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)

*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/" \l "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/" \l "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" \l "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/" \l "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.