  + We don’t want the unit test of the addCustomer() fail if there is a defect in save() method in the CustomerDao.
* So we should somehow mock the behavior of the dependencies. This is where mocking frameworks comes in to play.

When you use Mockito in your unit tests, you will need to download the jar file and place it in a path that your build system can find. Mockito is available in two versions: mockito-core (which contains only the core of Mockito, and mockito-all (which contains all modules).

The preferred way of installing Mockito is to declare a dependency on mockito-core with a build system of choice. The second-best way is to download the artifacts using a manual approach and add them to the classpath. You can also add dependencies to your existing Maven or Gradle project.

* Also Read: [Run JUnit Tests From Command Line](https://www.lambdatest.com/blog/run-junit-from-command-line/)
* Add the following dependencies in your pom.xml:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.1.0</version>

<scope>test</scope>

</dependency>

Like [JUnit annotations](https://www.lambdatest.com/blog/tutorial-on-junit-annotations-in-selenium-with-examples/), Mockito annotations are used to specify the behavior of the test code. It allows the users to focus more on their logic while still testing the code very effectively. This section of the JUnit 5 Mockito tutorial mainly focuses on Mockito annotations and how to use them in Selenium.

**There may be occasions when we need to provide additional mock settings during mock creation.** This might be useful when debugging, dealing with legacy code, or covering some corner cases.

**Put simply, the *MockSettings* interface provides a**[**Fluent API**](https://www.baeldung.com/mockito-fluent-apis)**that allows us to easily add and combine additional mock settings during mock creation.**

**When we create a mock object, all our mocks carry a set of default settings. Let's take a look at a simple mock example:**

List mockedList = mock(List.class);

Behind the scenes the Mockito mock method delegates to another overloaded method with a set of default settings for the mock:

public static <T> T mock(Class<T> classToMock) {

return mock(classToMock, withSettings());

}

**Let’s have a look at our default settings:**

public static MockSettings withSettings() {

return new MockSettingsImpl().defaultAnswer(RETURNS\_DEFAULTS);

}

As we can see, our standard set of settings for our mock objects is very simple. We configure the default answer for our mock interactions. Typically, using *RETURNS\_DEFAULTS* will return some empty value.

The important point to take away from this is that we can provide our own set of custom settings to our mock objects if the need arises.

Let's see how we can change the default return value for a mock object.

Let's imagine we have a very simple setup for a mock:

PizzaService service = mock(PizzaService.class);

Pizza pizza = service.orderHouseSpecial();

PizzaSize size = pizza.getSize();

When we run this code as expected, we'll get a *NullPointerException* because our unstubbed method *orderHouseSpecial* returns *null*.

This is OK, but sometimes when working with legacy code, we might need to handle a complicated hierarchy of mock objects, and it can be time-consuming to locate where these types of exceptions occur.

To help us combat this, we can provide a different default answer via our mock settings during mock creation:

PizzaService pizzaService = mock(PizzaService.class,

withSettings().defaultAnswer(RETURNS\_SMART\_NULLS));

By using the RETURNS\_SMART\_NULLS as our default answer, Mockito gives us a much more meaningful error message that shows us exactly where the incorrect stubbing occurred:

org.mockito.exceptions.verification.SmartNullPointerException: You have a NullPointerException here: -> at

com.baeldung.mockito.mocksettings.MockSettingsUnitTest.whenServiceMockedWithSmartNulls\_thenExceptionHasExtraInfo(MockSettingsUnitTest.java:45) because this method call was \*not\* stubbed correctly: -> at com.baeldung.mockito.mocksettings.MockSettingsUnitTest.whenServiceMockedWithSmartNulls\_thenExceptionHasExtraInfo(MockSettingsUnitTest.java:44) pizzaService.orderHouseSpecial();

This can really save us some time when debugging our test code. The Answers enumeration also supplies some other preconfigured mock answers of note:

* RETURNS\_DEEP\_STUBS – an answer that returns [deep stubs](https://www.baeldung.com/mockito-fluent-apis#deep-mocking) – this can be useful when working with Fluent APIs
* RETURNS\_MOCKS – using this answer will return ordinary values such as empty collections or empty strings, and thereafter, it tries to return mocks
* CALLS\_REAL\_METHODS – as the name suggests, when we use this implementation, unstubbed methods will delegate to the real implementation

We can give our mock a name by using the name method of MockSettings. This can be particularly useful for debugging as the name we provide is used in all verification errors:

PizzaService service = mock(PizzaService.class, withSettings() .name("pizzaServiceMock") .verboseLogging() .defaultAnswer(RETURNS\_SMART\_NULLS));

In this example, we combine this naming feature with verbose logging by using the method verboseLogging().

Using this method enables real-time logging to the standard output stream for method invocations on this mock. Likewise, it can be used during test debugging in order to find wrong interactions with a mock.

When we run our test, we'll see some output on the console:

pizzaServiceMock.orderHouseSpecial(); invoked: -> at com.baeldung.mockito.mocksettings.MockSettingsUnitTest.whenServiceMockedWithNameAndVerboseLogging\_thenLogsMethodInvocations(MockSettingsUnitTest.java:36) has returned: "Mock for Pizza, hashCode: 366803687" (com.baeldung.mockito.fluentapi.Pizza$MockitoMock$168951489)

if we're using the @Mock annotation, our mocks automatically take the field name as the mock name.

*POWERMOCK*

PowerMock is an open-source Java framework used for creating a mock object in unit testing. It extends other mocking frameworks such as EasyMock and Mockito to enhance the capabilities. The PowerMock framework uses a custom classloader and bytecode manipulation techniques to enable the mocking of static methods, final classes, final methods, private methods, constructor, and removal of static initializers.

The main aim of PowerMock is to extend the existing APIs with some methods and annotations to provide extra features that make unit testing quite easy.

The PowerMock framework provides a class called PowerMockito used to create mock objects and initiates verification and expectation. The PowerMockito provides the functionality to work with the Java reflection API.

To see more details of implementation visit:

[Mockito PowerMock - Javatpoint](https://www.javatpoint.com/mockito-powermock)

**DIFFERENCES BETWEEN MOCK VS STUB VS SPY**

|  |  |  |
| --- | --- | --- |
| Parameters | Stub | Mock |
| Data Source | The data source of stubs is hardcoded. It is usually tightly coupled to the test suite. | Data on mocks is set up by the tests. |
| Created by | Stubs are usually handwritten, and some are generated by tools. | Mocks are usually created by using the third-party library such as Mockito, JMock, and WireMock. |
| Usage | Stubs are mainly used for simple test suites. | Mocks are mainly used for large test suites. |
| Graphics User Interface (GUI) | Stubs do not have a GUI. | Mocks have a GUI. |

|  |  |  |
| --- | --- | --- |
| Parameters | Mock | Spy |
| Usage | Mocks are used to create fully mock or dummy objects. It is mainly used in large test suites. | Spies are used for creating partial or half mock objects. Like mock, spies are also used in large test suites. |
| Default behavior | When using mock objects, the default behavior of methods (when not stubbed) is do nothing (performs nothing.) | When using spy objects, the default behavior of the methods (when not stubbed) is the real method behavior. |

**BEST PRACTICES WITH MOCKITO**

The common understanding of unit testing is testing the smallest possible part of the software, specifically a method. In reality, we do not test methods; rather, we test a logical unit or the behavior of the system.

* Readability

JUnit tests are written to test logical units. A test method name should portray the intention of the test so that a reader can understand what is being tested, such as the condition and the expectation or action.

Good test method names can be:

should\_not\_register\_a\_null\_user()

should\_throw\_exception\_when\_a\_null\_user\_is\_registered()

* Break Everything

An Extreme Programming concept is test everything that could possibly break. This means trying all different combinations of inputs to make sure we don’t miss any combination that can cause the class to generate an error.

* Ignore Trivial Tests

Writing trivial JUnits (such that for getter and setter) is mostly a waste of time and money. We don’t have the luxury to write infinite tests as it can eat our development time, application build time, and reduce test maintainability. If we start writing tests for getter/setters, we may miss more useful test cases.

* Staying away from debugging

A common practice when we find a bug is to start debugging an application—stop doing this. Rather, add more tests to break the code; this will enrich your test suite and improve the system documentation.

So anyway, before starting to debug, create an (integration) test that reproduces the issue and then debug it. This will narrow down the problem, create a unit test for the lowest possible unit, and keep both the tests for future reference.

**CODE COVERAGE**

Code coverage describes the percentage of code covered by automated tests. In other words, it checks which parts of code run during the test suite and which don’t.

First of all, we can’t say that implementing unit tests gives our application reliable protection from bugs unless we’re using a code coverage approach. Code coverage is a metric we use to quantify how much of our code based is being reached by our unit tests.

Creating unit tests and not knowing for sure if they cover at least the most important scenarios, paths, and edge cases seems to be a profitless idea.

*WAYS TO MEASURE CODE COVERAGE*

* Statement coverage

The main purpose of statement coverage is to run each executable statement in the program’s code at least once. It’s also often called line coverage.

**Percentage of statement coverage can be calculated as:**

**Statement coverage = Number of executed statements / Total number of statements \* 100**

That way of measuring code coverage is able to:

* + verify the do’s and don’ts of the written code
  + find dead code and unused statements
  + test different flow paths and checks which ones are not covered

Let’s analyze a simple example. Below we’re given pseudocode of program that sums two numbers and prints information if the result is less than, greater than or equals zero:

Text

Description automatically generated

Now, let’s talk about possible tests.

In the first option, we have given a = 3, b = 5. Statements at lines 1, 2, 3 and 4 will be executed during this test case. So we’re covering four statements out of eight. That means our statement coverage equals 50%.

Let’s discuss another option with a = 3, b = -5. In this scenario statements at lines 1, 2, 3, 5, 6 will be executed. Both tests altogether call lines 1 – 6. Total code coverage equals (6/8) \* 100 = 75%.

* Branch coverage

Branch coverage ensures if each decision in a decision-making tree is executed at least once. By branches we mean: conditional statements, loops, switch statements. We can calculate branch coverage using the below formula:

**Branch coverage = Number of executed branches / Total number of branches \* 100**

Advantages of branch coverage:

* + verify if the execution of the test suite reaches all branches
  + detects possible abnormal behavior of each branch
  + can test areas of the source code that other approaches may discount

The pseudocode given below represents a program that prints passed numbers and also prints additional comments if the number is even:

Text

Description automatically generated with medium confidenceDiagram

Description automatically generated

Branch coverage takes into consideration both types.

Let’s now explore two test scenarios:

* + Given a = 1. The test case will go only through branch labeled with “No”. Branch coverage: 1/3 \* 100 = 33%.
  + Given a = 4. The test case will go through a branch labeled with “Yes” and the unconditional one. Branch coverage: 2/3 \* 100 = 67%

Clearly, both tests together cover 100% of branches.

* Function coverage

Function coverage verifies if each function of a program is being called at least once. It is also important to test functions with different input parameters. That way, test suites will also check if functions behave properly in different scenarios.

To calculate function coverage, we can use the following formula:

**Function coverage = Number of executed functions / Total number of functions \* 100**

Function coverage is the broadest compared to the rest.

Let’s say, for example, that our application consists only of a single method. Implementing a single unit test for that method will result in 100% function coverage. Obviously, one unit test is not able to cover all paths and scenarios. Despite 100% function coverage, our application is clearly not well-tested.

As we can see a high percentage of code coverage doesn’t always mean that code is ideal and faultless.

There are many different tools for code coverage depending on which language we use, is the tool that we need.