Exploring Core Junit

# Core Concepts

## Test Class

*A test class* may be **a top-level class**, **a static member class**, or **an inner class annotated as @Nested** that contains one or more test methods.

**Test classes cannot be abstract.**

**They must have a single constructor**. **The constructor must have no arguments, or arguments that can be dynamically resolved at runtime through dependency injection.** (We discuss the details of dependency injection in section 2.6.)

A test class is allowed to be package-private as a minimum requirement for visibility. It is no longer required that test classes be public, as was the case up to JUnit 4.x.

## Test Method

*A test method* is an instance method that is annotated with @Test, @Repeated-Test, @ParameterizedTest, @TestFactory, or @TestTemplate.

Test methods must not be abstract **and must not return a value** (the return type should be void).

## Life-Cycle Method

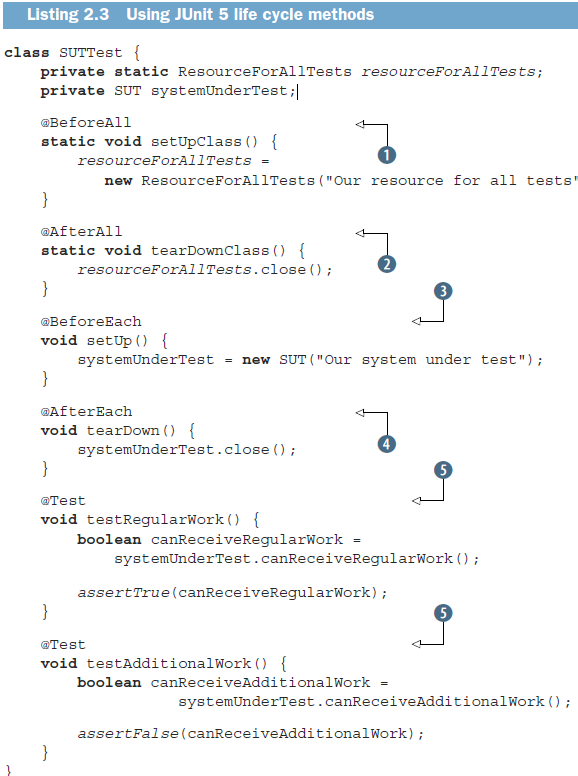
*A life cycle method* is a method that is annotated with @BeforeAll, @AfterAll, @BeforeEach, or @AfterEach.

* @BeforeEach and @AfterEach Methods must have a return type of void and **must not** be static
* @BeforeAll and @AfterAll Methods must have a return type of void and **must** be static if the test instance life cycle is PER\_METHOD(the default behavior). In case of the PER\_CLASS life cycle, they can be non-static. But the point is, they are run just once before and once after all the test methods

## How Everything works together

* JUnit creates **a new instance of the test class** before invoking each @Test method to ensure the independence of test methods and prevent unintentional side effects in the test code.
* it is a universally accepted fact **that the tests must produce the same result independent of the order of their execution.**
* Because each test method runs on a new test class instance, you cannot reuse instance variable values across test methods. One test instance is created for the execution of each test method, which is **the default behavior in JUnit 5 and all previous versions**.
* If you annotate your **test class** with @TestInstance(Lifecycle.PER\_CLASS)as opposed to the default of Lifecycle.PER\_METHOD, JUnit 5 will execute all test methods on the same test instance. A new test instance will be created for each test class when using this annotation.

Let’s explain **the life cycle of test execution** with an example:



* The method annotated with @BeforeAll is executed once: before all tests. This method needs to be static unless the entire test class is annotated with @TestInstance(Lifecycle.PER\_CLASS).
* The method annotated with @BeforeEach is executed before each test. In our case, it will be executed twice.
* The two methods annotated with @Test are executed independently.
* The method annotated with @AfterEach is executed after each test. In our case, it will be executed twice.
* The method annotated with @AfterAll is executed once: after all tests. **This method needs to be static unless the entire test class is annotated with @TestInstance(Lifecycle.PER\_CLASS).**

To run this test class, you can execute the following from the command line:

mvn clean test -Dtest=”SUTTest”

or for testing a single method:

mvn clean test -Dtest=”SUTTest#testRegularWork”

Note:

If you don’t specify surefire plugin in the POM, maven will pick up the version from its own configurations and they’re not up to date apparently because they’re updated from release to release of Maven itself. So, you better mention the surefire plugin version in the build tag of the POM.

By default, surefire automatically includes all test classes whose name starts with Test, or ends with Test, Tests or TestCase. In case you want to change it if you’re not using intellij to run tests:

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.2.5</version>

<configuration>

<excludes>

<exclude>DataTest.java</exclude>

</excludes>

<includes>

<include>DataCheck.java</include>

</includes>

</configuration>

</plugin>

## The @DisplayName annotation

The @DisplayName annotation can be used over classes and test methods. Typically, this annotation is used for test reporting in IDEs and build tools. The string argument of the @DisplayName annotation may contain spaces, special characters, and even emojis.

It’s usually **a full phrase** that provides significant information about **the purpose of the test**.

A test that does not have an associated display name simply shows the method name.

## The @Disabled annotation

The @Disabled annotation can be used over classes and test methods. It signals that the annotated test class or test method is disabled and should not be executed. If this annotation is applied to a class, it disables all the methods of the test.

You provide a reason for the method or class being disabled within the annotation: @Disabled(“Feature is still under construction.")

## Nested Tests

…………………………………………………………

## Tagged Tests

…………………………………………………………

## Assertions