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

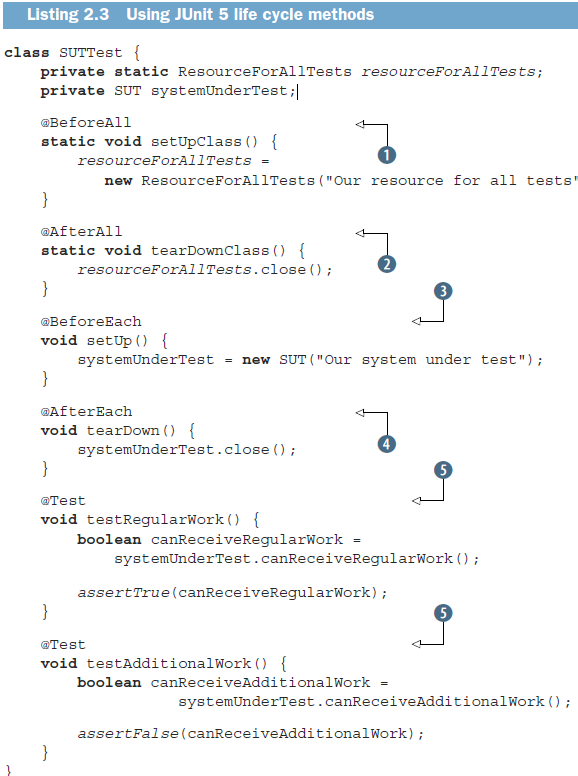
*A life cycle method* is a method that is annotated with @BeforeAll, @AfterAll, @BeforeEach, or @AfterEach.

* @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

…………………………………………………………

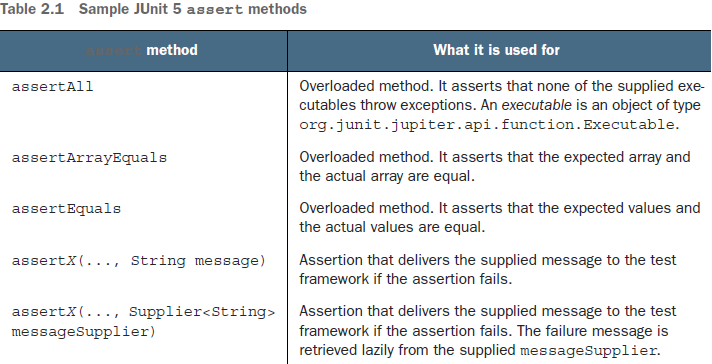
## Assertions

To perform test validation, you use the assert methods provided by the Junit Assertions class. They’re all static methods.

As far as I happen to see, these methods will throw and AssertionError which is a subtype of Error if the assertion fails.

JUnit 5 provides a lot of overloaded assertion methods. It includes many assertion methods **from JUnit 4 and adds a few that can use Java 8 lambdas.**

Lambda expressions treat **functionality as a method argument** and code as data.



Note:

The assertThat() method that works with Hamcrest matchers has been removed. The recommended approach in such a case is to use the Hamcrest MatcherAssert.assertThat() overloaded methods, which are more flexible and in the spirit of the Java 8 capabilities.

DEFINITION *Hamcrest* is a framework that assists with the writing of software tests in JUnit. It supports the creation of customized assertion matchers (*Hamcrest* is an anagram of *matchers*), letting us **define match rules declaratively**. Later in this document, we discuss the capabilities of Hamcrest.

### assertAll()

public static void assertAll(String heading, Executable... executables) throws MultipleFailuresError

* The heading parameter allows us to recognize the group of assertions within the assertAll() methods. If one of the tests fails, this heading will be shown. It acts as a label for the entire group of assertions and It’s useful for example, if you're testing a Person class and you want to validate multiple properties (e.g., name, age, address) in a single test:

assertAll("Person properties validation",

() -> assertEquals("John", person.getName()),

() -> assertEquals(30, person.getAge()),

() -> assertEquals("123 Main St", person.getAddress()) );

* The Executable parameter is a functional interface which throws a Throwable.
* The assertAll method will always check all the assertions that are provided to it, even if some of them fail—**if any of the executables fail, the remaining ones will still be run.** That **is not true for the JUnit 4 approach**: if you have a few assert methods, one under the other, and one of them fails, that failure will stop the execution of the others.
* Failure in this case means that if the executables throw a Throwable.
* If the Throwable is of type OutOfMemoryError it’s unrecoverable and the rest of the executables won’t run.

### Supplier<String> for messages

Thanks to Supplier<String>, the instructions required to create a complex **message aren’t**

**provided in the case of success.**

The advantage of using lambda expressions as arguments for assertion methods is that **all of them are lazily created**, resulting in improved performance.

*assertTrue*(**systemUnderTest**.isVerified(),

() -> "System should be under verification");

**If** the condition at the assertion is fulfilled, meaning the test **succeeded**, the **invocation of the lambda** expression at **does not take place**, which would be impossible if the test were written in the old style.

### AssertTimeOut() and AssertTimeOutPreemptively()

There may be situations in which you expect a test to be executed within a given interval. **JUnit 5** offers an elegant solution for this kind of use case which replaces the JUnit 4 Timeout rule. *The methods* need to check whether the SUT **is performant enough**, **meaning it is executing itsjobs within a given timeout.**

**class** AssertTimeoutTest {

**private** SUT **systemUnderTest** = **new** SUT("Our system under test");

@Test

@DisplayName("A job is executed within a timeout")

**void** testTimeout() **throws** InterruptedException {

**systemUnderTest**.addJob(**new** Job("Job 1"));

*assertTimeout*(*ofMillis*(500), () -> **systemUnderTest**.run(200));

}

@Test

@DisplayName("A job is executed preemptively within a timeout")

**void** testTimeoutPreemptively() **throws** InterruptedException {

**systemUnderTest**.addJob(**new** Job("Job 1"));

*assertTimeoutPreemptively*(*ofMillis*(500),

() -> **systemUnderTest**.run(200));

}

}

**assertTimeout waits until the executable finishes**. The failure message looks something like this: execution exceeded timeout of 500ms by 193ms.

assertTimeoutPreemptively **stops the executable when the time has expired**. The failure message looks like this: execution timed out after 500 ms.

### AssertThrows()

In some situations, you expect a test to be executed and to throw an exception, so you may force the rest to run under inappropriate conditions or to receive inappropriate input. In our example, it is natural that the SUT that tries to run without a job assigned to it will throw an exception. Again, JUnit 5 offers an elegant solution.

Listing 2.14 shows the use of some assertThrows methods, which replace the JUnit 4 ExpectedException rule and the expected attribute of the @Test annotation. All assertions can be made against the returned instance of Throwable. This makes the tests more readable, as we are verifying that the SUT is throwing exceptions: a current job is expected but not found.

**class** AssertThrowsTest {

**private** SUT **systemUnderTest** = **new** SUT("Our system under test");

@Test

@DisplayName("An exception is expected")

**void** testExpectedException() {

*assertThrows*(NoJobException.**class**, **systemUnderTest**::run);

}

@Test

@DisplayName("An exception is caught")

**void** testCatchException() {

Throwable throwable = *assertThrows*(NoJobException.**class**,

() -> **systemUnderTest**.run(1000));

*assertEquals*("No jobs on the execution list!",

throwable.getMessage());

}

}

In this example:

 We verify that the systemUnderTest object’s call of the run method throws NoJobException.

 We verify that a call to systemUnderTest.run(1000) throws a NoJobException, and we keep a reference to the thrown exception in the throwable variable.

 We check the message kept in the throwable exception variable.

## Assumptions

Page 29…

Is assertAll run in multiple threads?